



UNIVERSITÀ POLITECNICA DELLE MARCHE  
FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e dell'Automazione  
Progetto per l'Esame di Ricerca Operativa 2

**Il Problema del Commesso Viaggiatore:  
algoritmi a confronto.**

Docenti:

**Prof. Ferdinando Pezzella**  
**Prof.ssa Ornella Pisacane**

Studente:

**Melvin Mancini**

Introduzione .....	4
1. Storia del TSP .....	4
2. Reti e grafi.....	5
2.1 Definizione e caratteristiche.....	5
3. Il TSP .....	8
3.1 TSP Asimmetrico .....	8
3.2 TSP Simmetrico .....	9
4. Metodi risolutivi ed implementazioni .....	10
4.1 Metodi esatti .....	10
4.1.1 Generazione di vincoli per ATSP .....	10
4.1..2 Implementazione in AMPL del metodo della generazione dei vincoli .....	11
4.1.3 Branch and Bound.....	21
4.1.4 Implementazione in AMPL per il Branch and Bound .....	21
4.2 Metodi euristici .....	23
4.2.1 Nearest Neighbour Algorithm.....	23
4.2.2 Repetitive Nearest Neighbor algorithm .....	24
4.2.3 Implementazione del NN e del RNN .....	25
5. Determinazione ottima dei giri di consegna .....	35
5.1 Il problema .....	35
5.2 Generazione dei vincoli .....	36
5.3 Branch & Bound .....	39
5.3.1 P0 .....	40
5.3.2 P1 (ottenuto da P0).....	42
5.3.3 P2 (ottenuto da P0).....	44
5.3.4 P3 (ottenuto da P1).....	46
5.3.5 P4 (ottenuto da P1).....	47
5.3.6 P5 (ottenuto da P2).....	48
5.3.7 P6 (ottenuto da P2).....	49
5.3.8 P7 (ottenuto da P3).....	50
5.3.9 P8 (ottenuto da P3).....	51
5.3.10 P9 (ottenuto d P3) .....	51
5.3.11 Soluzione ottima .....	52
5.4 Nearest Neighbor e Repetitive Nearest Neighbor .....	54
5.4.1 Risoluzione con il metodo NN .....	54

5.4.2 Risoluzione con il metodo RNN .....	56
6. Conclusioni .....	58

# Introduzione

In questo documento verrà presentato il problema del commesso viaggiatore, alcune tecniche risolutive e un caso reale. In particolare, verranno esposte alcune implementazioni di metodi risolutivi esatti ed euristicci, realizzate rispettivamente attraverso il linguaggio AMPL e Java, analizzando le differenze ed esponendo i relativi vantaggi.

Il problema del commesso viaggiatore consiste nel trovare il percorso più breve che passi in determinati nodi e ritorni al punto di partenza. Per cammino minimo si intende quello di costo minore tra coppie di punti. Diversi sono i parametri che possono essere considerati per la determinazione del percorso minimo. Infatti, con il termine costo, possiamo intendere la distanza tra i vari punti, il tempo necessario per raggiungere i diversi nodi o la quantità di carburante utilizzata. In generale, il TSP è considerato un problema di ottimizzazione definito su un grafo, che può essere orientato o meno, con l'obiettivo di determinare un percorso a costo minimo che visita tutti i nodi una e una sola volta.

## 1. Storia del TSP

Problemi matematici riconducibili al TSP sono stati introdotti nel periodo ottocentesco da parte di Sir William Rowan Hamilton e da Thomas Penyngton Kirkman i quali cercavano una soluzione ottima riguardo un gioco da tavolo. Negli anni successivi, formulazioni matematiche riguardanti il TSP, erano state introdotte soltanto in campi economici e statistici. Solo nel 1954 George Dantzig, Ray Fulkerson e Selmer Johnson formularono un metodo per determinare il percorso ottimo da intraprendere per visitare 49 città statunitensi minimizzando la distanza. Successivamente, il TSP divenne di grande interesse tanto che nel 1977 fu bandito un concorso che prevedeva un premio a chi riusciva a trovare il percorso minimo per visitare le 120 città principali della Germania Federale. Le imprese più grandi riguardo la risoluzione del problema del TSP sono avvenute recentemente. Infatti, nel 1994 Applegate, Bixby, Chvátal e Cook calcolarono il cammino minimo considerando 15112 nodi e nel 2004 riuscirono a determinare il percorso più breve che permetteva di visitare 24978 città della Svezia.

## 2. Reti e grafi

Prima di andare ad analizzare il problema del TSP, è necessario una breve introduzione sui grafi.

La rappresentazione attraverso i grafi è uno strumento ampiamente utilizzato per descrivere problemi appartenenti a diversi ambiti come l'allocazione di una struttura di telecomunicazione (rete stradale o ferroviaria), la gestione e la pianificazione di risorse, la distribuzione di servizi ecc.

Essa ci consente di descrivere graficamente le componenti del sistema, le connessioni e alcune informazioni aggiuntive come la lunghezza di ogni collegamento.

### 2.1 Definizione e caratteristiche

Un grafo viene definito come un insieme di punti, detti nodi o vertici, ed un insieme di linee chiamate archi che collegano i vari nodi. In maniera formale un grafo è definito dalla seguente espressione:

$$G = (V, A)$$

in cui  $V$  è un insieme finito di nodi  $V = \{v_1, \dots, v_n\}$  ed  $A$  è un insieme finito di archi

$A = \{a_1, \dots, a_n\}$ . Un arco o collegamento viene rappresentato scrivendo una coppia di nodi  $(u, v)$  in cui  $u, v$  appartengono a  $V$ .

Un grafo si definisce completo quando ogni nodo è collegamento direttamente con tutti gli altri.

In base alle caratteristiche dei collegamenti un grafo può essere definito:

- Non orientato o simmetrico
- Orientato o asimmetrico.

Nel primo caso il costo per andare da un nodo  $V_i$  a  $V_j$  è uguale al costo per andare da  $V_j$  a  $V_i$  e quindi

$a_k = (v_i, v_j) = (v_j, v_i)$  mentre nel secondo solitamente  $a_k = (v_i, v_j) \neq (v_j, v_i)$ .

Grafo simmetrico completo	Grafo asimmetrico completo
<p>The diagram shows a complete graph with four nodes labeled V1, V2, V3, and V4. Every node is connected to every other node by a straight edge. The edges are labeled a1 through a6. Specifically, there are edges between V1-V2, V1-V3, V1-V4, V2-V3, V2-V4, and V3-V4.</p>	<p>The diagram shows a complete graph with four nodes labeled V1, V2, V3, and V4. Every node has a directed edge to every other node. The edges are labeled a1 through a10. For example, there is a directed edge from V1 to V2 labeled a1, and another from V2 to V1 labeled a2. This pattern repeats for all pairs of nodes.</p>
$G = (V, A)$ $V = \{v_1, \dots, v_n\}$ $A = \{a_1, \dots, a_n\}$ $a_k = (v_i, v_j) = (v_j, v_i)$	$G = (V, A)$ $V = \{v_1, \dots, v_n\}$ $A = \{a_1, \dots, a_n\}$ $a_k = (v_i, v_j) \neq (v_j, v_i)$

Dato un grafo  $G = (V, A)$ , si definisce:

- Percorso: una sequenza di archi consecutivi che connette una serie di nodi
- Cammino: percorso che non contiene nodi ripetuti
- Ciclo: un cammino che inizia e finisce sullo stesso nodo
  - Ciclo semplice: senza ripetizione di vertici
  - Ciclo elementare: senza ripetizione di archi.

Inoltre, dato un generico grafo  $G = (V, A)$ , si dice che due nodi sono connessi se esiste almeno un cammino che li congiunge. In base a questa definizione possiamo classificare un grafo come connesso se ogni coppia di nodi è connessa. Un grafo connesso prende il nome di albero se non contiene alcun ciclo.

Altre nozioni importanti, soprattutto per la classificazione dei cammini, sono la definizione di taglio, grado e star:

- Dato un insieme  $S \subset V$ , un taglio associato a  $S$  si definisce come:

$$\partial(S) = \{(v_i, v_j) : |S \cap \{v_i, v_j\}| = 1\}$$

- Dato un grafo  $G = (V, A)$  si definisce  $\gamma(v)$  cioè star di  $v$ :

$$\gamma(v) = \{v_j : (v, v_j) \in A\}$$

- Dato un grafo  $G = (V, A)$  si definisce  $d(v)$  cioè grado di  $v$ :

$$d(v) = |\gamma(v)|$$

Le definizioni di  $\gamma(v)$  e  $d(v)$  valgono per i grafi non orientati. Nel caso di grafi orientati, in base all'orientazione degli archi, si definiscono anche: semigrado esterno di  $v$  o  $d^+(v)$ , semigrado interno di  $v$  o  $d^-(v)$ , forward star di  $v$  o  $\gamma^+(v)$ , backward star di  $v$  o  $\gamma^-(v)$ .

Le nozioni precedentemente descritte ci permettono di definire il cammino hamiltoniano, nozione di fondamentale importanza per il problema del TSP. Un cammino o ciclo semplice è hamiltoniano se visita tutti i vertici del grafo una e una sola volta. Le condizioni necessarie per l'esistenza di un ciclo hamiltoniano sono le seguenti:

dato un grafo connesso  $G = (V, A)$  con  $d(v) \geq 2 \forall v \in V$  e t.c  $|\partial(S)| \geq 2 \forall S \subset V$

### 3. Il TSP

Il problema del commesso viaggiatore o Travelling Salesman problem (TSP) è un problema di ottimizzazione definito su un grafo. In base alle caratteristiche del grafo possiamo definire due tipi di problemi:

- TSP Asimmetrico
- TSP Simmetrico

In generale, la risoluzione del TSP consiste nel determinare, se esiste, un ciclo hamiltoniano di costo minimo sul grafo. Per motivi di semplicità formuleremo il problema, sia nel caso simmetrico che asimmetrico, considerando il grafo completo. Poiché i modelli e le tecniche risolutive possono essere specializzate in riferimento alle caratteristiche del grafo (orientato o non orientato), in questo articolo formuleremo matematicamente il problema per entrambi i casi, mentre analizzeremo i metodi risolutivi solo nel caso del TSP Asimmetrico.

#### 3.1 TSP Asimmetrico

Il TSP Asimmetrico è definito su un grafo orientato  $G = (V, A)$  in cui indicheremo con  $c_{ij}$  i costi associati a ciascun arco  $(i, j) \in A$ . Matematicamente il problema può essere formulato come segue:

1.  $\text{Min } \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}$
2.  $\sum_{j=1}^n x_{ij} = 1 \forall i = 1 \dots n$
3.  $\sum_{i=1}^n x_{ij} = 1 \forall j = 1 \dots n$
4.  $\sum_{i \in S, j \in S: (i,j) \in A} x_{ij} \leq |S| - 1 \forall S \subset V : 2 \leq |S| \leq |V| - 1$
5.  $x_{ij} \in \{0,1\} \forall (i,j) \in A$

La (1) è la funzione obiettivo la quale minimizza la somma dei costi degli archi selezionati. La (2) e la (3) vengono definiti vincoli di grado e ci permettono di garantire che ogni nodo possieda un arco entrante e un arco uscente. In questo modo garantiamo che ogni nodo venga visitato una e una sola volta. La (4) viene definito vincolo di eliminazione dei sotto-cicli. Questo vincolo ci consente di eliminare tutte le soluzioni che contengono sotto-cicli. Infatti, prendendo un qualunque insieme  $S \subset V$ , se il numero di archi è minore nel numero dei nodi allora evitiamo che ci siano dei sotto-cicli. L'ultimo vincolo (5) dichiara che le variabili decisionali devono essere binarie.

Il gruppo di vincoli (4) può essere formulato in maniera differente:

$$\sum_{i \in S, j \notin S: (i,j) \in A} x_{ij} \geq 1 \forall S \subsetneq V$$

In questo caso preso un qualunque insieme  $S \subsetneq V$  si impone che ci sia almeno un arco uscente dall'insieme. Questo tipo di vincolo è stato utilizzato per l'implementazione del metodo esatto che verrà analizzato successivamente.

### 3.2 TSP Simmetrico

Il TSP simmetrico è definito su un grafo non orientato  $G = (V, E)$  con costi  $c_e$  associati ad ogni arco  $e \in E$ . Il problema può essere trasformato in TSP Asimmetrico trasformando il grafo non orientato in uno orientato. La simmetria dei costi permette di avere un modello e metodi risolutivi migliorabili poiché non è rilevante l'ordine con cui si visitano coppie di nodi. Il modello matematico può essere rappresentato nella seguente maniera:

1.  $\text{Min } \sum_{e \in E} c_e x_e$
2.  $\sum_{e \in \partial(v)} x_e = 2 \forall v \in V$
3.  $\sum_{e \in \partial(S)} x_e = 2 \forall S \subset V : 3 \leq |S| \leq |V| - 1$
4.  $x_e \in \{0,1\} \forall e \in E$

La funzione obiettivo minimizza il costo degli archi selezionati. La (2) sono i vincoli di grado e in questo caso è sufficiente scegliere esattamente due spigoli in  $\partial(v)$  senza esplicitare il verso di percorrenza. La (3) rappresenta i vincoli di eliminazione dei sotto-cicli e la (4) impone che le variabili decisionali siano binarie.

## 4. Metodi risolutivi ed implementazioni

Il TSP è un problema di ottimizzazione definito su un grafo e può essere classificato come un problema NP-HARD la cui complessità computazione è dell'ordine esponenziale. La risoluzione di problemi di questo tipo può essere effettuata attraverso metodi esatti o tramite tecniche euristiche. In questo documento verranno analizzati due metodi esatti (generazione di vincoli, Branch & Bound) e due metodi euristici (Nearest Neighbor e Ripetitive Nearest Neighbor), presentando le loro caratteristiche e descrivendo il procedimento utilizzato per implementarle. Per la realizzazione dei metodi esatti sono stati creati due script nel linguaggio AMPL, mentre per quanto riguarda le tecniche euristiche è stato realizzato un programma nel linguaggio di programmazione JAVA.

### 4.1 Metodi esatti

Risolvere il problema del TSP applicando un metodo esatto significa seguire il modello matematico e introdurre un numero di vincoli pari a  $2|V|$  (per i vincoli di grado) più il numero dei possibili sotto-cicli che è dell'ordine esponenziale. Il modello, pertanto, non può essere utilizzato direttamente per la soluzione del problema, a meno che il numero dei nodi è tale da poter esPLICITARE tutti i vincoli. Per tale motivo, è necessario introdurre degli approcci risolutivi in grado di gestire un numero esponenziale di vincoli.

Le tecniche risolutive esatte per il TSP sono:

- Generazione di vincoli
- Branch and Bound

In questo articolo verranno discussi entrambi e successivamente verranno presentate le rispettive implementazioni tramite il linguaggio AMPL. Attraverso un esempio verrà analizzato il procedimento di queste tecniche e il funzionamento degli algoritmi che permettono di realizzarle.

#### 4.1.1 Generazione di vincoli per ATSP

L'idea base di questa tecnica è quella di non considerare tutti i vincoli di eliminazione dei sotto-cicli, ma di aggiungere di volta in volta i vincoli che permettono di eliminare i sotto-cicli individuati. In generale, secondo un approccio iterativo, si risolve il problema rilassato e successivamente si individuano i sotto-cicli presenti nella soluzione determinata. Alle iterazioni successive si aggiungono i vincoli che permettono l'eliminazione dei sotto-cicli individuati. Nel momento in cui si ottiene una soluzione in cui non sono più presenti sotto-cicli, allora corrisponderà ad un ciclo hamiltoniano di costo minimo.

Eliminando dal modello i vincoli di eliminazione dei sotto-cicli, otteniamo un problema di assegnamento che può essere risolto in maniera efficiente attraverso il metodo del simplex o il

metodo ungherese. In generale, la soluzione che si ottiene potrebbe contenere sotto-cicli e quindi inammissibile per il problema TSP. In ogni caso la soluzione determinata rappresenta un lower-bound della soluzione ottima ammissibile del TSP, poiché è stata ricavata da un modello con meno vincoli. Se la soluzione ottenuta contiene dei sotto-cicli, si riformula il problema aggiungendo i vincoli che permettono di eliminare i sotto-cicli individuati. L'aggiunta di un nuovo vincolo fa perdere la totale unimodularità della matrice dei vincoli e quindi il problema non è più riconducibile ad un semplice problema di assegnamento e di conseguenza non più risolvibile in maniera efficiente.

I passi che ci permettono di implementare il metodo della generazione di vincoli sono i seguenti:

1. Si eliminano, dal modello matematico iniziale, tutti i vincoli di eliminazione dei sotto-cicli e quelli di interezza e si risolve il problema così ottenuto.
2. Se la soluzione corrente non contiene sotto-cicli allora essa corrisponderà alla soluzione ottima e quindi ci fermeremo.
3. Si individuano i sotto-cicli nella soluzione corrente.
4. Si aggiunge al modello iniziale i vincoli che permettono l'eliminazione di almeno un sotto-ciclo tra quelli individuati e i vincoli di interezza delle variabili coinvolte nel ciclo.
5. Si risolve il problema ottenuto e si ritorna al passo 2.

Notiamo che al limite, l'algoritmo garantisce la determinazione della soluzione ottima e ammissibile aggiungendo tutti i vincoli di eliminazione dei sotto-cicli. In generale, non tutti i vincoli di eliminazione dei sotto-cicli sono necessari per la determinazione della soluzione ottima, ma solo un sotto-insieme di essi. Inoltre, ad ogni iterazione bisogna risolvere un problema di programmazione lineare intera e ciò comporta una complessità computazionale esponenziale. Questa situazione può essere risolta aggiungendo i vincoli di interezza delle sole variabili interessate nei sotto-cicli individuati. In questo modo, oltre a limitare il numero di variabili intere da gestire, garantiamo che anche le altre variabili risultano essere binarie.

#### **4.1..2 Implementazione in AMPL del metodo della generazione dei vincoli**

In questa sezione verrà presentata l'implementazione del metodo della generazione di vincoli attraverso il linguaggio AMPL (A Mathematical Programming Language). Quest'ultimo è un linguaggio ad alto livello che permette di risolvere complicati problemi di programmazione matematica.

Invece di eseguire l'algoritmo precedentemente esposto, andando di volta in volta a determinare i sotto-cicli e inserendo “a mano” ad ogni nuova iterazione i corrispondenti vincoli di eliminazione, è stato creato uno script in AMPL. Lo script risolve in maniera iterativa il problema dato, aggiornando ad ogni iterazione il modello matematico. Il programma costruito riesce

automaticamente a determinare i sotto-cicli dalla soluzione ottenuta e successivamente riformula il nuovo problema inserendo i vincoli di eliminazione dei sotto-cicli. Nel momento in cui si determina la soluzione ottima ammissibile il programma si interrompe. Per mostrare il funzionamento e i singoli passi che l'algoritmo compie, ad ogni iterazione vengono stampate a video alcuni parametri e variabili.

L'intero progetto è costituito essenzialmente da 3 files:

- modello.mod : file contenente il modello matematico del problema
- dataTSP.dat : file contenente i dati e i parametri di input
- algoritmo.run : lo script che permette di eseguire l'algoritmo di generazione di vincoli.

In seguito verranno presentati i vari moduli, opportunamente commentati, che ci permettono di capire il funzionamento dell'algoritmo costruito.

Il file modello.mod contiene la descrizione ad alto livello della formulazione matematica del problema. In generale, è stato implementato il modello matematico seguendo le regole grammaticali del linguaggio AMPL.

$$\begin{aligned}
 & \text{Min } \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\
 & \sum_{j=1}^n x_{ij} = 1 \quad \forall i = 1 \dots n \\
 & \sum_{i=1}^n x_{ij} = 1 \quad \forall j = 1 \dots n \\
 & \sum_{i \in S, j \notin S: (i,j) \in A} x_{ij} \geq 1 \quad \forall S \subsetneq V
 \end{aligned}$$

```

# Modello per TSP

# Numero dei nodi del grafo
param n > 0, integer;

# Insieme dei nodi
set V := 1..n;

# I costi associati ai vari archi
param c{V,V} >= 0;

# Parametro contatore che consente di tener conto del numero dei
sottocicli
param numerocicli >= 0, integer, default 0;

# Insieme i cui elementi sono i nodi che corrispondono ai sotto-cicli
# individuati. Questo set di elementi ci consente di generare i vincoli
# di eliminazione dei sotto-cicli individuati. Al suo interno ci
# saranno tanti elementi quanti sono i cicli (numerocicli)
set ciclo{1..numerocicli};

# Definizioni delle variabili come variabili binarie
var x{V,V} binary;

# Funzione obiettivo
minimize costociclo : sum{i in V, j in V : i != j} c[i,j]*x[i,j];

# Vincoli di grado
subject to successore {i in V} : sum{j in V : i != j} x[i,j] = 1;

subject to predecessore {j in V} : sum{i in V : i != j} x[i,j] = 1;

#Vincoli di eliminazione dei sotto-cicli
subject to nocicli {k in 1..numerocicli} :
    sum{i in ciclo[k], j in V diff ciclo[k]} x[i,j] >= 1;

```

Da notare è l'implementazione dei vincoli di eliminazione dei sotto-cicli secondo le regole grammaticali dell'AMPL.

```

#Vincoli di eliminazione dei sotto-cicli
subject to nocicli {k in 1..numerocicli} :
    sum{i in ciclo[k], j in V diff ciclo[k]} x[i,j] >= 1;

```

Poiché l'insieme ‘ciclo’ contiene tutti i sotto-cicli individuati alle iterazioni precedenti, si considera ogni volta un elemento differente fino a che non sono stati scansionati tutti. Ogni elemento di

‘ciclo’ è un insieme di nodi che formano un sotto-ciclo determinato dalla soluzione. In pratica si impone che la somma delle variabili  $x[i][j]$  per ogni nodo i del sotto-ciclo (i in  $ciclo[k]$ ) e per ogni nodo j che non fa parte del sotto-ciclo (j in  $V \setminus ciclo[k]$ ), deve essere maggiore uguale a uno. In questo modo imponiamo che ci sia almeno un arco uscente tra i nodi del sotto-ciclo e poiché sono presenti i vincoli di grado che permettono di impostare per ogni nodo un arco entrante e uno uscente, si risolverà il problema “spezzando” il sotto-ciclo. Un semplice esempio rende il concetto più chiaro: consideriamo un grafo con 4 nodi e all’interno dell’insieme ‘ciclo’ un elemento è costituito dai seguenti nodi che formano un sotto-ciclo (1,4,1). Il problema che verrà risolto, oltre ai vincoli di grado, avrà il seguente vincolo di eliminazione del sotto-ciclo individuato:

$$x_{12} + x_{13} + x_{42} + x_{43} \geq 1$$

Il file algoritmo.run è il cuore del progetto. Esso è costituito da un insieme di regole grammaticali del linguaggio AMPL che permette di implementare il metodo di generazione di vincoli. Le parti interessati dell'implementazione risiedono nei loop per la determinazione dei sotto-cicli partendo dalla soluzione data e dalla generazione dei corrispondenti vincoli di eliminazione. In particolare, il codice è stato ottimizzato facendo in modo che i vincoli di eliminazione dei sotto-cicli non siano ridondanti.

```
# algoritmo.run
# Usare il solve di tipo cplex e non stampare
# i messaggi del solutore numerico
reset;
option solver cplex;
option solver_msg 0;
# leggi modello e i dati
model modello.mod;
data dataTSP.dat;

# Impongo la variabile numerocicli uguale a zero
# in modo tale che all'inizio il problema viene
# risolto senza vincoli di eliminazione dei sotto-cicli.
let numerocicli := 0;

# Strutture dati per la determinazione dei cicli
param nodosuccessore{V} >= 0, integer;
param nodocorrente >= 0, integer;

# Parametro che tiene conto delle iterazioni effettuate
param numeroIterazione, integer;

# Parmanetro che ci permette di uscire dallo script
param termination binary;

# Le seguenti variabili sono degli indici che ci
# permettono di indirizzare i sotto-cicli individuati
# alle varie iterazioni.
param indice_cicli_precedenti;
param nuovi_cicli;

# Variabile che permette di definire se bisogna
# aggiungere il sotto-ciclo individuato all'insieme
# dei sotto-cicli (ciclo).
param aggiungi_ciclo;

# Insieme i cui elementi sono nodi corrispondenti
# ad un sotto-ciclo individuato
```

```

set candidato;
# Insieme i cui elementi sono i nodi
#corrispondenti alla soluzione ottima.
set cicloHamiltoniano;

# Impongo la variabile termination uguale a 0
let termination := 0;
# Impongo il numero delle iterazioni a zero.
let numeroIterazione := 0;

# Algoritmo: risolvi il modello senza vincoli
# di rottura dei sottocicli, trova un sottociclo,
# aggiungi il vincolo corrispondente, e quando
# non esistono sottocicli propri, esci.
# Il ciclo finale determinato equivale alla
# soluzione del problema.
repeat while (termination = 0) {
    let numeroIterazione := numeroIterazione +1;
    printf "\n";
    printf "Iterazione numero %d", numeroIterazione;
    printf "\n";
    printf "La funzione obiettivo:\n";
    expand costociclo;
    printf "\nVincoli di grado:\n";
    expand successore;
    expand predecessore;
    printf "Vincoli di eliminazione dei sottocicli:\n";
    expand nocicli;

    # Soluzione del problema generato
    # Alla prima iterazione, non ci saranno vincoli
    # di eliminazione di sottocicli
    solve;

    # Salvataggio dell'indice dell'ultimo sottociclo
    # inserito all'iterazione precedente
    let indice_cicli_precedenti := numerocicli;

    # Inizializzazione del contatore di nuovi cicli
    # individuati
    let nuovi_cicli := 0;

    # Per ogni nodo, cerco il successore all'interno
    # dei percorsi (cicli) determinati dalla soluzione
    for {i in V} {
        let nodosuccessore[i] := sum{j in V : j != i} j * x[i,j];
    }
}
```

```

}

# Ricerca di sottocicli.
# Per ogni nodo determino il suo successore.
# Attraverso un opportuno loop riusciamo a
# determinare i vari sotto-cicli (memorizzati
# di volta in volta nel parametro candidato)
# e successivamente si valuta se bisogna aggiungerlo
# o meno ai sotto-cicli determinati precedentemente.
for {i in V} {
    let nodocorrente := i;
    let candidato := {};

    # Partendo da un nodo, determina l'eventuale
    # percorso per tornare a se stesso.
    # Metodo per determinare i sottocicli (o il
    # ciclo hamiltoniano)
    repeat {
        let candidato := candidato union {nodocorrente};
        let nodocorrente := nodosuccessore[nodocorrente];
    } until (nodocorrente = i);

    # Aggiunta del nuovo percorso all'insieme dei
    # sottocicli evitando percorsi duplicati. In questo
    # modo evitiamo di aggiungere successivamente
    # vincoli di eliminazione di sotto-cicli ridondanti.
    let aggiungi_ciclo := 1;
    for {d in 1..numerocicli} {
        # Se, tra i sottocicli determinati precedentemente
        # della stessa cardinalita' del nuovo ciclo
        # determinato, la differenza degli elementi
        # e' un insieme vuoto, allora i due percorsi
        # sono uguali.
        # Due percorsi uguali andrebbero a creare vincoli
        # duplicati e non vanno inseriti.
        if(card(ciclo[d]) = card(candidato) and
            card(ciclo[d] diff candidato) = 0) then
        {
            let aggiungi_ciclo := 0;
        }
    }

    # Aggiungo il ciclo appena trovato (nel caso in cui
    # non era presente nelle iterazioni precedenti).
    # Aggiungendo un ciclo si procedera' a risolvere il
    # problema con un vincolo in piu'.
}

```

```

    if (aggiungi_ciclo = 1) then {
        # Aumento il numero dei cicli, espandendo di
        # conseguenza l'insieme dei sottocicli
        let numerocicli := numerocicli + 1;
        # Assegno il ciclo appena trovato in fondo
        # all'insieme di sottocicli
        let ciclo[numerocicli] := candidato;
        # Incremento il contatore dei cicli trovati in
        # riferimento a questa iterazione
        let nuovi_cicli := nuovi_cicli + 1;
    }
}

printf "\n\nLa soluzione determinata: \n";
display x;
printf "I sottocicli individuati: \n";
for {i in (indice_cicli_precedenti + 1)..(indice_cicli_precedenti
+ nuovi_cicli)} {
    display ciclo[i];
}
printf "Il valore della funzione obiettivo: %f ", costociclo;
printf "\n \n";

# Verifica se si puo terminare.
# Se, tra i nuovi cicli determinati in questa iterazione,
# esiste anche un solo percorso che tocca un numero di nodi
# uguale a n (numero totale di nodi) allora abbiamo
# determinato una soluzione ottima. Altrimenti e'
# necessario effettuare un'altra iterazione, poiche'
# la soluzione non e' ammissibile.
for {i in (indice_cicli_precedenti + 1)..(indice_cicli_precedenti
+ nuovi_cicli)} {
    if(card(ciclo[i]) >= n) then {
        let termination := 1;
        let cicloHamiltoniano := ciclo[i];
        break;
    }
}
printf "\nLa soluzione finale determinata:\n";
display x;
printf "Il ciclo hamiltoniano minimo determinato:\n ";
display cicloHamiltoniano;
printf "\nIl costo del ciclo hamiltoniano minimo: %f\n", costociclo;

```

Diverse sono le parti interessanti di questo algoritmo, dalle istruzioni che permettono di determinare i sotto-cicli, alla decisione di aggiungerli o meno all'insieme dei cicli di cui generare i vincoli. Per l'identificazione dei sotto-cicli è stata utilizzata una tecnica particolare. Il primo passo è stato determinare, per ogni nodo, il suo successore attraverso il seguente for:

```
for {i in V} {
    let nodosuccessore[i] := sum{j in V : j != i} j * x[i,j];
}
```

Successivamente, per ogni nodo, si cerca un percorso che inizia e finisce nel vertice in esame:

```
let nodocorrente := i;
let candidato := {};
repeat {
    let candidato := candidato union {nodocorrente};
    let nodocorrente := nodosuccessore[nodocorrente];
} until (nodocorrente = i);
```

All'interno dell'insieme ‘candidato’ sono presenti i nodi che formano un ciclo che inizia e finisce nel vertice ‘*i*’. Per valutare se generare o meno il vincolo di eliminazione del sotto-ciclo identificato, si confronta l'insieme ‘candidato’ con tutti i sotto-cicli determinati alle iterazione precedenti (tutti gli elementi dell'insieme ‘ciclo’).

```
let aggiungi_ciclo := 1;
for {d in 1..numerocicli} {
    if(card(ciclo[d]) = card(candidato) and
        card(ciclo[d] diff candidato) = 0) then
    {
        let aggiungi_ciclo := 0;
    }
}
```

Confrontando ‘candidato’ con tutti i sotto-cicli determinati alle iterazioni precedenti (*ciclo*[*d*]) e che hanno la stessa cardinalità, si valuta la loro differenza. Se almeno una volta le espressioni all'interno dell'if risultano vere, allora ‘candidato’ è uguale ad un sottociclo determinato alle iterazioni precedenti. In questo caso risulterà inutile riformulare il problema aggiungendo il vincolo di eliminazione del sotto-ciclo ‘candidato’ poiché risulterà ridondante. Un semplice esempio rende il concetto più chiaro: considerando due iterazioni differenti una volta con *i*=1 e una volta con *i*=6 verranno identificati rispettivamente il sotto-ciclo (1,4,1) e (4,1,4). Generare i vincoli di eliminazione dei sotto-cicli per i due insiemi significa produrre le stesse disequazioni. Per semplicità ipotizziamo che nel grafo ci siano 4 nodi.

$$x_{12} + x_{13} + x_{42} + x_{43} \geq 1$$

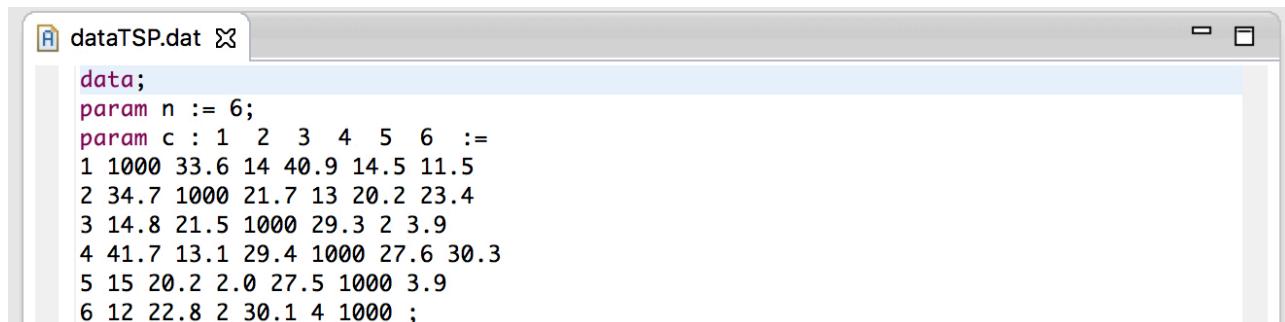
$$x_{42} + x_{43} + x_{12} + x_{13} \geq 1$$

Nel caso in cui, usciti dal for, la variabile ‘aggiungi\_ciclo’ è uguale a 1 significa che le espressioni all’interno dell’if non sono mai risultate vere e quindi bisogna aggiungere il sotto-ciclo ‘candidato’ all’insieme dei sotto-cicli di cui generare i vincoli di eliminazione. Perciò aggiungeremo ‘candidato’ in coda all’insieme ‘ciclo’. La variabile ‘numerocicli’ tiene conto del numero di sotto-cicli presenti nell’insieme ‘ciclo’ e perciò viene incrementata di uno e utilizzata come indice.

```
if (aggiungi_ciclo = 1) then {
    let numerocicli := numerocicli + 1;
    let ciclo[numerocicli] := candidato;
    let nuovi_cicli := nuovi_cicli + 1;
}
```

All’iterazione successiva si risolverà il problema riformulando il modello matematico e aggiungendo i vincoli di eliminazione dei sotto-cicli identificati.

Il file dataTSP.dat contiene i dati e i parametri di input in riferimento al problema analizzato. In seguito un esempio del file dataTSP.dat



```
data;
param n := 6;
param c : 1 2 3 4 5 6 :=
1 1000 33.6 14 40.9 14.5 11.5
2 34.7 1000 21.7 13 20.2 23.4
3 14.8 21.5 1000 29.3 2 3.9
4 41.7 13.1 29.4 1000 27.6 30.3
5 15 20.2 2.0 27.5 1000 3.9
6 12 22.8 2 30.1 4 1000 ;
```

#### 4.1.3 Branch and Bound

Il metodo Branch & Bound è un metodo di ricerca intelligente nello spazio della soluzione di un problema di programmazione matematica. Si tratta di un metodo iterativo che permette di ricavare la soluzione esatta scomponendo il problema padre in tanti sotto-problemi detti figli. In generale, si rivolve il problema in esame come un problema di assegnamento eliminando i vicoli di eliminazione dei sotto-cicli. Se la soluzione non contiene sotto-cicli allora sarà la soluzione ottima e non si generano sotto-problemi. Altrimenti, si creano i sotto-problemi introducendo i vincoli di eliminazione dei sotto-cicli identificati nel problema padre. Eliminando dal modello tutti i vincoli di eliminazione dei sotto-cicli, in qualunque caso, il valore della soluzione ottenuta è un limite inferiore (lower bound) della soluzione ottima ammissibile poiché ottenuta in corrispondenza di un problema meno vincolato. L'aggiunta di un vincolo di eliminazione del sotto-ciclo potrebbe far perdere la totale unimodularità della matrice dei costi. Per questo motivo si preferisce modificare la matrice dei costi in modo opportuno. Fissando determinati valori dei costi associati ai vari archi, si forza l'uso dell'arco voluto. Per imporre  $x_{ij} = 0$  basta settare  $c_{ij} = +\infty$ . Per imporre  $x_{ij} = 1$  basta porre  $c_{ij} = +\infty \forall j \neq i$ . Nel caso in cui otteniamo una soluzione con un sotto-ciclo che contiene  $n$  archi, si dovrebbero generare  $2^n - 1$  sotto-problemi e ciò comporterebbe un numero di figli molto elevato. Per tale motivo data una soluzione con un sotto-ciclo  $C$ , si sceglie un arco  $(i, j) \in C$  e si generano  $|C|$  nodi figli: un figlio con il vincolo  $x_{ij} = 0$  e per ciascuno degli  $|C| - 1$  archi  $(u, v) \in C, (u, v) \neq (i, j)$  un nodo con i vincoli  $x_{ij} = 1$  e  $x_{uv} = 0$ .

#### 4.1.4 Implementazione in AMPL per il Branch and Bound

Per quanto riguarda l'implementazione in AMPL del metodo Branch and Bound è stato costruito, secondo le regole grammaticali dell'AMPL, il modello matematico del TSP salvato in un file chiamato `modelloB.mod`.

Successivamente è stato implementato uno script che non fa altro che prendere in considerazione il modello e il file contenente i dati, per poi procedere nella risoluzione del problema. Ad ogni iterazione verranno identificati i vari sotto-cicli e si modificherà di conseguenza la matrice dei costi per poi risolvere i vari sotto-problemi.

\*modelloB.mod

```

# Modello per TSP
# Senza vincoli per l'eliminazione dei sottocicli

# Numero dei nodi del grafo
param n > 0, integer;

# Insieme dei nodi
set V := 1..n;

# Matrice dei costi
param c{V,V} >= 0;

# Definizioni delle variabili come variabili binarie
var x{V,V} binary;

# Funzione obiettivo
minimize costociclo : sum{i in V, j in V : i != j} c[i,j]*x[i,j];

# Vincoli di grado
subject to successore {i in V} : sum{j in V : i != j} x[i,j] = 1;

subject to predecessore {j in V} : sum{i in V : i != j} x[i,j] = 1;

```

In questo caso i vincoli di eliminazione dei sotto-cicli sono stati omessi poiché andiamo ad agire direttamente nella matrice dei costi. In seguito, viene mostrato lo script che permette di agevolare la stampa delle soluzioni dei vari sotto-problemi identificati.

branch\_and\_bound.run

```

# algoritmo.run
# Usare il solve di tipo cplex e non stampare
# i messaggi del solutore numerico
reset;
option solver cplex;
option solver_msg 0;
# leggi modello e i dati
model modelloB.mod;
data data1.dat;

# Stampa a video la funzione obiettivo
expand costociclo;
# Stampa a video dei vincoli di grado
expand successore;
expand predecessore;
# Richiamo il metodo risolutivo di AMPL per ricavare
# la soluzione del problema
solve;
# Stampa a video della soluzione trovata
display x;
# Stampa a video del valore della rispettiva funzione
# obiettivo
display costociclo;

```

## 4.2 Metodi euristici

Data la complessità di alcuni problemi di ottimizzazione è spesso necessario sviluppare algoritmi euristici. Quest'ultimi sono particolari metodi che non garantiscono la determinazione della soluzione ottima ma permettono di ottenere delle “buone” soluzioni ammissibili. La costruzione di algoritmi euristici efficaci richiedono un’attenta analisi del problema, estraendo le informazioni utili e una buona conoscenza delle tecniche algoritmiche disponibili. Infatti, dato un problema con delle determinate caratteristiche, esistono diverse tecniche generali che possono essere utilizzate.

Le principali classi algoritmiche utili per la realizzazione di metodi euristici per i problemi di ottimizzazione combinatoria si dividono essenzialmente in due tipologie:

- algoritmi Greedy
- algoritmi di ricerca locale.

I primi sono orientati alla costruzione graduale di un circuito. Infatti, gli algoritmi Greedy o voraci determinano una soluzione attraverso una sequenza di decisioni “localmente ottime”, senza mai tornare indietro modificando le decisioni precedenti. Gli algoritmi di ricerca locale invece, si basano su un’idea semplice ed intuitiva: partendo da una soluzione ammissibile, si esaminano le soluzioni ad essa vicine in cerca di una migliore (cioè con un valore migliore della funzione obiettivo).

Per quanto riguarda il problema del TSP gli algoritmi euristici si dividono in:

- costruttivi
- migliorativi.

I primi permettono di costruire in maniera graduale un circuito basandosi sulla soluzione ottima locale, mentre i secondi partendo da una soluzione iniziale determinano altre soluzioni ad essa “vicine” che migliorano la funzione obiettivo.

In questo documento analizzeremo i seguenti algoritmi euristici costruttivi per il TSP:

- il Nearest Neighbor
- il Repetitive Nearest Neighbor.

### 4.2.1 Nearest Neighbour Algorithm

L’algoritmo Nearest Neighbor si basa sull’idea di cercare di ottimizzare il percorso unendo ogni volta i nodi più vicini, cioè nodi i cui archi hanno un costo minore. L’algoritmo parte da un nodo e comincia a percorrere l’arco di costo minimo che incide sul quel nodo. In generale, si sceglierà come nodo successivo, il vertice il cui arco presenta un valore di costo minore degli altri. La strategia di questo algoritmo è che scegliendo ogni volta la direzione più economica si possa ottenere la soluzione complessivamente migliore. Il problema fondamentale di questo algoritmo è l’impossibilità di prevedere gli sviluppi finali che ogni scelta comporta. Infatti, ogni volta che viene

effettuata una scelta questa non contribuisce solo al costo complessivo, ma influenza tutte le scelte future. Questo significa che scegliendo all'inizio direzioni molto economiche, possono rendere necessarie scelte molto costose nei passi successivi.

Possiamo elencare i passi dell'algoritmo considerando  $S$  l'insieme dei nodi inseriti nella soluzione e con  $S' = V - S$  l'insieme dei nodi non ancora visitati.

1. Inizializzazione

Si sceglie qualsiasi nodo  $d$  e si pone  $t = d$ ,  $S = \{d\}$ ,  $S' = V - \{d\}$  e  $z(S) = 0$

2. Aggiunta di un nodo alla soluzione

Si individua il nodo  $k$  più vicino al nodo  $t$ . In maniera formale  $k: c_{tk} = \min_{j \in S'} c_{tj}$ .

Successivamente si collega  $k$  a  $t$  e si pone  $S = S \cup \{k\}$ ,  $S' = S' - k$  e  $z(S) = z(S) + c_{tk}$

3. Criterio di arresto

Se  $S' = \{\emptyset\}$  si connette  $k$  con il nodo iniziale  $d$  in modo tale da chiedere il circuito.

Altrimenti si pone  $t = k$  e si ritorna al passo 2.

Tale algoritmo funziona sia per il ATSP che per il STSP. Il limite fondamentale di questo algoritmo è che la soluzione dipende fortemente dal nodo iniziale scelto.

#### **4.2.2 Repetitive Nearest Neighbor algorithm**

Abbiamo visto precedentemente come la soluzione offerta dal Nearest Neighbor dipenda fortemente dalla scelta del nodo iniziale. Un metodo alternativo per valutare tutti i cammini hamiltoniani, considerando ogni volta un nodo differente di partenza, è il Repetitive Nearest Neighbor. In sostanza l'algoritmo prevede di ripetere il NN tante volte quanti sono i nodi del grafo. Una volta valutati tutti i cammini hamiltoniani con i relativi costi si sceglierà il percorso con il relativo valore della funzione obiettivo migliore. Possiamo elencare i passi dell'algoritmo:

1. Prendere  $X$  un nodo del grafo. Applicare il Nearest Neighbor considerando il  $X$  come nodo di partenza e ricavare il cammino hamiltoniano con il relativo valore della funzione obiettivo.
2. Ripetere il processo considerando come nodo di partenza ogni nodo del grafo.
3. Scegliere, come soluzione, il cammino hamiltoniano con il relativo valore della funzione obiettivo migliore.

#### **4.2.3 Implementazione del NN e del RNN**

In questa sezione verrà presentato il programma in Java costruito per l'implementazione dei metodi euristici precedentemente descritti. Il software creato offre una semplice interfaccia in cui l'utente può inserire i dati del problema e successivamente mostra la soluzione o le soluzioni determinate.

Essenzialmente i parametri di input da inserire sono:

- Il numero dei nodi presenti nel grafo che si vuole considerare
- La scelta del metodo da utilizzare (NN o RNN)
  - Nel caso del NN verrà chiesto anche il nodo di partenza.
- La matrice dei costi.

L'esecuzione del programma con la relativa interfaccia verrà mostrata nei capitoli successivi. In seguito, verrà presentato il codice in Java e le relative parti dell'implementazione dei metodi euristici.

Essenzialmente il programma è composto da due classi:

- Main: la classe che contiene il metodo principale richiamato non appena il programma viene mandato in esecuzione. Al suo interno sono stati definiti metodi e variabili che permettono di memorizzare i dati inseriti dall'utente. Inoltre, si occupa di creare gli oggetti utilizzati per richiamare i metodi relativi alle implementazioni NN e RNN.
- TSP: questa classe contiene gli attributi, i metodi e le strutture dati necessarie per ricavare la soluzione del problema.

Il codice che verrà mostrato contiene numerosi commenti esplicativi che permettono una completa descrizione delle singole parti.

In seguito viene presentato il codice della classe Main.java

```
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * La classe Main è la classe principale. La sua istanza viene creata non appena il programma viene
 * mandato in esecuzione. Al suo interno sono presenti metodi e strutture dati che consentono di
 * memorizzare i dati di input inseriti dall'utente.
 */
public class Main {

    /**
     * Il main è il primo metodo che viene richiamato. In questa sezione ci sono gli oggetti e le
     * variabili che permettono di prelevare i dati di input inseriti dall'utente e successivamente
     * verranno richiamati i metodi relativi alla determinazione della soluzione in base alla
     * tecnica scelta.
     *
     * @param args
     */
    public static void main(String[] args) {
        // Variabili e strutture dati richiesti per la risoluzione del problema
        int numeroNodi;
        int nodoPartenza = 0;
        int metodo;
        double matrix[][];
        // Oggetto per prelevare i dati di input inseriti dall'utente
        Scanner scanner = null;
        // In questa parte è presente il codice che permette di prelevare i dati di input da schermo
        // e successivamente vengono chiamati i metodi relative alle tecniche euristiche
        // implementate
        try {
            System.out.println("Inserisci il numero di nodi presenti nel grafo: ");
            scanner = new Scanner(System.in);
            numeroNodi = scanner.nextInt();
            System.out.println("Scegli l'algoritmo che vuoi applicare: \n 1-
Nearest Neighbour \n" +
                    " 2- Ripetitive Nearest Neighbour");
            scanner = new Scanner(System.in);
            metodo = scanner.nextInt();

            // L'if in esame serve nel caso in cui l'utente abbia scelto il NN e di conseguenza
            // verrà richiesto anche il
            // di partenza
            if (metodo == 1) {
                System.out.println("Inserisci il nodo da cui partire: ");
                scanner = new Scanner(System.in);
                nodoPartenza = scanner.nextInt();
                nodoPartenza = nodoPartenza - 1;
            }

            // Matrix contiene la matrice dei costi. La variabile matrix viene riempita
            // richiamando il metodo acquisisciMatrice() descritto successivamente
            matrix = acquisisciMatrice(numeroNodi);
            TSP tsp = new TSP(numeroNodi, matrix);

            // L'if in esame serve per richiamare il metodo opportuno in base alla tecnica euristica
            // scelta.
            if (metodo == 1) {
                System.out.println("La sequenza determinata: ");
                // La riga di codice successiva richiama il metodo nearestNeighbor() dell'oggetto
                // tsp precedentemente creato
                tsp.nearestNeighbor(nodoPartenza);
            } else {
                // La riga di codice successiva richiama il metodo repetitiveNearestNeighbor()
                // dell'oggetto tsp precedentemente creato
            }
        }
    }
}
```

```

        tsp.repetitiveNearestNeighbor();
    }
    scanner.close();
} catch (InputMismatchException inputMismatch) {
    System.out.println("Errore di input");
} catch (NullPointerException e) {
    e.printStackTrace();
}
}

/**
 * Questo metodo prende come paramentro il numero di nodi inseriti dall'utente e successivamente
 * consente di prelevare i dati relativi alla matrice dei costi
 *
 * @param nodi
 * @return
 */
public static double[][] acquisisciMatrice(int nodi) {
    Scanner scanner = null;
    scanner = new Scanner(System.in);
    double matrice[][] = new double[nodi][nodi];
    System.out.println("Inserisci la matrice dei costi: ");

    // Ogni valore inserito dall'utente viene inserito nell'opportuna posizione della matrice
    for (int i = 0; i < nodi; i++) {
        for (int j = 0; j < nodi; j++) {
            matrice[i][j] = scanner.nextDouble();
        }
    }
    // dato che il TSP non permette di avere nodi con autoanelli vengono sostituiti tutti i
    // valori della diagonale della matrice con lo 0. Questo procedimento viene effettuato
    // per evitare errori nella risoluzione del problema
    for (int i = 0; i < nodi; i++) {
        for (int j = 0; j < nodi; j++) {
            if (i == j) {
                matrice[i][j] = 0;
            }
        }
    }
    // Il valore di ritorno del metodo è la matrice inserita
    return matrice;
}
}

```

Successivamente viene presentato il codice della classe TSP.java con i relativi metodi e attributi.

```
import java.util.ArrayList;
/**
 * La classe TSP è la classe in cui al suo interno sono definiti attributi e metodi che permettono
 * di implementare gli algoritmi di NN e RNN
 */
public class TSP {
    /**
     * Attributo che memorizza il numero di nodi inseriti dall'utente precedentemente
     */
    private int nodi;
    /**
     * struttura dati che memorizza la matrice dei costi
     */
    private double matrice[][];
    /**
     * L'array visitati è una struttura dati che permette di tener conto quali nodi sono
     * stati visitati e quelli ancora da visitare
     */
    private Boolean[] visitati;

    /**
     * Metodo costruttore. Metodo richiamato nel momento in cui nel main viene creato l'oggetto
     * di tipo TSP. Questo metodo permette di inizializzare gli attributi e strutture dati
     * necessarie per la risoluzione del problema
     */
    public TSP(int nodes, double matrice[][]) {
        nodi = nodes;
        this.matrice = new double[nodi][nodi];
        this.matrice = matrice;
        this.visitati = new Boolean[nodi];
    }

    /**
     * Metodo che implementa l'algoritmo di Nearest Neighbor. Prende come parametro il nodo
     * di partenza inserito precedentemente dall'utente.
     * @param nodoPartenza
     */
    public void nearestNeighbor(int nodoPartenza) {
        // L'oggetto sequenza è un array che permette di memorizzare il cammino hamiltoniano
        // determinato
        ArrayList<Integer> sequenza = new ArrayList<Integer>();
        // La variabile costo tiene conto del valore della funzione obiettivo.
        // All'inizio viene settato il suo valore a 0
        double costo = 0;

        // Richiamo il metodo che permette di settare l'array visitati in maniera opportuna
        this.resetVisitati(nodoPartenza);
        // Aggiungo al ciclo il nodo di partenza
        sequenza.add(nodoPartenza);
        // Richiamo il metodo sequenceBuilder che determina la soluzione del problema e restituisce
        // il valore della funzione obiettivo
        costo = this.sequenceBuilder(sequenza, nodoPartenza);

        // Stampa a video della soluzione determinata e il relativo valore della funzione obiettivo
        System.out.print("Ciclo Hamiltoniano determinato: [");
        for(int n : sequenza) {
            System.out.print((n+1) + " ");
        }
        System.out.println("]");
        // Stampa a video il valore della funzione obiettivo
        System.out.println("Il costo associato: " + costo);
    }
}
```

```

/**
 * Metodo che implementa l'algoritmo di Repetitive Nearest Neighbor. Prende come parametro la
 * matrice dei costi inserita precedentemente dall'utente.
 */
public void repetitiveNearestNeighbor() {
    // Allocazione delle strutture dati per la memorizzazione dei cicli hamiltoniani (determinati
    // considerando ogni volta un nodo di partenza diverso) e della soluzione finale.
    ArrayList<Integer> sequenza = new ArrayList<Integer>();
    int[][] soluzione = new int[nodi + 1][nodi + 1];
    // La variabile costoMinimo è utilizzata per la memorizzazione del valore ottimo della
    // funzione obiettivo
    double costoMinimo = Double.MAX_VALUE;
    double[] costi = new double[nodi];
    // Loop che consente di applicare il NN considerando ogni volta un nodo di partenza diverso.
    // Il loop termina quando tutti i nodi sono stati considerati come nodi di partenza.
    // La variabile nodoPartenza corrisponde al nodo di partenza. All'inizio viene impostato
    // come nodo di partenza il primo e ad ogni iterazione viene incrementato
    for (int nodoPartenza = 0; nodoPartenza < nodi; nodoPartenza++) {
        //Imposto il valore della funzione obiettivo a 0. In questo caso la variabile 'costo'
        //corrisponde al valore della funzione obiettivo considerando 'nodoPartenza' come
        //nodo iniziale
        double costo;

        //Ad ogni iterazione la variabile sequenza tiene conto dei vari cammini hamiltoniani
        //determinati considerando un nodo di partenza differente.
        //Ogni volta la variabile viene svuotata. Questo permette di creare meno strutture
        //dati e avere una complessità spaziale migliore.
        sequenza.clear();
        //Aggiungo come nodo di partenza del ciclo la variabile 'nodoPartenza' aggiungendolo
        //alla variabile 'sequenza'
        sequenza.add(nodoPartenza);

        // Metodo che permette di settare i valori dell'array visitati in maniera opportuna.
        // Ad ogni iterazione il vettore viene aggiornato mettendo a true solo il valore del
        // nodo di partenza passato come parametro
        this.resetVisitati(nodoPartenza);

        // Richiamo il metodo che implementa il NN. Questo metodo viene richiamato tante volte
        // quanti sono i nodi del grafo. Infatti, ad ogni iterazione, si determina il cammino
        // hamiltoniano considerando un diverso nodo di partenza. I vari nodi di partenza
        // vengono passati come parametro
        costo = this.sequenceBuilder(sequenza, nodoPartenza);
        //System.out.println("Fouri: "+sequenza);

        // Stampa a video dei vari cicli hamiltoniani determinati e i relativi valori della
        // funzione obiettivo
        System.out.print("Ciclo hamiltoniano considerando " + (nodoPartenza + 1) + " come
nodo" +
                    " " +
                    "di partenza: " + "[");
        for (int n : sequenza) {
            System.out.print((n + 1) + " ");
        }
        System.out.println("]");
        System.out.println("Il costo associato: " + (costo) + "\n");

        // In questa sezione vengono memorizzati i costi dei vari cammini determinati
        // all'interno del vettore 'costi' . Successivamente viene salvato il ciclo
        // Hamiltoniano nella struttura dati soluzione.
        costi[nodoPartenza] = costo;
        for (int i = 0; i < sequenza.size(); i++) {
            soluzione[nodoPartenza][i] = sequenza.get(i);
        }

        // L'if in esame serve per tener conto del costo migliore determinato
    }
}

```

```

        if (costo < costoMinimo) {
            costoMinimo = costo;
        }
    }
    // Una volta usciti dal while abbiamo nel vettore costi tutti i valori della funzione
    // obiettivo dei vari cicli Hamiltoiani determinati.
    for (int i = 0; i < nodi; i++) {
        // Si confronta ogni valore del vettore costi con il valore del costoMinimo determinato
        // Nel caso in cui abbiamo una corrispondenza si stampa a video la sequenza dei nodi
        // a cui corrisponde il miglior valore della rispettiva funzione obiettivo.
        // Questo ci permette di stampare più sequenze di nodi nel caso in cui abbiamo cicli
        // Hamiltoiani con nodo di partenza differente ma stesso valore della funzione obiettivo
        if (costi[i] == costoMinimo) {
            System.out.print("Ciclo Hamiltoniano ottimo con nodo di partenza " + (i + 1) +
": " +
                "[ ");
            for (int k = 0; k < nodi + 1; k++) {
                System.out.print(((soluzione[i][k]) + 1) + " ");
            }
            System.out.println("]");
        }
    }
    System.out.println("Il costo ottimo determinato: " + costoMinimo);
}

/**
 * Metodo che permette di ricavare il ciclo Hamiltoniano seguendo i passi dell'algoritmo NN.
 * Prende come parametro la struttura dati 'sequenza' e il nodo da cui partire per applicare
 * l'algoritmo. Questo metodo viene richiamato sia nel caso in cui bisogna applicare il NN
 * sia nel caso in cui bisogna utilizzare il RNN. Nel primo caso questo metodo viene
 * richiamato una sola volta considerando un solo nodo di partenza.
 * Nel secondo caso questo metodo viene richiamato tante volte quante sono i nodi nel grafo.
 * Ad ogni iterazione la variabile nodoCorrente assume un valore differente.
 * @param sequenza
 * @param nodoCorrente
 * @return
 */
private double sequenceBuilder(ArrayList<Integer> sequenza, int nodoCorrente) {
    // La variabile next consente di memorizzare il nodo più vicino determinato
    int next = 0;
    // La variabile min consente di memorizzare il costo minimo del nodo più vicino determinato.
    // La variabile costo tiene conto del valore della funzione obiettivo.
    double min, costo = 0;

    // Loop che implementa l'algoritmo del NN. Il ciclo termina quando la sequenza determinata
    // ha un numero di nodi pari al numero di nodi presenti nel grafo
    while (sequenza.size() != nodi) {
        // imposto la variabile min con il massimo valore che un double può assumere
        min = Double.MAX_VALUE;
        // Ciclo che permette di determinare il nodo più vicino non ancora visitato rispetto al
        // nodo considerato (nodoCorrente).
        // Per nodo più vicino si intende il nodo il cui costo per raggiungerlo è minore di
        // tutti gli altri. Corrisponde al nodo, non ancora visitato, il cui valore della
        // matrice dei costi è il minore di tutti.
        for (int j = 0; j < nodi; j++) {
            if (nodoCorrente != j &&
                (matrice[nodoCorrente][j] < min) &&
                !visitati[j]) {
                next = j;
                min = matrice[nodoCorrente][j];
            }
        }
        // Una volta determinato il nodo più vicino si aggiunge alla sequenza
        sequenza.add(next);
        // Si aggiorna il nodo da considerare per la prossima iterazione
    }
}

```

```

        nodoCorrente = next;
        // Si imposta il nodo determinato come visitato
        visitati[next] = true;
        // Si aggiorna il valore della funzione obiettivo
        costo += min;
    }
    // Una volta determinata la sequenza si aggiorna il valore della funzione obiettivo
    // aggiungendo il costo per ritornare al nodo di partenza
    costo += matrice[sequenza.size() - 1][sequenza.get(0)];
    // Si aggiunge il nodo di partenza come ultimo nodo visitato
    sequenza.add(sequenza.get(0));
    // Ritorna il valore della funzione obiettivo
    return costo;
}
}

```

Di particolare importanza è il metodo sequenceBuilder presentato in seguito

```

private double sequenceBuilder(ArrayList<Integer> sequenza, int nodoCorrente) {
    int next = 0;
    double min, costo = 0;
    while (sequenza.size() != nodi) {
        min = Double.MAX_VALUE;
        for (int j = 0; j < nodi; j++) {
            if (nodoCorrente != j && (matrice[nodoCorrente][j] < min) && !visitati[j]) {
                next = j;
                min = matrice[nodoCorrente][j];
            }
        }
        sequenza.add(next);
        nodoCorrente = next;
        visitati[next] = true;
        costo += min;
    }
    costo += matrice[sequenza.get(sequenza.size() - 1)][sequenza.get(0)];
    sequenza.add(sequenza.get(0));
    return costo;
}

```

Questa funzione è il cuore dell'implementazione dei due metodi euristici. La procedura in esame prende come parametro una struttura dati di interi (`ArrayList<Integer> sequenza`) e una variabile intera (`int nodoCorrente`). Le variabili che vengono dichiarate e inizializzate sono next, min e costo. La prima serve per tener conto del nodo più vicino, non ancora visitato, rispetto al nodo che si sta analizzando memorizzato in `nodoCorrente`. In generale, viene salvato il nodo il cui costo per raggiungerlo risulta essere il minore di tutti gli altri. Per la determinazione del ciclo Hamiltoniano si utilizza un ciclo while che si arresta solo quando il numero di elementi della variabile ‘sequenza’ è uguale al numero dei nodi presenti nel grafo. All’interno del ciclo while, attraverso un for, si scorre la matrice dei costi per colonna andando a cercare il valore minore tra i nodi non ancora visitati (nodi il cui elemento all’interno dell’array `visitati` è diverso da false). Una volta usciti dal ciclo, nella variabile ‘next’ sarà presente il nodo che dovrà essere aggiunto alla sequenza e che, all’iterazione successiva, corrisponderà al `nodoCorrente`. Infine, si aggiorna il valore della funzione

obiettivo aggiungendo alla variabile ‘costo’ la variabile ‘min’, corrispondente al costo per raggiungere il nodo ‘next’. Una volta usciti dal while, si aggiunge in coda alla struttura sequenza il nodo di partenza e la funzione termina facendo ritornare il valore della funzione obiettivo (costo).

La procedura viene richiamata una volta solo nel momento in cui l’utente sceglie come metodo risolutivo il NN. In questo caso il parametro `nodoCorrente` risulterà essere uguale al nodo di partenza richiesto dal programma come valore di input. Nel caso in cui venga selezionato il RNN, questo metodo viene richiamato tante volte quante sono i nodi del grafo. In sostanza, verranno determinati diversi cicli hamiltoniani considerando ogni volta come nodo di partenza un nodo diverso. In questo caso, il parametro `nodoCorrente` assumerà, ad ogni iterazione, un valore differente. Una caratteristica particolare del software, riguardo l’implementazione del metodo euristico RNN, è che è in grado di tener conto delle soluzioni multiple. Infatti, nel caso in cui abbiamo più cicli Hamiltoniani con nodi partenza differenti ma con stesso valore della funzione obiettivo, il programma ne tiene conto stampando tutte le soluzioni ottime determinate.

Un semplice esempio, considerando il NN come metodo risolutivo selezionato, rende il concetto del funzionamento della funzione `sequenceBuilder()` più chiaro. Consideriamo la seguente matrice dei costi e come nodo di partenza il nodo 2.

	1	2	3	4
1	0	55	23	7
2	4	0	44	16
3	33	24	0	9
4	25	63	48	0

La funzione `sequenceBuilder()` viene richiamata e la variabile ‘nodoCorrente’ è uguale a 2. Le variabili ‘next’ e ‘costo’ vengono inizializzate a 0 mentre ‘sequenza’ è un array in cui nella prima posizione è presente 2 (questa operazione viene effettuata prima di richiamare la funzione). Una volta entrato nel while si imposta la variabile ‘min’ con il massimo valore che può assumere un double. Questo perché alle varie iterazioni permetterà di settare ‘min’ con il primo elemento della matrice considerata.

In forma schematica abbiamo la seguente situazione:

next	0
nodoCorrente	2
sequenza	[2]
min	1.7976931348623157R+308
visitati	[false,true,false,false]
costo	0

Il for interno non fa altro che scansionare la riga 2 della matrice alla ricerca del minimo, escludendo la posizione (2,2) e tutti i nodi già visitati (coloro che hanno il valore del vettore visitati[i] a true). In questo caso verranno scansionati tutti gli elementi della riga poiché solo il nodo 2 è stato visitato. In rosso vengono indicati le posizioni escluse, in verde il valore minimo.

4	0	44	16
---	---	----	----

Il minimo si trova in posizione (1, 1) quindi la variabile ‘next’ è uguale a 1 e ‘min’ uguale a 4. Viene aggiunto ‘next’ a ‘sequenza’, viene impostato nodoCorrente uguale a ‘next’, sarà impostato visitati[1] a true e infine viene aggiornato ‘costo’ aggiungendo ‘min’. In forma schematicaabbiamo la seguente situazione:

next	1
nodoCorrente	1
sequenza	[2,1]
min	4
visitati	[true,true,false,false]
costo	4

All’iterazione successiva si scansiona la riga 1 escludendo la posizione (1,1) e (1,2) questo perché visitati[2] è a true. Ad ogni iterazione la variabile ‘min’ assume il massimo valore che può essere una variabile double.

0	55	23	7
---	----	----	---

Il minimo è 7 in posizione (1, 4). Alla seconda iterazione abbiamo la seguente situazione:

next	4
nodoCorrente	4
sequenza	[2,1,4]
min	7
visitati	[true,true,false,true]
costo	11

Alla terza iterazione si cercherà il minimo scansionando la riga 4, escludendo le posizioni (4,1),(4,2),(4,4):

25	63	48	0
----	----	----	---

Avremo la seguente situazione:

next	3
nodoCorrente	3
sequenza	[2,1,4,3]
min	25
visitati	[true,true,true,true]
costo	59

A questo punto si esce dal while e si aggiorna ‘sequenza’ inserendo il nodo 2 per completare il ciclo. Infine si aggiorna il valore della funzione obiettivo inserendo l’ultimo costo che completa il ciclo Hamiltoniano. Il ciclo determinato e il relativo valore della funzione obiettivo sono:

$$\text{sequenza} = (2, 1, 4, 3, 2)$$

$$\text{costo} = 83$$

## 5. Determinazione ottima dei giri di consegna

Nell'ambito dei trasporti i giri di consegna rappresentano la fase finale della distribuzione dei prodotti. Si ricorre a metodi risolutivi per la determinazione dei percorsi ottimi soprattutto per la consegna di prodotti di largo consumo come la stampa, prodotti farmaceutici, i volantini delle pubblicità ecc ecc.

Questa tipologia di problemi si basa principalmente sulla minimizzazione dei chilometri percorsi o del tempo impiegato. In generale, possiamo rappresentare il problema attraverso un grafo in cui ogni nodo rappresenta una località e ai vari archi che collegano i vertici sono associati dei coefficienti di costo che possono rappresentare la distanza fra le varie città o il tempo necessario per raggiungerle.

### 5.1 Il problema

Un'agenzia di distribuzione di giornali inglese distribuisce in diverse edicole quotidiani e periodici. L'attività principale svolta da questa azienda è la distribuzione, confezionamento e spedizione di prodotti di stampa. Poiché l'agenzia gestisce diverse zone per semplicità verrà considerato un singolo quartiere. Date una serie di località determinare il percorso ottimo che minimizzi la distanza percorsa. In seguito viene presentata la tabella dei costi in cui i coefficienti rappresentano la distanza in metri tra le varie località.

	Mansfield Woodhouse	Nottingham	Rawmarsh	Whitefield	Aldridge	Heywood
Mansfield Woodhouse	0	26190	39860	96640	87270	98710
Nottingham	26460	0	63240	115590	74120	117660
Rawmarsh	39760	63210	0	77220	115520	74350
Whitefield	96710	115630	76780	0	121751	8790
Aldridge	87420	74750	115770	122080	0	127400
Heywood	98670	117590	73830	9000	127041	0

In seguito verranno presentate le varie iterazioni e le soluzioni offerte applicando prima i metodi esatti (generazione dei vincoli, Branch & Bound) tramite l'esecuzione degli script in AMPL implementati, successivamente verrà risolto il problema mandando in esecuzione il programma in Java applicando i metodi euristici NN e RNN.

## 5.2 Generazione dei vincoli

Costruendo in maniera opportuna il file dataTSP.dat inserendo il numero di nodi ‘n’ e la matrice dei costi, possiamo mandare in esecuzione lo script algoritmo.run scrivendo nella console il seguente comando:

```
include algoritmo.run;
```

In questo modo viene eseguito il programma costruito e verrà stampato a video, ad ogni iterazione, la funzione obiettivo, i vincoli di grado e i vincoli di eliminazione dei sotto-cicli, la soluzione determinata, i vari sotto-cicli individuati e il valore della funzione obiettivo.

```
ampl: include algoritmo.run;
```

Iterazione numero 1

La funzione obiettivo:

minimize costociclo:

$$\begin{aligned} & 26190*x[1,2] + 39860*x[1,3] + 96640*x[1,4] + 87270*x[1,5] + \\ & 98710*x[1,6] + 26460*x[2,1] + 63240*x[2,3] + 115590*x[2,4] + \\ & 74120*x[2,5] + 117660*x[2,6] + 39760*x[3,1] + 63210*x[3,2] + \\ & 77220*x[3,4] + 115520*x[3,5] + 74350*x[3,6] + 96710*x[4,1] + \\ & 115630*x[4,2] + 76780*x[4,3] + 121751*x[4,5] + 8790*x[4,6] + \\ & 87420*x[5,1] + 74750*x[5,2] + 115770*x[5,3] + 122080*x[5,4] + \\ & 127400*x[5,6] + 98670*x[6,1] + 117590*x[6,2] + 73830*x[6,3] + \\ & 9000*x[6,4] + 127041*x[6,5]; \end{aligned}$$

Vincoli di grado:

subject to successore[1]:

$$x[1,2] + x[1,3] + x[1,4] + x[1,5] + x[1,6] = 1;$$

subject to successore[2]:

$$x[2,1] + x[2,3] + x[2,4] + x[2,5] + x[2,6] = 1;$$

subject to successore[3]:

$$x[3,1] + x[3,2] + x[3,4] + x[3,5] + x[3,6] = 1;$$

subject to successore[4]:

$$x[4,1] + x[4,2] + x[4,3] + x[4,5] + x[4,6] = 1;$$

subject to successore[5]:

$$x[5,1] + x[5,2] + x[5,3] + x[5,4] + x[5,6] = 1;$$

subject to successore[6]:

$$x[6,1] + x[6,2] + x[6,3] + x[6,4] + x[6,5] = 1;$$

subject to predecessore[1]:  
 $x[2,1] + x[3,1] + x[4,1] + x[5,1] + x[6,1] = 1;$

subject to predecessore[2]:  
 $x[1,2] + x[3,2] + x[4,2] + x[5,2] + x[6,2] = 1;$

subject to predecessore[3]:  
 $x[1,3] + x[2,3] + x[4,3] + x[5,3] + x[6,3] = 1;$

subject to predecessore[4]:  
 $x[1,4] + x[2,4] + x[3,4] + x[5,4] + x[6,4] = 1;$

subject to predecessore[5]:  
 $x[1,5] + x[2,5] + x[3,5] + x[4,5] + x[6,5] = 1;$

subject to predecessore[6]:  
 $x[1,6] + x[2,6] + x[3,6] + x[4,6] + x[5,6] = 1;$

Vincoli di eliminazione dei sottocicli:

CPLEX 12.6.3.0:

La soluzione determinata:

```
x [*,*]
:   1   2   3   4   5   6    :=
1   0   0   1   0   0   0
2   0   0   0   0   1   0
3   1   0   0   0   0   0
4   0   0   0   0   0   1
5   0   1   0   0   0   0
6   0   0   0   1   0   0
; 
```

I sottocicli individuati:

```
set ciclo[i] := 1 3;
```

```
set ciclo[i] := 2 5;
```

```
set ciclo[i] := 4 6;
```

Il valore della funzione obiettivo: 246280.00000

Notiamo che alla prima iterazione verrà risolto il problema senza nessun vincolo di eliminazione dei sotto-cicli. La soluzione presentata conterrà 3 sotto-cicli:

$$(1, 3, 1), (2, 5, 2), (4, 6, 4)$$

All'iterazione successiva (iterazione 2) verranno creati i vincoli di eliminazione per i sotto-cicli identificati.

Vincoli di eliminazione dei sottocicli:

subject to nocicli[1]:

$$x[1,2] + x[1,4] + x[1,5] + x[1,6] + x[3,2] + x[3,4] + x[3,5] + \\ x[3,6] \geq 1;$$

subject to nocicli[2]:

$$x[2,1] + x[2,3] + x[2,4] + x[2,6] + x[5,1] + x[5,3] + x[5,4] + \\ x[5,6] \geq 1;$$

subject to nocicli[3]:

$$x[4,1] + x[4,2] + x[4,3] + x[4,5] + x[6,1] + x[6,2] + x[6,3] + \\ x[6,5] \geq 1;$$

Finchè non viene individuata la soluzione ottima ammissibile verranno stampati a video, ad ogni iterazione, la funzione obiettivo, i vincoli di grado, i vincoli di eliminazione dei sotto-cicli, il valore della funzione obiettivo. Per rendere più leggibili i vari passi dell'esecuzione del programma di qui in poi verranno presentati la soluzione  $x$ , i sotto-cicli identificati, i rispettivi vincoli di eliminazione al passo successivo e il valore della funzione obiettivo.

Considerando l'iterazione 2

La soluzione determinata:

```
x [*,*]
:   1   2   3   4   5   6   :=
1  0   1   0   0   0   0
2  0   0   0   0   1   0
3  1   0   0   0   0   0
4  0   0   0   0   0   1
5  0   0   0   1   0   0
6  0   0   1   0   0   0
;
```

I sottocicli individuati:

```
set ciclo[i] := 1 2 5 4 6 3;
```

Il valore della funzione obiettivo: 344770.000000

Il sotto-ciclo identificato nella soluzione ricavata all'iterazione 2 è:

$$(1, 2, 5, 4, 6, 3, 1)$$

Notiamo che questo è un ciclo Hamiltoniano e quindi siamo arrivati alla soluzione ottima ammissibile del problema. Successivamente il programma si arresta e stampa a video la soluzione ottima determinata.

La soluzione finale determinata:

```
x [*,*]
:   1   2   3   4   5   6      :=
1   0   1   0   0   0   0
2   0   0   0   0   1   0
3   1   0   0   0   0   0
4   0   0   0   0   0   1
5   0   0   0   1   0   0
6   0   0   1   0   0   0
;
```

Il ciclo hamiltoniano minimo determinato:

```
set cicloHamiltoniano := 1 2 5 4 6 3;
```

presolve, constraint nocicli[4]:

```
no variables, but lower bound = 1, upper = Infinity
```

Il costo del ciclo hamiltoniano minimo: 344770.000000

Il percorso ottimo determinato è il seguente:

(1, 2, 5, 4, 6, 3, 1)

L'ordine con cui visitare le località è:

( Mansfield Woodhouse, Nottingham, Aldridge, Whitefield, Heywood, Rawmarsh, Mansfield Woodhouse )

I chilometri percorsi sono: 344,770 Km

### 5.3 Branch & Bound

Risolvere il problema del commesso viaggiatore con il metodo Branch & Bound significa risolverlo come un problema di assegnamento eliminando dal modello matematico i vincoli di eliminazione dei sotto-cicli. Ogni volta che viene individuata la soluzione si valuta se sono presenti dei sotto-cicli. Se non ci sono sotto-cicli allora la soluzione è ottima altrimenti si effettua il branching generando sotto-problemi e inserendo i vincoli di eliminazione dei sotto-cicli individuati. Nella risoluzione del problema P0 e P1 verrà mostrata l'intera formulazione del problema, visualizzando la funzione obiettivo e i relativi vincoli di grado. Per agevolare la lettura dei risultati, alle iterazioni

successive, verrà mostrata la matrice che viene passata al risolutore, la soluzione ricavata e il valore della funzione obiettivo.

### 5.3.1 P0

Il problema P0 viene ottenuto eliminando tutti i vincoli di eliminazione dei sotto-cicli in modo tale da ottenere un problema di assegnamento risolvibile in modo efficiente con i attraverso i tradizionali metodi risolutivi (Simplex, Ungherese). La matrice che viene passata al risolutore è la seguente:

	1	2	3	4	5	6
1	0	26190	39860	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	39760	63210	0	77220	115520	74350
4	96710	115630	76780	0	121751	8790
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

Eseguendo lo script creato attraverso il comando:

```
include branch_and_bound.run;
```

il programma ci restituisce i seguenti valori:

```
ampl: include branch_and_bound.run;
minimize costociclo:
  26190*x[1,2] + 39860*x[1,3] + 96640*x[1,4] + 87270*x[1,5] +
  98710*x[1,6] + 26460*x[2,1] + 63240*x[2,3] + 115590*x[2,4] +
  74120*x[2,5] + 117660*x[2,6] + 39760*x[3,1] + 63210*x[3,2] +
  77220*x[3,4] + 115520*x[3,5] + 74350*x[3,6] + 96710*x[4,1] +
  115630*x[4,2] + 76780*x[4,3] + 121751*x[4,5] + 8790*x[4,6] +
  87420*x[5,1] + 74750*x[5,2] + 115770*x[5,3] + 122080*x[5,4] +
  127400*x[5,6] + 98670*x[6,1] + 117590*x[6,2] + 73830*x[6,3] +
  9000*x[6,4] + 127041*x[6,5];

subject to successore[1]:
  x[1,2] + x[1,3] + x[1,4] + x[1,5] + x[1,6] = 1;

subject to successore[2]:
  x[2,1] + x[2,3] + x[2,4] + x[2,5] + x[2,6] = 1;

subject to successore[3]:
  x[3,1] + x[3,2] + x[3,4] + x[3,5] + x[3,6] = 1;
```

```

subject to successore[4]:
  x[4,1] + x[4,2] + x[4,3] + x[4,5] + x[4,6] = 1;

subject to successore[5]:
  x[5,1] + x[5,2] + x[5,3] + x[5,4] + x[5,6] = 1;
subject to successore[6]:
  x[6,1] + x[6,2] + x[6,3] + x[6,4] + x[6,5] = 1;

subject to predecessore[1]:
  x[2,1] + x[3,1] + x[4,1] + x[5,1] + x[6,1] = 1;

subject to predecessore[2]:
  x[1,2] + x[3,2] + x[4,2] + x[5,2] + x[6,2] = 1;

subject to predecessore[3]:
  x[1,3] + x[2,3] + x[4,3] + x[5,3] + x[6,3] = 1;

subject to predecessore[4]:
  x[1,4] + x[2,4] + x[3,4] + x[5,4] + x[6,4] = 1;

subject to predecessore[5]:
  x[1,5] + x[2,5] + x[3,5] + x[4,5] + x[6,5] = 1;

subject to predecessore[6]:
  x[1,6] + x[2,6] + x[3,6] + x[4,6] + x[5,6] = 1;

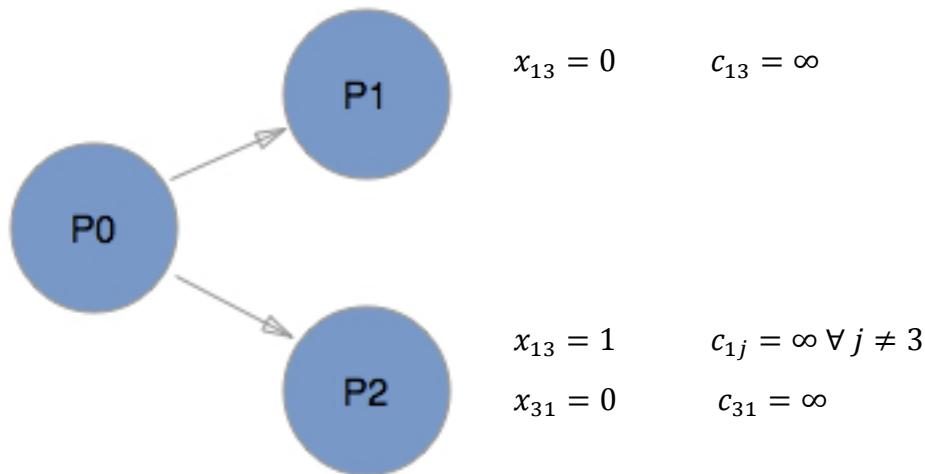
CPLEX 12.6.3.0: x [*,*]
:   1   2   3   4   5   6      :=
1   0   0   1   0   0   0
2   0   0   0   0   1   0
3   1   0   0   0   0   0
4   0   0   0   0   0   1
5   0   1   0   0   0   0
6   0   0   0   1   0   0
;
costociclo = 246280

```

La soluzione presenta dei sotto-cicli e quindi il valore della funzione obiettivo è un limite inferiore della soluzione ottima ammissibile. I sotto-cicli individuati sono i seguenti:

$$(1, 3, 1), (2, 5, 2), (4, 6, 4)$$

Prendendo in considerazione il primo sotto-ciclo (1,3,1) si genereranno i sotto-problemi P1 e P2.



### 5.3.2 P1 (ottenuto da P0)

Il sotto-problema P1 è ottenuto in corrispondenza del problema P0 forzando la variabile  $x_{13} = 0$ . Per preservare la unimodularità della matrice dei vincoli, viene modificata la matrice dei costi. Imponendo  $c_{13} = \infty$  si forza a non percorrere determinati cammini. La matrice che viene passata come parametro di input al risolutore è la seguente:

	1	2	3	4	5	6
1	0	26190	1000000	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	39760	63210	0	77220	115520	74350
4	96710	115630	76780	0	121751	8790
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

Viene impostato il valore di  $c_{13}$  a 1000000 e si richiama di nuovo lo script branch\_and\_bound.run

La soluzione che viene presentata è la seguente:

```

ampl: include branch_and_bound.run;
minimize costociclo:
  26190*x[1,2] + 1e+07*x[1,3] + 96640*x[1,4] + 87270*x[1,5] +
  98710*x[1,6] + 26460*x[2,1] + 63240*x[2,3] + 115590*x[2,4] +

```

$74120*x[2,5] + 117660*x[2,6] + 39760*x[3,1] + 63210*x[3,2] +$   
 $77220*x[3,4] + 115520*x[3,5] + 74350*x[3,6] + 96710*x[4,1] +$   
 $115630*x[4,2] + 76780*x[4,3] + 121751*x[4,5] + 8790*x[4,6] +$   
 $87420*x[5,1] + 74750*x[5,2] + 115770*x[5,3] + 122080*x[5,4] +$   
 $127400*x[5,6] + 98670*x[6,1] + 117590*x[6,2] + 73830*x[6,3] +$   
 $9000*x[6,4] + 127041*x[6,5];$

subject to successore[1]:

$$x[1,2] + x[1,3] + x[1,4] + x[1,5] + x[1,6] = 1;$$

subject to successore[2]:

$$x[2,1] + x[2,3] + x[2,4] + x[2,5] + x[2,6] = 1;$$

subject to successore[3]:

$$x[3,1] + x[3,2] + x[3,4] + x[3,5] + x[3,6] = 1;$$

subject to successore[4]:

$$x[4,1] + x[4,2] + x[4,3] + x[4,5] + x[4,6] = 1;$$

subject to successore[5]:

$$x[5,1] + x[5,2] + x[5,3] + x[5,4] + x[5,6] = 1;$$

subject to successore[6]:

$$x[6,1] + x[6,2] + x[6,3] + x[6,4] + x[6,5] = 1;$$

subject to predecessore[1]:

$$x[2,1] + x[3,1] + x[4,1] + x[5,1] + x[6,1] = 1;$$

subject to predecessore[2]:

$$x[1,2] + x[3,2] + x[4,2] + x[5,2] + x[6,2] = 1;$$

subject to predecessore[3]:

$$x[1,3] + x[2,3] + x[4,3] + x[5,3] + x[6,3] = 1;$$

subject to predecessore[4]:

$$x[1,4] + x[2,4] + x[3,4] + x[5,4] + x[6,4] = 1;$$

subject to predecessore[5]:

$$x[1,5] + x[2,5] + x[3,5] + x[4,5] + x[6,5] = 1;$$

subject to predecessore[6]:

$$x[1,6] + x[2,6] + x[3,6] + x[4,6] + x[5,6] = 1;$$

CPLEX 12.6.3.0: x [\*,\*]

:	1	2	3	4	5	6	:=
1	0	1	0	0	0	0	

```

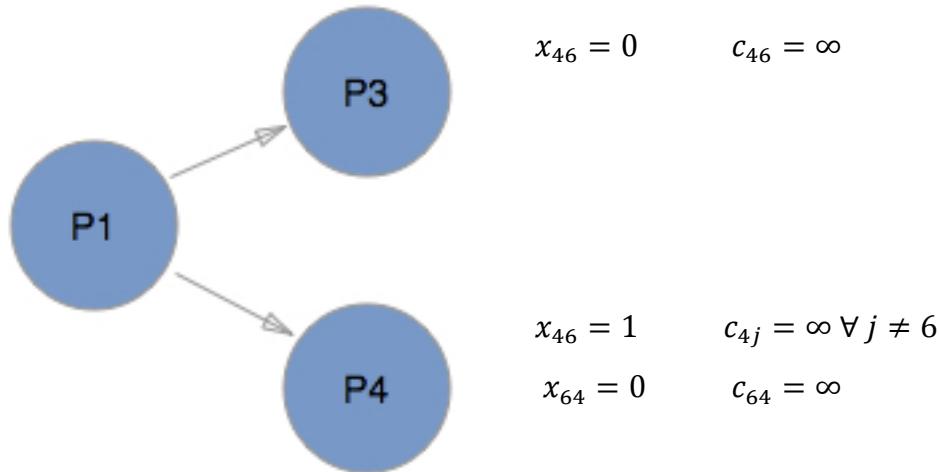
2   0   0   0   0   1   0
3   1   0   0   0   0   0
4   0   0   0   0   0   1
5   0   0   1   0   0   0
6   0   0   0   1   0   0;
costociclo = 273630

```

Anche in questo caso la soluzione non è ottima poiché sono presenti dei sotto-cicli:

$$(1,2,5,3,1), (4,6,4)$$

Scegliendo il sotto-ciclo (4,6,4) verranno generati i sotto-problemi P3 e P4.



### 5.3.3 P2 (ottenuto da P0)

Il problema P2 è stato ottenuto in corrispondenza del problema P1 imponendo le variabili  $x_{13} = 1$  e  $x_{31} = 0$ . Per mantenere la totale unimodularità è stata modificata la matrice dei costi imponendo  $c_{31} = 1000000$  e  $c_{1j} = 1000000 \forall j \neq 3$ . La matrice che viene passata al risolutore è la seguente:

	1	2	3	4	5	6
1	0	1000000	39860	1000000	1000000	1000000
2	26460	0	63240	115590	74120	117660
3	1000000	63210	0	77220	115520	74350
4	96710	115630	76780	0	121751	8790
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

La soluzione restituita:

```
CPLEX 12.6.3.0: x [*,*]
```

```

: 1 2 3 4 5 6 :=
1 0 0 1 0 0 0
2 1 0 0 0 0 0
3 0 0 0 0 1 0
4 0 0 0 0 0 1
5 0 1 0 0 0 0
6 0 0 0 1 0 0
;

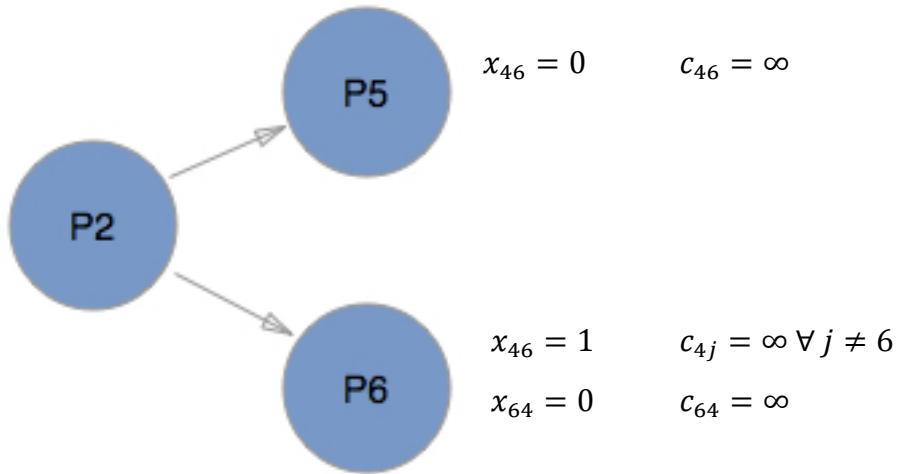
```

`costociclo = 274380`

Sono presenti due sotto-cicli:

$$(1, 3, 5, 2, 1), (4, 6, 4)$$

Scegliendo il secondo sotto-ciclo genereremo i problemi figli P5 e P6.



Poiché il valore della funzione obiettivo del problema P1 è migliore rispetto a quella di P2, si proseguirà nella risoluzione dei sotto-problemi ottenuti da P1.

### 5.3.4 P3 (ottenuto da P1)

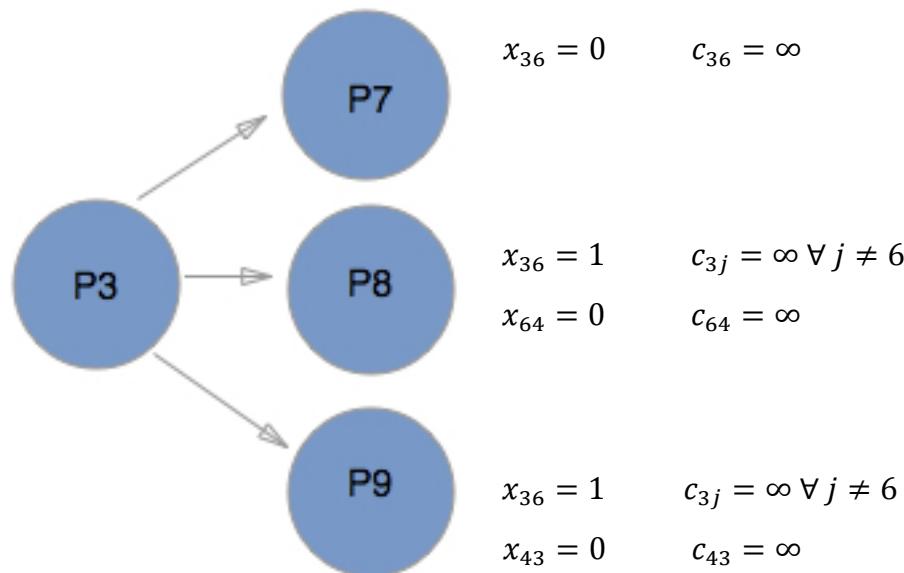
P3 è stato ottenuto in corrispondenza del problema P1 impostando  $x_{46} = 0$  e quindi settando  $c_{46} = 1000000$ .

	1	2	3	4	5	6
1	0	26190	1000000	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	39760	63210	0	77220	115520	74350
4	96710	115630	76780	0	121751	1000000
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

La soluzione offerta è la seguente:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6   :=
1 0 1 0 0 0 0
2 0 0 0 0 1 0
3 0 0 0 0 0 1
4 0 0 1 0 0 0
5 1 0 0 0 0 0
6 0 0 0 1 0 0
;
costociclo = 347860
```

Notiamo la presenza dei sotto-cicli (1, 2, 5, 1) e (3, 6, 4, 3). Scegliendo il secondo sotto-ciclo otteniamo i problemi figli P7, P8 e P9.



### 5.3.5 P4 (ottenuto da P1)

P4 è ottenuto in corrispondenza del problema P1 settando  $x_{46} = 1$  e  $x_{64} = 0$  e quindi impostando  $c_{4j} = 1000000 \forall j \neq 6$ , e  $c_{64} = 1000000$ .

	1	2	3	4	5	6
1	0	26190	1000000	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	39760	63210	0	77220	115520	74350
4	1000000	1000000	1000000	0	1000000	8790
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	1000000	127041	0

La soluzione determinata è la seguente:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6      :=
1 0 1 0 0 0 0
2 0 0 0 0 1 0
3 1 0 0 0 0 0
4 0 0 0 0 0 1
5 0 0 0 1 0 0
6 0 0 1 0 0 0
;
costociclo = 344770
```

La soluzione del problema P4 è un ciclo Hamiltoniano

$$(1, 2, 5, 4, 6, 3, 1)$$

il cui valore della funzione obiettivo è 344,770 Km. Abbiamo determinato la prima soluzione ammissibile per il problema. Per verificare se è anche la soluzione ottima, dobbiamo andare a risolvere tutti gli altri problemi figli.

Poiché i valori della soluzione del problema P3 risulta essere maggiore della soluzione ottenuta risolvendo P2, si procederà nella risoluzione dei problemi figli di quest'ultimo. Questo perché il valore della soluzione che otteniamo è un limite inferiore (lower bound) e quindi è conveniente risolvere i sotto-problemi provenienti da problemi padri con valori della funzione obiettivo minore (politica di esplorazione del best bound first).

### 5.3.6 P5 (ottenuto da P2)

P5 è stato ottenuto in corrispondenza del problema P2 impostando  $x_{46} = 0$  e perciò settando  $c_{46} = 1000000$ .

	1	2	3	4	5	6
1	0	1000000	39860	1000000	1000000	1000000
2	26460	0	63240	115590	74120	117660
3	1000000	63210	0	77220	115520	74350
4	96710	115630	76780	0	121751	1000000
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

La soluzione ricavata è la seguente:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6   :=
1 0 0 1 0 0 0
2 1 0 0 0 0 0
3 0 0 0 0 0 1
4 0 0 0 0 1 0
5 0 1 0 0 0 0
6 0 0 0 1 0 0
;
costociclo = 346171
```

La soluzione del problema P5 è un ciclo Hamiltoniano

$$(1, 3, 6, 4, 5, 2, 1)$$

Abbiamo determinato la seconda soluzione ammissibile per il problema il cui valore della funzione obiettivo è 346,171 Km. Possiamo affermare che non si tratta della soluzione ottima poiché la soluzione del problema P4 risulta essere migliore.

### 5.3.7 P6 (ottenuto da P2)

P6 è stato ottenuto dal problema P2 settando  $x_{46} = 1$  e  $x_{64} = 0$  e quindi impostando  $c_{4j} = 1000000 \forall j \neq 6$ , e  $c_{64} = 1000000$ .

	1	2	3	4	5	6
1	0	1000000	39860	1000000	1000000	1000000
2	26460	0	63240	115590	74120	117660
3	1000000	63210	0	77220	115520	74350
4	1000000	1000000	1000000	0	1000000	8790
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

La soluzione ottenuta è la seguente:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6    :=
1 0 0 1 0 0 0
2 1 0 0 0 0 0
3 0 0 0 1 0 0
4 0 0 0 0 0 1
5 0 1 0 0 0 0
6 0 0 0 0 1 0
;
costociclo = 354121
```

Anche in questo caso abbiamo un ciclo Hamiltoniano e quindi una soluzione ammissibile per il problema.

$$(1, 3, 4, 6, 5, 2, 1)$$

Il corrisponde valore della funzione obiettivo è 354,121 Km. Notiamo che la soluzione del problema P6 è peggiore di quella ottenuta dal problema P4 e quindi non sarà la soluzione ottima.

Di seguito si procederà nella risoluzione dei problemi figli P7, P8 e P9 ottenuti in corrispondenza del problema padre P3.

### 5.3.8 P7 (ottenuto da P3)

Il problema P7 è ottenuto dal problema P3 impostando  $x_{36} = 0$  e quindi settando  $c_{36} = 1000000$ .

	1	2	3	4	5	6
1	0	26190	1000000	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	39760	63210	0	77220	115520	1000000
4	96710	115630	76780	0	121751	1000000
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	9000	127041	0

La soluzione ottenuta è la seguente:

```
CPLEX 12.6.3.0: x [*,*]
:   1   2   3   4   5   6      :=
1  0   1   0   0   0   0
2  0   0   0   0   1   0
3  1   0   0   0   0   0
4  0   0   1   0   0   0
5  0   0   0   0   0   1
6  0   0   0   1   0   0
;
costociclo = 353250
```

Non sono presenti sotto-cicli e quindi abbiamo una soluzione ammissibile per il problema. Il ciclo Hamiltoniano determinato

$$(1, 2, 5, 6, 4, 3, 1)$$

non è il percorso ottimo poiché il valore della funzione obiettivo risulta essere maggiore rispetto a quello ottenuto risolvendo il problema P4.

### 5.3.9 P8 (ottenuto da P3)

Il problema P8 è ottenuto sempre in corrispondenza del problema P3 ma impostando  $x_{36} = 1$  e  $x_{64} = 0$  e quindi si modificherà la matrice dei costi sostituendo  $c_{3j} = 1000000 \forall j \neq 6$  e  $c_{64} = 1000000$ .

	1	2	3	4	5	6
1	0	26190	1000000	96640	87270	98710
2	26460	0	63240	115590	74120	117660
3	1000000	1000000	0	1000000	1000000	74350
4	96710	115630	76780	0	121751	1000000
5	87420	74750	115770	122080	0	127400
6	98670	117590	73830	1000000	127041	0

La soluzione restituita dal programma:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6    :=
1 0 1 0 0 0 0
2 1 0 0 0 0 0
3 0 0 0 0 0 1
4 0 0 0 0 1 0
5 0 0 0 1 0 0
6 0 0 1 0 0 0
;
costociclo = 444661
```

Sono presenti i sotto-cicli

$$(1, 2, 1), (3, 6, 3), (4, 5, 4)$$

e il valore della funzione obiettivo è 444,661. Notiamo che la soluzione ottenuta risulta essere maggiore della migliore soluzione trovata fino ad ora e che corrisponde con 344,770 ricava risolvendo P4. Sapendo che i valori delle funzioni obiettivo ottenute alle varie iterazioni sono un lower bound della soluzione ottima, risulterebbe inutile generare i problemi figli a partire da P8, poiché i valori delle soluzioni risulteranno essere maggiori di 444,661.

### 5.3.10 P9 (ottenuto d P3)

Il problema P9 è ottenuto da P3 imponendo  $x_{36} = 1$  e  $x_{43} = 0$  modificando la matrice dei costi inserendo  $c_{3j} = 1000000 \forall j \neq 6$  e  $c_{43} = 1000000$ .

	1	2	3	4	5	6	
1	0	26190	1000000	96640	87270	98710	
2	26460	0	63240	115590	74120	117660	
3	1000000	1000000	0	1000000	1000000	74350	
4	96710	115630	1000000	0	121751	1000000	
5	87420	74750	115770	122080	0	127400	
6	98670	117590	73830	9000	127041	0	

La soluzione che si ottiene:

```
CPLEX 12.6.3.0: x [*,*]
: 1 2 3 4 5 6   :=
1 0 1 0 0 0 0
2 1 0 0 0 0 0
3 0 0 0 0 0 1
4 0 0 0 0 1 0
5 0 0 1 0 0 0
6 0 0 0 1 0 0
;
costociclo = 373521
```

Sono presenti i sotto-cicli

$$(1, 2, 1), (3, 6, 4, 5, 3)$$

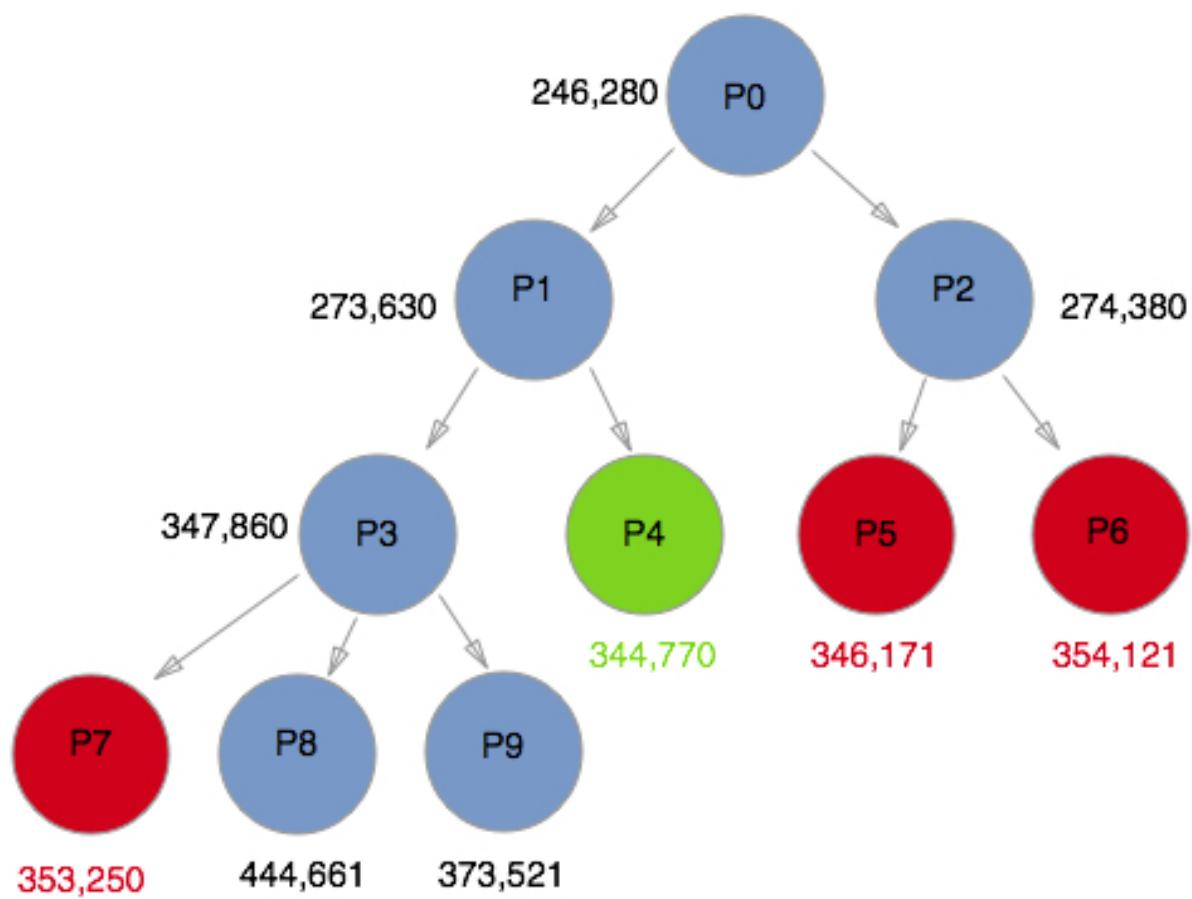
e il valore della funzione obiettivo è 373,521. Come per il caso del problema P8 è inutile generare i problemi figli poiché la soluzione risulta essere maggiore del miglior valore identificato risolvendo il problema P4.

### 5.3.11 Soluzione ottima

La soluzione ottima ammissibile per il problema è stata ottenuta risolvendo il problema P4. Il ciclo Hamiltoniano ottimo determinato è

$$(1, 2, 5, 4, 6, 3, 1)$$

con il corrispondente valore della funzione obiettivo 344, 770 Km. In seguito, verrà rappresentato l'albero ottenuto risolvendo il problema attraverso il Branch & Bound. In rosso sono selezionati i problemi che ammettono una soluzione ammissibile mentre in verde viene identificato il problema da cui si ottiene la soluzione ottima.



## 5.4 Nearest Neighbor e Repetitive Nearest Neighbor

Per risolvere il problema attraverso i metodi euristici Nearest neighbor e Repetitive Nearest Neighbor, verrà eseguito il programma in Java precedentemente descritto. Nel momento in cui l'applicazione sarà lanciata, verrà richiesto il primo parametro di input che corrisponde al numero di nodi all'interno del grafo. Successivamente compare un menù in cui si richiede all'utente di scegliere il metodo risolutivo da utilizzare. Inserendo (1) il problema verrà risolto tramite il NN e successivamente sarà richiesto un secondo input che è il nodo di partenza . Inserendo (2) invece, si eseguirà direttamente la procedura risolutiva del RNN. Dato che in questo caso l'algoritmo prevede la risoluzione del problema considerando ogni volta un nodo di partenza differente, questo non sarà richiesto all'utente come parametro. In entrambi i casi sarà richiesto l'inserimento della matrice dei costi ed infine il software si preoccuperà di risolvere il problema e a stampare a video la soluzione determinata. In seguito, verrà presentato l'output del programma prima risolvendo il problema dato scegliendo l'algoritmo NN e successivamente il RNN.

### 5.4.1 Risoluzione con il metodo NN

L'interfaccia che si presenta non appena il software viene mandato in esecuzione risulta essere semplice e intuitiva. L'utente deve soltanto inserire i valori richiesti correttamente e premere invio per confermare.

```
Inserisci il numero di nodi presenti nel grafo:  
6  
Scegli l'algoritmo che vuoi applicare:  
1- Nearest Neighbour  
2- Ripetitive Nearest Neighbour  
1  
Inserisci il nodo da cui partire:  
5  
Inserisci la matrice dei costi:  
0 26190 39860 96640 87270 98710  
26460 0 63240 115590 74120 117660  
39760 63210 0 77220 115520 74350  
96710 115630 76780 0 121751 8790  
87420 74750 115770 122080 0 127400  
98670 117590 73830 9000 127041 0  
Ciclo Hamiltoiano determinato: [5 2 1 3 6 4 5 ]  
Il costo associato: 346171.0
```

Si è scelto di partire dal quinto nodo. La decisione è stata puramente casuale dato che il NN è un algoritmo euristico che non ci assicura la soluzione ottima.

Confermando la matrice dei costi inserita, se non presenta errori formattazione, l'applicazione esegue l'algoritmo e stampa i risultati ottenuti.

La sequenza di nodi che ha inizio e fino con il vertice 5 è la seguente:

$$(5, 2, 1, 3, 6, 4, 5)$$

con valore della funzione obiettivo 346,171 Km.

Rispetto al costo determinato tramite il metodo esatto (344,770 Km) abbiamo una leggera differenza. L'errore relativo tra la soluzione ricava attraverso il metodo euristico NN e quello ottenuto con il metodo esatto è dell'0,406 %. Un errore accettabile in confronto della diversa complessità computazione dei due metodi.

### 5.4.2 Risoluzione con il metodo RNN

Mandando di nuovo in esecuzione il programma si risolve il problema scegliendo questa volta come metodo risolutivo il RNN. L'interfaccia che si presenta è sempre la stessa, ma in questo caso l'applicazione non richiederà, come input il nodo di partenza. Infatti, il RNN determina tutti i cammini Hamiltoiani considerando ogni volta come nodo di partenza un vertice differente. In seguito viene mostrato ciò che il software offre dopo l'esecuzione.

```
Inserisci il numero di nodi presenti nel grafo:  
6  
Scegli l'algoritmo che vuoi applicare:  
1- Nearest Neighbour  
2- Ripetitive Nearest Neighbour  
2  
Inserisci la matrice dei costi:  
0 26190 39860 96640 87270 98710  
26460 0 63240 115590 74120 117660  
39760 63210 0 77220 115520 74350  
96710 115630 76780 0 121751 8790  
87420 74750 115770 122080 0 127400  
98670 117590 73830 9000 127041 0  
Ciclo hamiltoniano considerando 1 come nodo di partenza: [1 2 3 6 4 5 1 ]  
Il costo associato: 381951.0  
  
Ciclo hamiltoniano considerando 2 come nodo di partenza: [2 1 3 6 4 5 2 ]  
Il costo associato: 346171.0  
  
Ciclo hamiltoniano considerando 3 come nodo di partenza: [3 1 2 5 4 6 3 ]  
Il costo associato: 344770.0  
  
Ciclo hamiltoniano considerando 4 come nodo di partenza: [4 6 3 1 2 5 4 ]  
Il costo associato: 344770.0  
  
Ciclo hamiltoniano considerando 5 come nodo di partenza: [5 2 1 3 6 4 5 ]  
Il costo associato: 346171.0  
  
Ciclo hamiltoniano considerando 6 come nodo di partenza: [6 4 3 1 2 5 6 ]  
Il costo associato: 353250.0  
  
Ciclo Hamiltoiano ottimo con nodo di partenza 3: [ 3 1 2 5 4 6 3 ]  
Ciclo Hamiltoiano ottimo con nodo di partenza 4: [ 4 6 3 1 2 5 4 ]  
Il costo ottimo determinato: 344770.0
```

Non appena la matrice dei costi viene confermata, il software risolve il problema e mostra tutte le possibili soluzioni. Dopo che sono stati ricavati tutti i cammini Hamiltoiani con un nodo di partenza differente, vengono stampate a video le soluzioni migliori. In questo esempio due sono i percorsi Hamiltoiani che presentano il miglior valore della funzione obiettivo.

(3, 1, 2, 5, 4, 6, 3)

(4, 6, 3, 1, 2, 5, 4)

Il costo relativo delle sequenze ottime determinate è 344,770 Km.

In questo caso la soluzione ottima determinata con il metodo euristico è uguale al percorso ottimo ricavato con i metodi esatto e quindi l'errore è 0.

## 6. Conclusioni

Il caso reale presentato si tratta di una semplice istanza del problema del Commesso Viaggiatore. L'efficienza e l'efficacia degli algoritmi esatti e euristici risultano essere pressoché identici. In generale, sappiamo che la complessità computazione di un algoritmo esatto risulta essere più elevata rispetto ad un algoritmo euristico. L'algoritmo della generazione dei vincoli ha una complessità computazionale esponenziale. Infatti, la convergenza del metodo viene garantita dal fatto che, al limite, si aggiungeranno tutti i vincoli di eliminazione dei sotto-cicli. Il metodo del Branch & Bound, allo stesso modo del metodo della generazione dei vincoli, possiede una complessità computazionale esponenziale. La situazione può essere migliorata dal fatto che invece di inserire esplicitamente i vincoli di eliminazione dei sotto-cicli si va a modificare la matrice dei costi in modo tale da non perdere la totale unimodularità della matrice dei vincoli, così da poter risolvere il problema attraverso un metodo efficiente (simplesso o ungherese). Inoltre, grazie alla tecnica di "bounding", è possibile escludere un numero di elevato di sotto-problemi. Considerando i metodi euristici NN e RNN, abbiamo un miglioramento della complessità computazionale a discapito dell'esattezza del metodo. Infatti, la complessità computazionale del NN è  $\Theta(n^2)$  mentre quella del RNN è di ordine  $\Theta(n^3)$  ma, la soluzione ottenuta attraverso il NN considerando il nodo 5 come nodo di partenza, non corrisponde con la soluzione ottima.

In conclusione, possiamo affermare che i metodi esatti presentati garantiscono la determinazione della soluzione ottima, ma l'occupazione spaziale e la complessità temporale assumono valori molti elevati considerando un numero di nodi maggiore di 50. D'altro canto l'utilizzazione degli algoritmi euristici, pur non garantendo la determinazione della soluzione ottima, offrono risultati accettabili con tempi di calcolo ristretti ed un limitato uso delle risorse. Diversi sono i test effettuati utilizzando istanze del problema con un numero elevato di nodi. Abbiamo notato che sia il NN che il RNN consentono di ottenere una soluzione ammissibile in tempi brevissimi considerando istanze fino a 200 nodi. I test in riferimento al metodo della generazione dei vincoli sono stati effettuati considerando istanze di al massimo 20 nodi a causa della versione demo di AMPL che non permette di risolvere problemi con più di 500 variabili. Tuttavia, testando l'algoritmo con 20 nodi i tempi di risposta sono brevi a discapito di un'elevata percentuale di utilizzo delle risorse.