**ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA**

---

**SCUOLA DI INGEGNERIA E ARCHITETTURA**

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE "GUGLIELMO MARCONI" - DEI

*CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA*

**TESI DI LAUREA**

in

Algoritmi di ottimizzazione L-S

# Heuristic Algorithms for the Travelling Deliveryman Problem

CANDIDATO
Ahmad Hariri

RELATORE:
Chiar.mo Prof. Paolo Toth

CORRELATORE
Dott. Roberto Roberti

Anno Accademico 2012/13

Sessione III

# INDICE

# CAPITOLO 1
# INTRODUZIONE AL PROBLEMA

## 1.1- Descrizione del problema:

Il *Travelling Deliveryman Problem* (TDP) è un problema NP-difficile, in cui si deve determinare un circuito che visiti una sola volta tutti i nodi di un grafo (circuito Hamiltoniano) in modo da minimizzare il tempo totale di attesa dei nodi clienti per essere visitati, cioè il "peso" di un arco dipende dal suo "costo" e dalla sua posizione nel circuito. Alcune applicazioni del TDP possono essere trovate nei problemi di schedulazione di macchine, nell'instradamento di automi guidati in sistemi di produzione flessibili, nei servizi di consegna a domicilio, nella logistica per gli aiuti di emergenza Roberti e Mingozzi (2013).

Il TDP è conosciuto in letteratura sotto altri nomi come, per esempio, il *travelling repairman problem*, il *minimum latency problem*, *cumulative TSP* e *school bus driver problem* Roberti e Mingozzi (2013).

Ci sono molti metodi sia esatti che euristici per risolvere il TDP. Nel secondo capitolo considereremo due metodi esatti e due metodi euristici per risolvere il TDP. In particolare nella categoria dei metodi esatti saranno illustrati il metodo di Abeledo et al. (2012), che introduce diverse famiglie di tagli che vengono usati insieme all'algoritmo branch-cut-and-price per trovare la soluzione ottima, poi vedremo il metodo di Roberti e Mingozzi (2013) dove vengono introdotte alcune procedure per calcolare dei lower bound stretti, il migliore dei quali viene usato per calcolare la soluzione ottima.

Per quanto riguarda invece i metodi euristici saranno esaminati l'algoritmo di Salehipour et al. (2011), dove vengono definiti due metodi metaeuristici, i primi in letteratura basati sulla costruzione di una soluzione iniziale casuale, non ottima, nella prima fase che, in una

seconda fase, viene raffinata tramite degli scambi tra i nodi corrispondenti ai clienti. Infine l'algoritmo di Silva et al. (2012) dove viene introdotto un altro algoritmo euristico diviso in due fasi: come nel caso precedente nella prima fase viene generata una soluzione ammissibile e nella seconda fase questa soluzione viene raffinata ed alla fine viene restituita la migliore soluzione trovata. Questi algoritmi differiscono tra loro nel modo in cui lavorano nella seconda fase.

**1.2- Formulazione del problema:**

Sia $G = (V, A)$ un grafo completo e orientato, dove $V = \{0, 1, \ldots, n\}$ è un insieme di nodi. I nodi $1, 2, \ldots, n$ saranno chiamati nel corso di questa tesi nodi *clienti* mentre il nodo 0 è il *deposito* di partenza. $A$ è l'insieme degli archi del grafo $G$ che collegano tra loro tutti i nodi appartenenti a $V$ l'uno all'altro. L'obiettivo del TDP consiste nel trovare un circuito Hamiltoniano sul grafo $G$,

$$x = (x_0 = 0, x_1, x_2, x_3, \ldots, x_n, x_0 = 0),$$

tale che, partendo dal deposito, $x_0 = 0$, minimizzi il tempo di attesa di tutti i clienti. Assumiamo che si viaggi con una velocità costante $v$, e indichiamo con $d(i, j)$ la distanza associata all'arco che collega i nodi $i$ e $j$. Allora il tempo impiegato per raggiungere un cliente $i$ sarà il rapporto tra la sua distanza dal deposito, lungo un determinato percorso, e la velocità $v$. La distanza percorsa per raggiungere il cliente $x_1$ è $\delta_1 = d(x_0, x_1)$, in modo analogo la distanza per raggiungere il cliente $x_2$ è $\delta_2 = d(x_0, x_1) + d(x_1, x_2)$. Per raggiungere il cliente $x_k$ la distanza percorsa sarà

$$\delta_k = d(x_0, x_1) + d(x_1, x_2) + \ldots + d(x_{k-1}, x_k).$$

Quindi il tempo corrispondente per raggiungere il k-esimo cliente è uguale a $t_k = \delta_k / v$. Di conseguenza la funzione obiettivo sarà la somma dei tempi impiegati per raggiungere tutti i clienti,

$$\sum_{k=1}^{n} t_k. \qquad (0)$$

Quest'ultima formula può essere riscritta nella seguente maniera:

$$\sum_{k=1}^{n} \frac{(n-k+1)d(x_{k-1},x_k)}{v} \qquad (1)$$

Poiché la velocità *v* è costante, essa non esercita alcuna influenza nella determinazione della soluzione ottima. Si può omettere la velocità nella funzione obiettivo che quindi diventa

$$\sum_{k=1}^{n}(n-k+1)d(x_{k-1},x_k) \qquad (2)$$

Si noti qui che nella funzione obiettivo non è stato preso in considerazione il tempo del ritorno al *deposito*. Alla fine di questo capitolo sarà mostrato come si possono modificare le suddette equazioni per tenere conto del ritorno al *deposito*.
In Mladenović et al. (2012) viene usata la seguente formulazione. Siano $X_{ij}$ e $Y_{ij}$ due insiemi di variabili tali che:

$$X_{ij} = \begin{cases} 1, & se\ viene\ attraversato\ l'arco(i,j) \in A \\ 0, & altrimenti \end{cases}$$

$$Y_{ij} = \begin{cases} 0, & se\ l'arco\ (i,j)\ \in A\ non\ viene\ attraversato \\ n-k, & se\ l'arco\ (i,j)\ \in A\ \text{è}\ il\ k-esimo\ arco\ nel\ tour \end{cases}$$

Quindi il modello matematico risulta essere il seguente problema di programmazione mista a numeri interi (MIP):

$$\min \sum_{i=0}^{n}\sum_{j=0}^{n} d(i,j)Y_{ij} \qquad (3)$$

t.c.

$$\sum_{j=0,j\neq i}^{n} X_{ij} = 1 \qquad i \in V \qquad (4)$$

$$\sum_{i=0,i\neq j}^{n} X_{ij} = 1 \qquad j \in V \qquad (5)$$

$$\sum_{i=1}^{n} Y_{0\mathrm{i}} = n \qquad (6)$$

$$\sum_{i=0,i\neq k}^{n} Y_{ik} - \sum_{i=0,i\neq k}^{n} Y_{ki} = 1 \qquad k \in V \setminus \{0\} \qquad (7)$$

$$Y_{i0} = 0 \qquad i \in V \setminus \{0\} \qquad (8)$$

$$Y_{0\mathrm{i}} \leq nX_{0\mathrm{i}} \qquad i \in V \setminus \{0\} \qquad (9)$$

$$Y_{ij} \leq (n-1)X_{ij} \qquad i,j \in V \setminus \{0\};\ i \neq j \qquad (10)$$

$$X_{ij} \in \{0,1\} \qquad i,j \in V \setminus \{0\};\ i \neq j \qquad (11)$$

$$Y_{ij} \in \{0,\ldots,n\} \qquad i,j \in V \setminus \{0\};\ i \neq j \qquad (12)$$

Questa formulazione contiene $2n(n+1)$ variabili e $n^2 + 4n + 3$ vincoli. Di seguito saranno descritti dettagliamente i singoli vincoli:

(4): Dato un nodo *j* assicurarsi che ci sia un solo arco in uscita;

(5): Simile al vincolo precedente ma per gli archi entranti in un determinato nodo;

(6): L'arco che esce dal deposito sarà preso in considerazione *n* volte;

(7): Questo vincolo è per fare in modo che il peso di ciascuno arco nel cammino Hamiltoniano diminuisca di uno rispetto a quello dell'arco precedente, in altre parole, per imporre il coefficiente "*n* – *k*" alla distanza tra i clienti *i* e *j* quando sono nella k-esima posizione lungo il cammino individuato. Questi vincoli insieme al vincolo (6) sono vincoli di flusso e fanno in modo che non ci siano circuiti parziali nel cammino finale individuato;

(8): In questa formulazione non è stato preso in considerazione il ritorno al deposito per il calcolo del cammino ottimo, e quindi questi vincoli servono per azzerare il costo del ritorno;

(9) – (10): Sono vincoli per mettere in relazione logica le variabili $X_{ij}$ e $Y_{ij}$, per esempio se, per *i* e *j* fissi, se $X_{ij} = 0$ allora anche $Y_{ij}$ deve essere = 0;

(11) – (12): Definiscono i domini delle variabili del problema.

Come già detto la formulazione precedente non prende in

considerazione il ritorno al deposito. Per poterlo fare dobbiamo modificare alcune equazioni, in particolare le equazioni (0), (1), (2) e (3). Prima però definiamo due nuove variabili, $\delta_0$ come la distanza del circuito Hamiltoniano individuato. Cioè $\delta_0 = d(x_0, x_1) + d(x_1, x_2) + \ldots + d(x_{n-1}, x_n) + d(x_n, x_0)$, e in modo analogo $t_0 = \delta_0 / v$. A questo punto dobbiamo solo aggiungere la componente $t_0$ all'equazione (0) che diventa

$$\sum_{k=1}^{n} t_k + t_0 \qquad (0')$$

Possiamo derivare la seguente equazione da (1) e (0'), quando sommiamo a $t_k$ la componente $t_0$, ciascun elemento della sommatoria di (1) viene incrementato di uno e consideriamo a parte la distanza dal nodo *n* al *deposito*, il tutto diviso per la velocità *v*

$$\sum_{k=1}^{n} \frac{(n-k+2)d(x_{k-1},x_k)+d(x_n,\ x_0)}{v} \qquad (1')$$

In modo analogo, non consideriamo la velocità *v* visto che è una costante e non influenza quindi il calcolo della soluzione ottima

$$\sum_{k=1}^{n}(n-k+2)d(x_{k-1},x_k) + d(x_n,\ x_0) \qquad (2')$$

Per quanto riguarda invece il modello matematico viene modificata soltanto la funzione obiettivo (3) in modo da aggiungere la componente $\delta_0$. A questo punto otteniamo

$$\min \sum_{i=0}^{n}\sum_{j=0}^{n} d(i,j)Y_{ij} + \sum_{i=0}^{n}\sum_{j=0}^{n} d(i,j)X_{ij} \qquad (3')$$

i vincoli restano identici al caso in cui non si considera il ritorno al *deposito*.

La tesi sarà strutturata nel modo seguente. Nel secondo capitolo saranno

esaminati quattro metodi di risoluzione del TDP, tra cui due metodi esatti e due metodi euristici. Nel terzo capitolo verrà spiegato in modo esteso l'algoritmo proposto in Mladenović et al. (2012) oggetto di questa tesi, mentre nel quarto capitolo verrà descritta la sua implementazione. Ed infine nel quinto capitolo saranno illustrati i risultati computazionali condotti sul modello matematico precedentemente definito, su una sua variazione che tiene conto del tempo di attesa del ritorno al deposito e sull'euristico descritto al terzo capitolo.

# CAPITOLO 2
# LETTERATURA

Viste le numerose applicazioni di cui il TDP gode, diverse ricerche sono state condotte per trovare metodi per la sua risoluzione. Questi metodi vengono classificati in due categorie: 1) metodi esatti e 2) metodi euristici. In questo capitolo saranno illustrati due metodi appartenenti a ciascuna delle suddette categorie.

## 2.1- Metodi esatti

Questi metodi risolvono il problema alla sua ottimalità, e sono caratterizzati dal loro tempo di calcolo e dalla dimensione massima delle istanze che riescono a risolvere.

### 2.1.1- *Branch-cut-and-price* (Abeledo et al. 2012)

In questo lavoro gli autori hanno studiato il politopo associato alla formulazione del Time Dependent TSP proposta da Picard e Queyranne (1978). Dopo aver determinato la dimensione del suddetto politopo, gli autori hanno individuato alcune famiglie di tagli che hanno usato insieme ad un algoritmo *Branch-Cut-and-Price* (BCP). Gli autori iniziano la fase di risoluzione con il BCP eseguendo un algoritmo euristico chiamato *Iterated Local Search* (ILS), che opera spostando un vertice da una posizione ad un'altra, facendo lo *swap* di un paio di vertici ed invertendo i vertici appartenenti ad un circuito parziale. Quindi la soluzione corrente viene perturbata e si parte con una nuova iterazione, il numero totale di iterazioni è uguale a $n^2$. Questo euristico riesce a trovare dei buoni bound, e a volte la soluzione ottima. Infine gli autori sono riusciti a risolvere istanze con un numero di nodi che arriva fino a 107.

### 2.1.2- *Dynamic ng-path relaxation* (Roberti, Mingozzi 2013)

Questo lavoro parte da una ricerca condotta in un tempo precedente da

Roberti et al. (2011) dove gli autori hanno introdotto l'algoritmo *ng-path relaxation* per calcolare dei lower bound stretti per il *Vehicle Routing Problem* (VRP) cercando di risolvere un rilassamento della sua formulazione secondo il set-partitioning, dove le route non sono elementari e possono contenere dei circuito parziale. La qualità dei lower bound trovati dipende dai circuito parziale. In Roberti e Mingozzi (2013) gli autori hanno introdotto una nuova procedura chiamata *dynamic ng-path relaxation* per risolvere il TDP. Questa procedura si basa sulla generazione di colonne e calcola una sequenza di lower bound non decrescenti per il problema. Il miglior lower bound trovato viene poi usato per trovare la soluzione ottima tramite la programmazione dinamica, cercando di espandere il grafo ng-path iterativamente in modo che, a ciascuna iterazione, il grafo trovato non contenga alcuni dei loop individuati nell'iterazione precedente. I risultati computazionali hanno mostrato dei miglioramenti rispetto al metodo proposto da Abeledo et al. (2012). L'algoritmo è anche capace di chiudere cinque istanze aperte con un numero di nodi che arriva fino a 152.

## 2.2- Metodi euristici

2.2.1- *Efficient GRASP+VND and GRASP+VNS* (Salehipour et al. 2011)

Questo è un algoritmo multi-start in cui ogni iterazione consiste in due fasi: una fase di costruzione di una soluzione in una maniera greedy e casuale, ed una seconda fase di miglioramento in cui si esegue una ricerca locale per trovare un minimo locale. *Greedy Randomized Adaptive Search Procedure* (GRASP) è una procedura che introduce un elemento di casualità controllata in un euristico puramente greedy. La procedura GRASP viene usata in combinazione con il *Variable Neighborhood Serach* (VNS) e *Variable Neighborhood Descent* (VND). VNS si basa sul principio di esplorazione di *neighborhood* diversi,

insieme ad una fase di perturbazione, chiamata *shake*¸ per passare da un *neighborhood* ad un altro. Il VNS può essere applicato per risolvere una varietà di problemi tra cui il VRP. Il VND è una variante deterministica del VNS che omette la fase di *shake*, ed usa cinque strutture di *neighborhood*: 1) *swap adjacent (1-opt)*; 2) *swap*; 3) *2-opt*; 4) *Or-opt*; 5) *drop/add*. Le strutture (1) e (3) saranno esaminati nel terzo capitolo.

2.2.2- *GILS-RVND* (Silva et al., 2012)

Anche questo algoritmo si basa su due fasi. Nella prima fase come nel caso precedente, si usa la procedura GRASP per generare una soluzione, e nella seconda fase si cerca di migliorare questa soluzione usando l'algoritmo RVND, cioè VND con un ordinamento casuale dei *neighborhood*, e l'algoritmo ILS che serve come meccanismo di perturbazione della migliore soluzione trovata con RVND. Si ripete la seconda fase per un determinato numero di iterazioni. Si costruisce una soluzione iniziale mettendo in prima posizione il nodo *deposito* e poi si sceglie a caso uno tra un determinato numero dei nodi più vicini all'ultimo nodo inserito. Poi si cerca in un insieme di *neighborhood* della soluzione corrente per trovare una soluzione migliore, in particolare l'insieme dei *neighborhood* contiene le seguenti cinque strutture: 1) *swap* in cui due *clienti* nel circuito vengono scambiati tra di loro; 2) *2-opt* dove vengono eliminati due archi non adiacenti ed altri due inseriti in modo da creare un nuova soluzione ammissibile; 3) *Reinsertion* che cambia la posizione di un *cliente* nel circuito; 4) *Or-opt2* in cui due *clienti* adiacenti sono spostati in una altra posizione nel circuito; 5) *Or-opt3* in modo del tutto analogo a prima ma per tre *clienti* adiacenti. Inoltre gli autori hanno introdotto delle procedure e strutture dati che permettono di fare il confronto tra il costo della soluzione e del *neighborhood* in $O(1)$, che invece richiederebbe $O(n^3)$ operazioni per un intera struttura di *neighborhood* nel caso si calcoli direttamente il suo costo.

# CAPITOLO 3
# GENERALIZED VNS

Nel paper oggetto di questa tesi, Mladenović et al. (2012), è stato proposto il Generalized VNS (GVNS) che è composto di due fasi. La prima è la generazione di una soluzione in maniera simile al GRASP. Nella seconda fase si fa una ricerca locale per trovare una miglior soluzione. Gli autori hanno elaborato due metodi di ricerca VNS: *sequential-VNS* e *mixed-VNS*, dove nel primo si usano quattro strutture di *neighborhood* mentre il secondo usa anche una quinta struttura. Di seguito saranno illustrate tutte le strutture *neighborhood* usate.

## 3.1- Le strutture *neighborhood*

Il VNS è un euristico la cui idea di base è di cambiare *neighborhood* in cerca di una soluzione migliore. È un algoritmo iterativo, ed ogni iterazione consiste in tre passi: 1) perturbazione della soluzione incombente *x*; 2) ricerca locale; 3) cambiamento di *neighborhood*.

### 3.1.1- *2-opt*

Sia *x* una soluzione ammissibile per il TDP

$$x = (x_0, x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$$

la soluzione che appartiene al *neighborhood 2-opt* di *x* si ottiene eliminando gli archi $(x_i, x_{i+1})$ e $(x_j, x_{j+1})$. Il nuovo circuito *x*' diventa:

$$x' = (x_0, x_1, x_2, \ldots, x_{i-1}, x_i, x_j, x_{j-1}, \ldots, , x_{i+2}, x_{i+1}, x_{j+1}, \ldots, x_n) \qquad (13)$$

*Proprietà 1*: l'esplorazione completa del *neighborhood 2-opt* richiede un tempo $O(n^2)$

*Dimostrazione*: le funzioni obiettivo di *x* e *x*' sono

$$f(x) = n\, d(x_0, x_1) + \ldots + (n - i + 1)\, d(x_{i-1}, x_i) + (n - i)\, d(x_i, x_{i+1}) + (n - i)\, d(x_{i+1}, x_{i+2}) + \ldots + (n - j + 1)\, d(x_{j-1}, x_j) + (n - j)\, d(x_j, x_{j+1}) + (n - j - 1)\, d(x_{j+1}, x_{j+2}) + \ldots + d(x_{n-1}, x_n)$$

e

$$f(x') = n\, d(x_0, x_1) + \ldots + (n - i + 1)\, d(x_{i-1}, x_i) + (n - i)\, d(x_i, x_j) + (n - i - 1)\, d(x_j, x_{j-1}) + \ldots + (n - j + 1)\, d(x_{i+2}, x_{i+1}) + (n - j)\, d(x_{i+1}, x_{j+1}) + (n - j - 1)\, d(x_{j+1}, x_{j+2}) + \ldots + d(x_{n-1}, x_n)$$

sommando entrambi le equazioni otteniamo

$$f(x) + f(x') = 2 \sum_{k=0}^{i-1}(n - k)d(x_k, x_{k+1}) + 2 \sum_{k=j+1}^{n-1}(n - k)d(x_k, x_{k+1}) + (2n - i - j)\, \delta(x_{i+1}, x_j) + (n - i)\, (d(x_i, x_{i+1}) + d(x_i, x_j)) + (n - j)\, (d(x_j, x_{j+1}) + d(x_{i+1}, x_{j+1}))$$

dove $\delta(x_{i+1}, x_j)$ denota la distanza del circuito parziale che va dal *cliente* $x_{i+1}$ al *cliente* $x_j$. Generalizzando

$$\delta(x_i, x_j) = \sum_{h=i}^{j-1} d(x_h,\ x_{h+1})$$

Scegliendo opportunamente le strutture dati ed eseguendo una fase di preelaborazione quest'ultima formula può essere calcolata in tempo $O(1)$. Allo stesso tempo possiamo determinare se $x'$ è una soluzione migliore rispetto a $x$ visto che il suo costo, $f(x)$ è noto, e quindi possiamo calcolare $f(x')$.

Prima di iniziare la ricerca nel *neighborhood 2-opt* dobbiamo calcolare e memorizzare le seguenti sequenze:

$$S_b(i) = \sum_{k=0}^{i}(n - k)d(x_k, x_{k+1});$$

$$S_e(i) = \sum_{k=i}^{n-1}(n-k)d(x_k, x_{k+1})$$

e

$$\delta(i) = \sum_{k=1}^{i} d(x_k,\ x_{k+1})$$

Il tutto può essere calcolato in tempo $O(n)$. Se abbiamo questi valori allora la somma diventa

$$f(x) + f(x') = 2\,S_b(i-1) + 2\,S_e(j+1) + (2n - i - j)\,(\delta(j-1) - \delta(i)) + (n-i)$$
$$(d(x_i, x_{i+1}) + d(x_i, x_j)) + (n-j)\,(d(x_j, x_{j+1}) + d(x_{i+1}, x_{j+1}))$$

Questa somma può essere calcolata in un tempo costante $O(1)$ e di conseguenza l'intero *neighborhood 2-opt* viene esplorato in tempo $O(n^2)$.

Una volta trovata la migliore soluzione in *2-opt* dobbiamo aggiornare le strutture dati sopraelencate, la complessità di questa operazione è in $O(n)$. Di seguito è riportato lo pseudocodice della suddetta operazione.

Algoritmo 1: Aggiornare $S_b$, $S_e$ e $\delta$ in *2-opt*

```
Function UpdateSbSeδ(x, i, j);
for k ← i - 1  to  n - 1 do
|    Sb(k) ← Sb(k - 1) + (n - k) * d(xk, xk+1)
└    δ(k)  ← δ(k - 1) + d(xk, xk+1)

for k ← j downto 0 do
└    Se(k)  ← Se(k + 1) + (n - k) * d(xk, xk+1)
```

3.1.2- *1-opt*

Il *neighborhood 1-opt* è un caso speciale del *2-opt* dove l'indice $j$ è uguale a $i + 2$ in (13). Considerando quattro *clienti* consecutivi nel circuito $x$ corrente

$$x_k, x_{k+1}, x_{k+2}, x_{k+3}$$

Il nuovo circuito *x*' si ottiene scambiando tra di loro $x_{k+1}$ e $x_{k+2}$.

*Proprietà 2*: Esplorare completamente il *neighborhood 1-opt* richiede un tempo $O(n)$.

*Dimostrazione*: La differenza tra la nuova funzione obiettivo e quella vecchia è data da

$$\Delta f = (n - k)\,(d(x_k, x_{k+2}) - d(x_k, x_{k+1})) + (n - k - 2)\,(d(x_{k+1}, x_{k+3}) - d(x_{k+2}, x_{k+3}))$$

Se $\Delta f$ è negativa allora *x*' è un circuito migliore rispetto a *x*. Trovare $\Delta f$ richiede un tempo $O(1)$ e siccome la cardinalità del *neighborhood 1-opt* è $n - 3$ allora l'operazione di esplorazione dell' *1-opt* richiede un tempo $O(n)$.

3.1.3 *Move Forward (mf)*

Una soluzione che appartiene a questo *neighborhood* si ottiene spostando *k clienti* consecutivi a destra del circuito, con $k = 1, 2, ..., l_{max}$. Se il blocco di *k clienti* inizia dalla posizione $i$ +1 e viene inserito dopo la posizione *j*, cioè dopo il *cliente* $x_j$, allora il circuito

$$x = (x_0, x_1, ..., x_i, x_{i+1}, ..., x_j, x_{j+1}, ..., x_n)$$

diventa

$$x' = (x_0, x_1, ..., x_i, x_{i+k+1}, ...,x_j, x_{i+1}, ..., x_{i+k}, x_{j+1}, ..., x_n)$$

*Proprietà 3*: Una completa esplorazione del *neighborhood move-forward* richiede un tempo $O(n^2 l_{max})$.

*Dimostrazione*: La differenza tra le due funzioni obiettivo $f(x)$ e $f(x')$ è

$$\begin{aligned}\Delta f = {} & k\,\delta(x_{i+k+1,}\, x_j) - (j - i - k)\,\delta(x_{i+1,}\, x_{i+k}) \\ & + (n - i)\,(d(x_i,\, x_{i+k+1}) - d(x_i,\, x_{i+1})) \\ & + (n - j)\,(d(x_{i+k},\, x_{j+1}) - d(x_j,\, x_{j+1})) \\ & + (n - j + k)\,d(x_j,\, x_{i+1}) - (n - i - k)\,d(x_{i+k},\, x_{i+k+1})\end{aligned}$$

Se $i$ e $j$ sono fissi allora l'incremento di $k$ di 1 fa sì che il calcolo della nuova $\Delta f$ sia in $O(1)$. Questo vale per tutte le coppie $(i, j)$. Allora trovare le funzioni obiettivo nell'intero *neighborhood move-forward* richiede un tempo $O(\binom{n}{2} l_{max}) = O(n^2\, l_{max})$.

In questa sezione considereremo più in dettaglio alcuni elementi della parte destra dell'equazione precedente:

- $k\,\delta(x_{i+k+1,}\, x_j)$: I *clienti* consecutivi $(x_{i+k+1,\,\ldots,}\, x_j)$ vengono spostati a sinistra del circuito di '$k$' posizioni e quindi il loro contributo nella funzione obiettivo aumenta proprio di $k$;
- $(j - i - k)\,\delta(x_{i+1,}\, x_{i+k})$: In modo del tutto analogo a quanto detto prima, il contributo nella funzione obiettivo dei '$k$' *clienti* che sono stati spostati a destra diminuisce di un fattore pari a '$j - i - k$';
- I restanti elementi sono dovuti alla cancellazione e aggiunta di *archi* tra due *clienti*.

3.1.4- *Move Backward (mb)*

Questo *neighborhood* è analogo a quello precedente, con la differenza che i $k$ *clienti* consecutivi sono spostati a sinistra. Se per esempio spostiamo $k$ *clienti* a partire dalla posizione $j + 1$ e li inseriamo dopo l'$i$-esimo *cliente* $(x_i)$, allora il circuito che otteniamo è

$$x' = (x_{0,}\, x_1,\, \ldots,\, x_i,\, x_{j+1},\, \ldots,\, x_{j+k},\, x_{i+1},\, \ldots,\, x_j,\, x_{j+k+1},\, \ldots,\, x_n)$$

e la differenza $\Delta f$ diventa

$$\begin{aligned}\Delta f = (j - i)\ \delta(x_{j+1,}\, x_{j+k}) - k\ \delta(x_{i+1,}\, x_j) \\ + (n - i)\ (d(x_i,\, x_{j+1}) - d(x_i,\, x_{i+1})) \\ + (n - j - k)\ (d(x_j,\, x_{j+k+1}) - d(x_{j+k},\, x_{j+k+1})) \\ + (n - i - k)\ d(x_{j+k},\, x_{i+1}) - (n - j)\ d(x_j,\, x_{j+1})\end{aligned}$$

Questi due *neighborhood*, *mf* e *mb*, possono essere considerati come un unico *neighborhood k-insertion*. Pertanto, insieme, possono essere visti come un estesione del *Or-opt*, dove *k clienti* consecutivi vengono spostati in un'altra posizione del circuito e *k* può assumere i valori 1, 2 e 3. Tuttavia gli autori hanno deciso di spezzare il *neighborhood k-insertion* in entrambi i suoi sottoinsiemi perché utile per risolvere altri problemi sui grafi.
Nel prossimo paragrafo vedremo lo pseudocodice dell'algoritmo usato per risolvere il TDP.

3.1.5- *Double Bridge* (*DB*)
Dati quattro indici *i, j, k* e *l* tali che $i < j < k < l$, gli archi $(x_i, x_{i+1})$, $(x_j, x_{j+1})$, $(x_k, x_{k+1})$ e $(x_l, x_{l+1})$ vengono eliminati e rimpiazzati da 4 nuovi archi $(x_i, x_{k+1})$, $(x_l, x_{j+1})$, $(x_k, x_{i+1})$ e $(x_j, x_{l+1})$. La dimensione di questo *neighborhood* è $O(n^4)$, tuttavia fissando *j* e *l* ai valori $i + 2$ e $k + 2$ rispettivamente la sua dimensione viene ridotta fino a $O(n^2)$. Il *double-bridge* sarà usato nell'algoritmo *Mix-VND* che sarà esaminato più avanti.

**3.2- General VNS**
Il *General VNS* (GVNS) è una variante del VNS dove VND viene usato come una routine di ricerca locale. Contiene tre parametri: 1) $t_{max}$ – indica il tempo massimo di esecuzione concesso; $k_{max}$ – il numero dei *neighborhood* usati nella fase di *shake*; 3) $l_{max}$ – il numero totale dei

*neighborhood* usati in VND. Il suo pseudocodice è riportato nel box seguente.

Algoritmo 2: Passi del GVNS

```
Function GVNS(x, l_max, k_max, t_max)
x    Initial(x)
repeat
| k ← 1
| repeat
| | x' ← Shake(x, k)
| | x'' ← VND(x', l_max)
| | if f(x'') < f(x) then
| | | x ← x''; k ← 1
| | else
| | └ k ← k + 1
| until k > k_max
until t > t_max
return x
```

I *neighborhood* usati nelle funzioni *Shake* e *VND* sono specifici al problema.

3.2.1- *Soluzione iniziale*

Gli autori usano due modi per generare una soluzione iniziale. Il primo è ottenuto attraverso una permutazione casuale degli $n - 1$ *clienti*. Il secondo modo è tramite la procedura *Greedy randomized* data nell'*algoritmo 3*.

Algoritmo 3: Algoritmo *Greedy randomized*

```
Function Greedy-R(x, q)
S ← {1, 2, 3, …, n}
X_0 ← 0; k ← 1
While S ≠ Ø
| S' ← sottoinsieme di S con i min(q, n - k)
|          clienti più vicini a x_{k-1}
| x_k ← un cliente in S' scelto a caso
| S ← S\{x_k}
└ k ← k + 1
return x.
```

In questo algoritmo la scelta del prossimo *cliente* $x_k$ è legata al parametro *q*. Se *q* è minore del numero, $n - k$, di *clienti* rimasti allora sceglie uno dei *q clienti* più vicini a $x_{k-1}$. Altrimenti se q è maggiore di *n* oppure $n - k$ allora uno dei *clienti* rimasti viene scelto in modo casuale. Mettendo il valore di *q* uguale a quello di *n* questo algoritmo genera una soluzione in modo completamente casuale.

3.2.2- *Shake*

Questa procedura prende la soluzione incombente *x* e la perturba per ottenere una soluzione *x*' a caso dal *k*-esimo *neighborhood* di *x* ($x \in N_k(x)$, $k = 1, ..., k_{max}$). Tale soluzione viene data in input alla procedura di ricerca locale come parametro. Gli autori hanno deciso di usare gli *insertion neighborhood* per lo *shake*. Ad ogni iterazione viene scelto un *cliente i* a caso ed inserito fra i *clienti j* e $j + 1$, dove anche *j* è scelto a caso. Si ripete per *k* iterazioni.

Algoritmo 4: *Shake*

```
Function Shake(x, k)
While k > 0
| i ← Random(1, n) // i ∈ [1, n]
| j ← Random(1, n) // j ∈ [1, n]
| if i ≠ j then
| | inserire x_i fra x_j e x_{j+1}
| └ k ← k - 1
└
return x.
```

Questa mossa di inserimento consiste del *move-forward* oppure del *move-backward* a seconda se *i* sia maggiore o minore di *j*.

3.2.3- Algoritmi *VND*

L'ordine in cui verrano visitati i *neighborhood* è stato determinato come

segue: 1) *1-opt*; 2) *2-opt*; 3) *move-forward*; 4) *move-backward*. Nel *Mix-VND* viene usato un ulteriore *neighborhood* 5) *double-bridge*. Gli autori hanno anche ridotto la complessità della ricerca nel *neighborhood 2-opt* confinando il numero di *clienti* candidati agli *r* più vicini a quello correntemente sotto esame. In altre parole, fissato *i*, scegliamo *j* in modo che $x_j$ sia uno degli *r clienti* più vicini a $x_i$. Per poterlo fare serve fare un ordinamento sulla matrice dei costi *D*, cioè si crea una matrice rank *R* le cui righe rappresentano tutti i nodi *clienti*, e le cui colonne contengono gli indici di colonna della matrice *D*, che data una qualsiasi riga *i*, inserisce in prima posizione il *cliente* $x_j$ più vicino a $x_i$, in seconda posizione mettiamo il secondo *cliente* più vicino a $x_i$ e così via andando in ordine non decrescente di distanza. Quindi serve una routine che esamina il contenuto della matrice *D* senza modificarlo, e mette nella matrice *R* gli indici delle colonne riordinate dal punto di vista di ciascuna riga. Nel codice dell'implementazione questa routine è incorporata nella funzione che legge il file di input che contiene gli elementi della matrice *D*.

Sono stati sviluppati ed implementati due varianti del VND per risolvere il TDP: *Seq-VND* (v. Algoritmo 5) e *Mix-VND* (v. Algoritmo 6)

Algoritmo 5: *Sequential VND* (*seq-VND*)

```
Function Seq-VND (R, x', r)
l ← 1
While l < 4 do
| if l = 1 then
| | x'' ← argmin{f(x)| x ∈ N1-opt(x')}
| if l = 2 then
| | x'' ← argmin{f(x)| x ∈ N2-opt(R, x', r)}
| if l = 3 then
| | x'' ← argmin{f(x)| x ∈ Nmf(x')}
```

Algoritmo 5: segue

```
| if l = 4 then
| | x″ ← argmin{f(x)| x ∈ N_mb(x′)}
| if f(x′) < f(x″) then
| | x′ ← x″; l ← 1
| else
| └ l ← l + 1
└
return x′
```

Nel *Mix-VND* viene eseguito il *Seq-VND* per ciascuna soluzione appartenente al *neighborhood double-bridge* ($N_{db}$) della soluzione corrente *x*'. Se la soluzione *x*" trovata tramite *Seq-VND* è migliore di quella corrente *x*' allora *x*" diventa quella corrente e si passa ad una nuova iterazione (*x*' ← *x*"). L'idea alla base *Mix-VND* è di dare una posizione speciale al *double-bridge* vista la sua importanza nel risolvere il TSP.

Algoritmo 6: *Mixed VND* (*seq-VND*)

```
Function Mix-VND (R, x′, r)
improve ← true
While improve do
| improve ← false
| while ci sono soluzioni inesplorate in
|     N_db(x′) && !improve
| | x″ ← prossimo soluzione in N_db(x′)
| | x″ ← Seq-VND(R, x″, r)
| | if f(x′) > f(x″) then
| | | x′ ← x″
| | └ improve ← true
| └
└
Return x′
```

3.2.4- *Algoritmo GVNS*

Come già accennato nel paragrafo precedente, sono state sviluppate due varianti del GVNS. Una usa soltanto il VND sequenziale (v. *Algoritmo 5*) che sarà indicato come *GVNS-1*, la seconda usa il VND misto (v. *Algoritmo 6*), chiamato *GVNS-2*. Si ricorda che si può generare una soluzione iniziale in modo comepletamente casuale ponendo il parametro *q* uguale al valore del numero di nodi *n*, altrimenti (*q*<*n*) si ottiene il comportamento *Greedy randomized*.

Algoritmo 6: *General VNS*

```
Function GVNS (D, x, k_max, t_max)
R ← Rank(D)
x ← Greedy-R(x,10) (o Rand(x))
repeat
| k ←1
| repeat
| | x' ← Shake(x, k)
| | x'' ← Seq-VND(R,x',4) (o Mix-VND((R,x',4))
| | if f(x) > f(x'') then
| | | x ← x''; k ←1
| | else
| | └ k ← k + 1
| until k > k_max
until t > t_max
return x
```

Per il *Greedy-R* gli autori hanno usato 10 come valore per il parametro *q*, e 4 per il valore di $l_{max}$ nella ricerca locale. Quindi quest'ultimo non viene considerato come parametro come lo è stato nell'*algoritmo 2*.

# CAPITOLO 4
# IMPLEMENTAZIONE GVNS

L'algoritmo GVNS è stato implmentato seguendo fedelmente I pseudocodici presentati in Mladenović et al. (2012), con un paio di eccezioni. La prima è che non è stato implementato l'algoritmo *Rand(x)* che genera una soluzione iniziale attraverso una permutazione casuale dei *clienti* $x_i$, in quanto si può sfruttare il parametro *q* dell'algoritmo *Greey-R(x)* per ottenere questo comportamento. In secondo luogo il *neighborhood 2-opt* è stato esplorato completamente invece di quanto spiegato nel paragrafo 3.2.3.

Come linguaggio di programmazione è stato adottato il linguaggio C e il Visual Studio 2012 come ambiente di programmazione. Per facilitare la manipolazione degli array nell'esplorazione dei vari *neighborhood* e la generazione della soluzione inziale è stata creata una struttura dati chiamata *vector* ispirata dalla classe *Vector* di JAVA. Nei prossimi paragrafi saranno esaminate le varie funzioni che implementano il comportamento del GVNS, insieme all'esame di due metodi di *vector*. Ma per cominciare consideriamo l'input e output del programma *gvns.cpp*.

## 4.1- Input ed output

Alla partenza *gvns.cpp* va a leggere da un file di input i vari costi degli archi (*i, j*), chiamando la funzione *readInput(filename)*. Il file è organizzato nel seguente modo: il primo dato letto è un intero che dice di che *tipo* sono i *dati* contenuti nel file, questo sarà utile per il calcolo della matrice dei costi; il secondo è sempre un numero intero che indica il numero di *clienti* totale; e alla fine troviamo i *dati* veri e propri. L'intero *tipo* potrebbe indicare, per esempio, se quello che si sta per leggere è la distanza *euclidea*, *pseudo-euclidea* oppure se si tratta della distanza *Manhattan*, in questi casi i *dati* letti sono le coordinate bidimensionali o tridimensionali dei vari *nodi* e da lì viene calcolata la

matrice dei costi *D*. Il file potrebbe anche contenere direttamente gli elementi della matrice *D*, o la triangolare superiore della matrice *D*.
Una volta letto il file di input, viene creata le matrici *D*, fatta la rank sulle distanze (la matrice *R*) e memorizzato il numero dei nodi *n*. A questo punto parte la ricerca della soluzione ottima. Ogni volta che viene trovata una nuova soluzione, vengono stampati su terminale la soluzione, intesa come circuito Hamiltoniano, il suo costo, e il tempo impiegato per trovarla. Alla scadenza del tempo concesso la migliore soluzione trovata fino a quel momento viene scritta su un file di testo.

### 4.2- La struttura dati vector

Per facilitare la manipolazione degli array contenenti gli indici dei nodi che compongono un cammino è stata creata la struttura dati *vector*. È da intendersi come una sostituzione agli array, si comporta come tale ma in più ha la possibilità di aumentare automaticamente la sua dimensione quando raggiunge il suo limite e contiene dei metodi che permettono di fare delle operazioni di riordinamento sui suoi elementi come, per esempio, spostare un elemento, o un sottoinsieme di elementi consecutivi, da una posizione ad un altra all'interno della struttura dati, oppure invertire un segmento di elementi consecutivi.
Questa struttura dati è stata usata nell'esplorazione dei *neighborhood move-forward* e *move-backward* in cui si deve cambiare la posizione di un determinato numero di *clienti* consecutivi. Scelti i *clienti* da spostare, questi vengono salvati in un array temporaneo e eliminati dalla stuttura dati *vector*. Nel caso del *move-forward* si aggiorna l'indice destinazione dello spostamento, altrimenti non ce n'è bisogno. Questo perché quando vengono eliminati gli elementi da spostare, vengono spostati all'indietro quelli successivi e quindi per inserirli nella posizione voluta si deve aggiornare l'indice della destinazione. Inoltre anche gli algoritmi *Greey-R* e *Shake* ne fanno uso, particolarmente per l'estrazione, ed eliminazione, di elementi appartenenti a *vector*.
Il codice della struttura dati *vector* è inserito nell'appendice A.

### 4.3- Le strutture dei *neighborhood*

#### 4.3.1- *1-opt*

L'implementazione di questa struttura *neighborhood* è abbastanza immediata in quanto richiede solo lo scambio di posizione tra due *clienti* consecutivi nel cammino. Lo è anche il calcolo della differenza dei costi tra la nuova soluzione e quella corrente, la cui formula è descritta nel capitolo precedente. La funzione si chiama *oneOpt* e prende in ingresso un array il cui *neighborhood* si va ad esplorare.

Ad ogni iterazione viene generata una soluzione appartenente al *neighborhood 1-opt* della soluzione passata come parametro in ingresso, se la differenza tra il costo di questa nuova soluzione e quello della soluzione di partenza è negativa, allora la nuova soluzione introduce un miglioramento e viene salvata in un array temporaneo. Alla fine viene data in uscita la soluzione la cui differenza di costo con la soluzione corrente genera il valore più piccolo.

#### 4.3.2- *2-opt*

Come già accennato, l'esplorazione del *neighborhood* 2-opt viene fatta in modo esaustivo sulla sua interezza e non è limitata alla visita di un determinato numero di *clienti* vicini, quindi in questa implementazione richiede un tempo $O(n^2)$. La funzione *twoOpt*, ogni volta che viene chiamata, inizializza le strutture dati $S_e$, $S_b$ e $\delta$ definiti nel capitolo precedente. Poi fa partire l'esplorazione del *neighborhood 2-opt*, della soluzione passata come parametro in ingresso, che sta in due cicli *for* innestati. Quest'esplorazione controlla se il nodo scelto per fare lo scambio *2-opt* produce una soluzione migliore, attraverso la verifica della somma $f(x) + f(x')$. Se il valore di questa somma è maggiore o uguale al doppio del valore di $f(x)$ allora la soluzione trovata non produce alcun miglioramento e viene scartata. Altrimenti, se questa somma è minore di $2*f(x)$, questa diventa la nuova soluzione corrente ed in entrambi i casi si passa all'iterazione successiva fino

all'esplorazione completa di tutto il *neighborhood*. Inoltre $\delta$ viene anche usata nell'esplorazione dei *neighborhood move-forward* e *move-backward*, e quindi deve essere calcolata anche prima dell'esplorazione di tali *neighborhood* per assicurarsi di fare i calcoli su valori aggiornati. Oltre al metodo che aggiorna le strutture dati $S_e$ e $S_b$, *updateSbSe*, descritto nell'*algoritmo 1* del capitolo precedente, sono state implementate delle funzioni per inizializzare queste strutture dati che vengono eseguite una volta sola quando viene lanciato il programma. Invece per quanto riguarda la funzione che calcola i valori $\delta(i)$, questa viene lanciata ogni volta che si trova una nuova soluzione ammissibile.
Nell'ottica di questa implementazione del *neighborhood 2-opt* non è stato necessario usare il metodo *updateSbSe*, descritto nell'*algoritmo 1* del terzo capitolo, in quanto le strutture $S_b$ e $S_e$ vengono sempre inizializzate prima di effettuare una ricerca locale, un altro motivo è che la soluzione restituita da questa ricerca locale potrebbe non essere la stessa quando si fa la successiva iterazione della medesima ricerca locale. Quindi si è pensato di omettere questa fase di aggiornamento e di conseguenza ottenere un risparmio di 1-2 secondi di tempo nel trovare la soluzione ottima.

### 4.3.3- *Move-forward* e *move-backward*

Data una soluzione come parametro in ingresso, si prende un blocco di *k clienti* consecutivi, con $k = 1, ..., l_{max}$, e lo si sposta a destra del circuito. L'esplorazione di questo *neighborhood* richiede un tempo $O(n^2)$. Mladenović et al. (2012) hanno attribuito a $l_{max}$ il valore 4, quindi si hanno quattro iterazioni in cui si spostano blocchi di 1, 2, 3 e 4 *clienti* consecutivi. Come nel caso del *1-opt* viene restituita la soluzione che genera una differenza, $\Delta f$, più piccola. Prima di iniziare la ricerca locale si calcola, per la soluzione passata in ingresso, la struttura dati $\delta$. Questo lo si fa ogni volta prima di iniziare la fase di ricerca.
Lo stesso ragionamento vale per il *neighorhood move-backward* con l'unica differenza è che i *clienti* vengono spostati a sinistra del circuito.

# CAPITOLO 5
# RISULTATI COMPUTAZIONALI

Tutti i test sono stati svolti su un processore Intel Core I5 con una frequenza di clock pari a 1.7 GHz e 8GB di RAM come sistema operativo Windows 8.1. Sono stati eseguiti dei test sia sul modello matematico che sugli algoritmi euristici. A questo scopo sono state generate delle istanze in modo casuale per testare il modello matematico, mentre le istanze usate per gli algoritmi euristici sono istanze appartenenti alla libreria TSPLib.

In Mladenović at al. (2012) i test sono stati eseguiti su un sistema Linux con un processore Intel Pentium IV con una frequenza di clock pari a 1.8GHz. Mentre in Silva et al. (2012) l'algoritmo è stato scritto nel linguaggio C++ (g++ 4.4.3) e i test eseguiti su un sistema GNU/Linux Ubuntu 10.04 (kernel 2.6.32-25) con un processore Intel Core i7 con una frequenza di clock pari a 2.93GHz e 8GB di RAM. I test sono stati eseguiti con un solo thread.

## 5.1- Modello matematico

Come già detto nel primo capitolo, il modello matematico definito dagli autori Mladenović at al. (2012) non considera l'attesa del ritorno al nodo deposito dopo avere visitato tutti i nodi *clienti*. È stato anche mostrato come si può modificare il modello per tenerne conto. Il modello matematico è stato implementato con il linguaggio C utilizzando la libreria *callable "cplex.h"*.

Sono state generate delle istanze con 10, 15, 20 e 25 nodi, cinque istanze per ciascuna tipologia. I valori degli elementi della matrice dei costi sono distribuiti uniformemente in un range che va da 1 a 100. Per tutte le istanze con 10 e 15 nodi e quattro delle istanze con 20 nodi sono state trovate le soluzioni ottime, mentre questo non si è verificato per una sola delle istanze con 20 nodi e tutte le istanze con 25 nodi. Nelle prossime due tabelle vedremo, per ciascuna tipologia, il valore medio

dei risultati dei test effettuati.

| Tabella 1: Risultati su istanze casuali | | | | |
|---|---|---|---|---|
| **Modello matematico senza ritorno** | | | | |
| | | | | |
| Istanze | Obj. Fun. | LB | Time (sec) | Gap (%) |
| r_10 | 554.4 | 554.4 | 0.42 | 0 |
| r_15 | 1042.6 | 1042.6 | 46.58 | 0 |
| r_20 | 1520 | 1391.14 | 4720.88 | 8.48 |
| r_25 | 2091.2 | 1006.67 | 7200 | 51.86 |

| Tabella 2: Risultati su istanze casuali e TSPlib | | | | |
|---|---|---|---|---|
| **Modello matematico con ritorno** | | | | |
| | | | | |
| Istanze | Obj. Fun. | LB | Time (sec) | Gap (%) |
| | 740.2 | 740.2 | 0.34 | 0 |
| r_15 | 1284.6 | 1284.6 | 18.01 | 0 |
| r_20 | 1816 | 1680.49 | 2462.33 | 7.46 |
| r_25 | 2318.8 | 1340.19 | 7200 | 42.2 |
| gr17 | 12994 | 10359.15 | 7200 | 20.28 |
| gr21 | 24600 | 14922.1 | 7200 | 39.34 |
| gr24 | 14372 | 7586.38 | 7200 | 47.21 |
| dantzig42 | 13338 | 5623.07 | 7200 | 57.84 |

Il termine “LB” indica il lower bound medio a fine calcolo o perché è stata trovata la soluzione ottima oppure perché si è raggiunto il limite di tempo concesso. Il “Gap” è differenza percentuale tra il valore medio della funzione obiettivo (“Obj Fun.”) e il “LB”.

I test sono stati effettuati con un limite massimo di due ore per ogni run. Ora guardando entrambe le tabelle notiamo che se si tiene conto del ritorno, CPLEX riesce a trovare la soluzione ottima in minor tempo rispetto al caso in cui non si tiene conto del ritorno al nodo *deposito*, oppure, nel caso in cui non riesce a trovare una soluzione ottima, esso trova una soluzione con un GAP minore. Da notare anche il GAP per le istanze con 20 nodi, per quattro di queste istanze CPLEX è riuscito a trovare la soluzione ottima, mentre per una non ci è riuscito il che spiega il valore del GAP.

Per quanto riguarda invece le istanze appartenenti alla libreria TSPlib, esse sono state testate usando il modello matematico che considera il ritorno al *deposito* per due motivi: 1) è stato osservato che quest’ultimo

modello dà dei risultati migliori; 2) sono noti i valori ottimi per queste istanze e quindi si riesce a misurare meglio le sue performance. Il modello matematico mostra una debolezza dal punto di vista del suo lower bound. Per esempio l'istanza "gr17" ha raggiunto il valore ottimo mentre il LB è lontano da quel valore, mentre l'istanza "gr21" ha un valore leggermente più alto rispetto a quello ottimo, invece il suo LB è ancora molto lontano. Nella tabella 2b è riportato inoltre il valore del Gap rispetto alla soluzione ottima sotto la colonna "Eff. Gap".

Tabella 2b: Risultati su TSPlib con Gap rispetto alla soluzione ottima

| **Modello matematico con ritorno** | | | | | |
|---|---|---|---|---|---|
| Istanze | Obj. Fun. | LB | Time (sec) | Gap (%) | Eff. Gap(%) |
| gr17 | 12994 | 10359.15 | 7200 | 20.28 | 0 |
| gr21 | 24600 | 14922.1 | 7200 | 39.34 | 1.05 |
| gr24 | 14372 | 7586.38 | 7200 | 47.21 | 4.18 |
| dantzig42 | 13338 | 5623.07 | 7200 | 57.84 | 6.47 |

### 5.2- GVNS

Per testare le implementazioni *GVNS-1* e *GVNS-2* ci si è serviti di ventisette istanze appartenenti alla libreria TSPlib di cui ventidue aventi numero di nodi inferiori a 107. Da qui in poi saranno riferite come istanze piccole, e cinque con numero di nodi superiore a 195, istanze grandi. I test sono stati effettuati con un limite di tempo massimo uguale a 600 secondi. In Mladenović et al. (2012) il *GVNS-1* ha dato dei risultati migliori rispetto al *GVNS-2* per le istanze piccole, mentre quest'ultimo ha dato dei risultati migliori per le istanze grandi. In questa implementazione il *GVNS-1* mostra risultati migliori rispetto al *GVNS-2* per tutte le istanze, in più mostra anche risultati migliori rispetto a quelli in Mladenović et al. (2012) in termini di tempo per le istanze piccole, ed in termini di UB per le istanze grandi tranne per una dove entrambi gli algoritmi implementati dagli autori hanno dato degli UB inferiori rispetto a quelli trovati in questa implementazione.

Inoltre per l'istanza "att532" il valore di UB trovato nei test è inferiore di un ordine di grandezza rispetto a quello trovato in Mladenović et al. (2012), che però è congruente con quello trovato in Silva et al. (2012) e quindi si presume che l'istanza è cambiata. Comunque si farà riferimento ai risultati trovati da quest'ultimi per quanto concerne quest'istanza.

La funzione della libreria "*math.h*" responsabile delle generazione di numeri casuali è stata inizializzata con una costante in modo da poter ottenere dei run replicabili. A questo punto si lavora sul parametro *k* della funzione *shake* per variare la casualità delle soluzioni ottenute, che sono sempre replicabili. Dopo alcuni test è stato scelto il valore *k=5* per le istanze piccole e *k=10* per le istanze grandi.

Le tabelle 3a e 3b riportano i risultati dei test condotti sulle istanze piccole della libreria TSPlib usando gli algoritmi *GVNS-1* e *GVNS-2* rispettivamente, mentre la tabella 4 mette a confronto i risultati dei test sulle istanze grandi trovati da parte di Mladenović et al. (2012), Silva et al. (2012) e di questa implementazione. *Time M* è il tempo impiegato da Mladenović et al. (2012) per trovare la soluzione.

Tabella 3a: Risultati GVNS-1

| **Algoritmo Sequential VND** | | | | | |
|---|---|---|---|---|---|
| Nome istanza | z* | UB trovato | Time (sec) | Time M(sec) | GAP |
| Dantzig42 | 12528 | 12528 | 0 | 0 | 0 |
| swiss42 | 22327 | 22327 | 0 | 0 | 0 |
| att48 | 209320 | 209320 | 0 | 0 | 0 |
| gr48 | 102378 | 102378 | 0 | 0 | 0 |
| hk48 | 247926 | 247926 | 0 | 0 | 0 |
| eil51 | 10178 | 10178 | 0 | 2 | 0 |
| berlin52 | 143721 | 143721 | 0 | 1 | 0 |
| brazil58 | 512361 | 512361 | 0 | 6 | 0 |
| st70 | 20557 | 20557 | 0 | 3 | 0 |
| eil76 | 17976 | 17976 | 0 | 1 | 0 |
| pr76 | 3455242 | 3455242 | 2 | 28 | 0 |
| gr96 | 2097170 | 2097170 | 0 | 2 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| rat99˚ | 57986 | 57986 | 17 | 24 | 0 |
| KroA100 | 983128 | 983128 | 0 | 52 | 0 |
| KroB100 | 986008 | 986008 | 0 | 30 | 0 |
| KroC100 | 961324 | 961324 | 0 | 2 | 0 |
| KroD100 | 976965 | 976965 | 0 | 7 | 0 |
| KroE100 | 971266 | 971266 | 0 | 3 | 0 |
| rd100 | 340047 | 340047 | 0 | 64 | 0 |
| eil101˚ | 27513 | 27513 | 7 | 62 | 0 |
| lin105 | 603910 | 603910 | 4 | 13 | 0 |
| pr107 | 2026626 | 2026626 | 1 | 4 | 0 |
| ˚ valore non ancora dimostrato di essere ottimo | | | | | |

*GVNS-1* riesce a trovare la soluzione ottima, il migliore UB noto nel caso si tratta delle istanze "rat99" e "eil101", in meno di 20 secondi.

Tabell 3b: Risultati GVNS-2

| Algoritmo Mixed VND | | | | |
|---|---|---|---|---|
| Nome istanza | z* | UB trovato | Time(sec) | GAP(%) |
| Dantzig42 | 12528 | 12528 | 0 | 0 |
| swiss42 | 22327 | 22327 | 0 | 0 |
| att48 | 209320 | 209320 | 1 | 0 |
| gr48 | 102378 | 102378 | 0 | 0 |
| hk48 | 247926 | 247926 | 0 | 0 |
| eil51 | 10178 | 10178 | 2 | 0 |
| berlin52 | 143721 | 143721 | 1 | 0 |
| brazil58 | 512361 | 512361 | 1 | 0 |
| st70 | 20557 | 20557 | 1 | 0 |
| eil76 | 17976 | 17976 | 71 | 0 |
| pr76 | 3455242 | 3455242 | 9 | 0 |
| gr96 | 2097170 | 2097170 | 4 | 0 |
| rat99˚ | 57986 | 57986 | 66 | 0 |
| KroA100 | 983128 | 983128 | 9 | 0 |
| KroB100 | 986008 | 986008 | 15 | 0 |
| KroC100 | 961324 | 961324 | 9 | 0 |
| KroD100 | 976965 | 976965 | 11 | 0 |
| KroE100 | 971266 | 971266 | 14 | 0 |
| rd100 | 340047 | 340047 | 6 | 0 |
| eil101˚ | 27513 | 27636 | 9 | 0.45 |
| lin105 | 603910 | 603910 | 23 | 0 |
| pr107 | 2026626 | 2029412 | 14 | 0.14 |
| ˚ valore non ancora dimostrato di essere ottimo | | | | |

I tempi in cui *GVNS-2* riesce a trovare la soluzione ottima, oppure il

migliore UB, sono maggiori rispetto a quelli del *GVNS-1* nonché per due istanze il risultato trovato è maggiore rispetto al migliore conosciuto ma con un gap inferiore al 0.5%.

Le tabelle 4a e 4b mettono a confronto i risultati ottenuti con le implementazioni oggetto di questa tesi degli algoritmi *seqVND* e *mixVND*, le implementazioni degli stessi algoritmi da parte di Mladenović et al. (2012), colonna *UB M*, e *GILS-RVND* di Silva et al. (2012), *UB S*. Analogamente *Gap M* indica la differenza percentuale tra i risultati trovati in questa implementazione e quelli trovati da parte di Mladenović et al. (2012), e *Gap S* per i risultati trovati da Silva et al. (2012). Infine *Time S* è il tempo impiegato da Silva et al. (2012).

Tabella 4a: Confronto tra GVNS-1, GVNS-1 di Mladenović e GILS-RVND

| Nome istanza | UB M | UB S | UB trovato | Time (sec) | Time M(sec) | Time S(sec) | GAP M | GAP S |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| rat195 | 210193 | 210191 | 216154 | 65 | 111.2 | 75.56 | 2.84 | 2.84 |
| pr226 | 7106546 | 7100308 | 7101223 | 3 | 144.29 | 59.05 | -0.08 | 0.01 |
| lin318 | 5612165 | 5560679 | 5594752 | 29 | 179.13 | 220.59 | -0.31 | 0.61 |
| pr439 | 17790562 | 17688561 | 17702851 | 253 | 327.15 | 553.74 | -0.49 | 0.08 |
| att532 | 17610285 | 5581240 | 5597225 | 558 | 488.95 | 1792.61 | - | 0.29 |

Il risultato trovato con *GVNS-1* sull'istanza "rat195" è maggiore di quello trovato dalla sua controparte di Mladenović et al. (2012), e da *GILS-RVND*, mentre man mano che aumenta il numero di nodi *GVNS-1* comincia a trovare soluzioni migliori rispetto a quelle trovate da Mladenović et al. (2012), e poco peggiori rispetto a quelle trovate con *GILS-RVND*. La prossima tabella mostra il risultato di *GVNS-2*.

Tabella 4b: Confronto tra GVNS-2, GVNS-2 di Mladenović e GILS-RVND

| Nome istanza | UB M | UB S | UB trovato | Time (sec) | Time M(sec) | Time S(sec) | GAP M | GAP S |
|---|---|---|---|---|---|---|---|---|
| rat195 | 210319 | 210191 | 217228 | 600 | 150.94 | 75.56 | 3.29 | 3.35 |
| pr226 | 7100708 | 7100308 | 7101223 | 556 | 37.57 | 59.05 | 0.01 | 0.01 |
| lin318 | 5601351 | 5560679 | 5571045 | 600 | 294.85 | 220.59 | -0.54 | 0.19 |
| pr439 | 17799975 | 17688561 | 18349897 | 600 | 396.84 | 553.74 | 3.09 | 3.74 |
| att532 | 17584979 | 5581240 | 5808527 | 600 | 532.02 | 1792.61 | - | 4.07 |

Da notare che malgrado i risultati di *GVNS-2* siano peggiori rispetto a tutte le altre implementazioni, esso è riuscito a raggiungere un UB migliore rispetto a quello trovato da *GVNS-1* per l'istanza "lin318" andando molto vicino a quello trovato da *GILS-RVND*.

# CAPITOLO 6
# CONCLUSIONI

In conclusion il modello matematico è debole, e ha difficoltà nella risoluzione di istanze con 20 o più nodi. D'altro canto l'algoritmo euristico ha dimostrato di poter risolvere fino all'ottimalità tutte le istanze in letteratura con un numero di nodi che arriva fino a 107. Se si guardano i risultati prodotti da *GVNS-1* si può notare che esso ha migliorato in termini di tempo richiesto per trovare la soluzione ottima entrambe le implementazioni fatte da Mladenović et al. (2012) per le istanze piccole e grandi, fatta eccezione l'istanza "rat195" dove *GVNS-1* ha prodotto un risultato peggiore e l'istanza "att532" dove non era possibile fare un confronto.

Inoltre facendo un confronto tra *GVNS-1* e *GILS-RVND* quest'ultimo ha dato risultati leggermente migliori rispetto al primo con un GAP inferiore all'1%, eccetto per l'istanza "rat195" dove il GAP era uguale al 2.84%.

*GVNS-2* ha dato dei risultati peggiori rispetto a tutti gli altri algoritmi tranne nel caso dell'istanza "lin318" dove è riuscito a migliorare il risultato di *GVNS-1* ma è rimasto leggermente più alto di quello dato dall'algoritmo *GILS-RVND*.

Infine il codice di *GVNS-1* e *GVNS-2* può essere ottimizzato per dare dei risultati migliori, specialmente per quanto riguarda le istanze grandi. Nel caso del *GVNS-2* un modo sarebbe di cambiare *neighborhood* solo quando troviamo la migliore soluzione rispetto a quella corrente invece di farlo quando si ottiene la prima soluzione migliore di quella corrente. Oppure cambiare l'implementazione per renderla identica a quella suggerita dagli autori Mladenović et al. (2012).

# REFERENZE

- Abeledo H, Fukusawa R, Pessoa A, Uchoa E (2012) The time dependent traveling salesman problem: polyhedra and algorithm.
- Mladenović N, Urošević D, Hanafi S (2012) Variable neighbourhood search for the Travelling Deliveryman Problem.
- Roberti R, Mingozzi A (2013) Dynamic ng-Path Relaxation for the Delivery Man Problem.
- Salehipour A, Sörensen K, Goos P, Bräysy O (2011) Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem.
- Silva M-M, Subramanian A, Thibaut V, Ochi L-S (2012) A simple and effective metaheuristic for the Minimum Latency Problem.

# RINGRAZIAMENTI

Vorrei inanzitutto ringraziare di cuore il professor Paolo Toth ed il Dott. Roberto Roberti per i numerosi e costanti aiuti durante la stesura della tesi, senza di loro questa non sarebbe stata possibile.

In secondo luogo vorrei ringrazziare la mia famiglia per la loro pazienza e per essermi stati sempre vicino.

Infine grazie a tutti i miei amici per gli incoraggiamenti datimi durante tutto questo periodo, e per essermi stati vicino a festeggiare gli ultimi esami. In particolare grazie di cuore a Elisa, Emmanuele, Giovanni, Giuseppe, Hassan, Massimiliano e Zeletina.

# APPENDICE

# CODICE IMPLEMENTATIVO

Di seguito è riportato il codice che implementa il modello matematico usando la libreria “cplex.h”.

tdp.cpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <cplex.h>
#include <windows.h>

#define N 200
//#define NR N^2+4*N+3
//#define NC 2*N*(N+1)^2

// Global variables - input data
int n, tipo;
int d[N+1][N+1];

int checkStatus (CPXENVptr env, int status){
	char errmsg[1024];
	if ( status == 0 ) return 0;
	CPXgeterrorstring(env, status, errmsg);
	printf(" %s \n", errmsg);
	return status;
}

void readInput(char* filename){
	FILE* inp, *out;
	errno_t err = 0;

	if( ( err = fopen_s(&inp, filename, "r") ) != 0 ) {
		printf("Could not open file! \n");
		return;
	}

	//legge il tipo dell'input, qui si da per
	//scontato che è una matrice
	//quindi si legge il tipo e lo si butta via
	fscanf_s(inp, "%i", &tipo);

	//legge il numero di nodi n
	fscanf_s (inp, "%i", &n);
	if( n > N ) {
		printf("N is too small!");
		fclose(inp);
		return;
	}
	--n;
```

```
	// ricostruisce la matrice delle distanze a
	//partire dala triangolare
	// superiore nel file di input
	for (int j = 0; j <= n ; j++){
		for(int i = 0; i < j; i++){
			fscanf_s(inp, "%i", &d[i][j]);
			d[j][i] = d[i][j];
		}
		fscanf_s(inp,"%i",&d[j][j]);
	}

	if( ( err = fopen_s(&out, "matrix2.txt", "w") ) !=
	0 ) {
		printf("Could not open file! \n");
		return;
	}
	for( int i = 0 ; i <= n ; i++){
		for(int j = 0 ; j <= n ; j++){
			fprintf_s(out,"%4i ",d[i][j]);
		}
		fprintf(out,"\n");
	}
	fclose(out);
	fclose(inp);
}

void resolveProblem(char* filename){
	FILE * out;
	errno_t err = 0;

	// cplex variables infeasout[NC],
	CPXENVptr env;
	CPXLPptr lp;
	double obj[1], rhs[1], matval[N+1], lb[1],ub[1],
	objval, bestobjval, gap,  begin, end;
	int status, matind[N+1], matbeg[1], nzcnt;
	char sense[1], xctype[1];

	// Solution variables
	double x[2*N*(N+1)];

	// Local variables
	int ind_x[N+1][N+1], ind_y[N+1][N+1], nvar;
	int i, j;

	// lettura dell'input e costruzione della matrice
	//delle distanze
	readInput(filename);

	// apertura ambiente cplex e creazione
	// problema
	env = CPXopenCPLEX(&status);
	if( checkStatus(env, status) ) return;

	lp = CPXcreateprob(env, &status, "TDP");
	if ( checkStatus(env, status) ) return;

	status = CPXsetintparam(env, CPX_PARAM_SCRIND,
	CPX_ON);
```

```
if (checkStatus(env, status)) goto Z;

CPXchgobjsen(env, lp, CPX_MIN);

// setto alcune variabili di ambiente
status = CPXsetdblparam(env, CPX_PARAM_TILIM,
7200.0);
if ( checkStatus(env, status) ) goto Z;

status = CPXsetintparam(env, CPX_PARAM_CLOCKTYPE,
2);
if ( checkStatus(env, status) ) goto Z;

status = CPXsetintparam(env, CPX_PARAM_FLOWCOVERS,
2);
if ( checkStatus(env, status) ) goto Z;

status = CPXsetintparam(env, CPX_PARAM_IMPLBD, 2);
if ( checkStatus(env, status) ) goto Z;

status = CPXsetintparam(env, CPX_PARAM_MIRCUTS, 2);
if ( checkStatus(env, status) ) goto Z;

status = CPXsetintparam(env, C
PX_PARAM_ZEROHALFCUTS, 2);
if ( checkStatus(env, status) ) goto Z;



// aggiunge le variabili y_ij. Le variabili //con
//indici i e j uguali non sono
// state aggiunte
nvar = 0;
xctype[0] = 'I';
lb[0] = 0.0;
ub[0] = n;
for(i = 0 ; i <= n ; i++){
      for( j = 0 ; j <= n ; j++){
            if(i == j) continue;
            obj[0] = d[i][j];
            status = CPXnewcols(env, lp, 1, obj,
            lb, ub, xctype, NULL);
            if(checkStatus(env, status)) goto Z;
            ind_y[i][j] = nvar;
            nvar ++;
      }
}
// aggiunge le variabili x_ij. Esse non
//compaiono nella funzione obiettivo
//quindi avranno coefficiente = 0, cioe //obj[0]=0

xctype[0] = 'B';
obj[0] = 0.0;
lb[0] = 0.0;
ub[0] = 1.0;
for ( i = 0 ; i <= n ; i++ ){
      for ( j = 0 ; j <= n ; j++ ){
            if(i == j) continue;
```

```
                  obj[0] = d[i][j]; // commentare
//questa riga per non considerare il ritorno al
//deposito
            status = CPXnewcols(env, lp, 1, obj,
            lb, ub, xctype, NULL);
            if( checkStatus(env, status) ) goto Z;
            ind_x[i][j] = nvar;
            nvar++;
      }
}

matbeg[0] = 0;
// aggiunge i vincoli ∑ x_ij = 1 per i = 0, ..., n
//&& i != j
rhs[0] = 1.0;
sense[0] = 'E';
for ( i = 0 ; i <= n ; i++){
      nzcnt = 0;
      for( j = 0 ; j <= n ; j++){
            if ( j == i ) continue;
            matind[nzcnt] = ind_x[i][j];
            matval[nzcnt] = 1.0;
            nzcnt ++;
      }
      status = CPXaddrows(env, lp, 0, 1, nzcnt,
      rhs, sense, matbeg, matind, matval, NULL,
      NULL);
      if ( checkStatus(env, status) ) goto Z;
}

// aggiunge i vincoli ∑ x_ij = 1 per j = 0, ..., n
//&& i != j
rhs[0] = 1.0;
sense[0] = 'E';
for ( j = 0 ; j <= n ; j++){
      nzcnt = 0 ;
      for( i = 0 ; i <= n ; i ++){
            if( i == j ) continue;
            matind[nzcnt] = ind_x[i][j];
            matval[nzcnt] = 1.0;
            nzcnt ++;
      }
      status = CPXaddrows(env, lp, 0, 1, nzcnt,
      rhs, sense, matbeg, matind, matval, NULL,
      NULL);
      if ( checkStatus(env, status) ) goto Z;
}

// aggiunge il vincolo ∑ y_0j = n per i = 1, ..., n
rhs[0] = n;
sense[0] = 'E';
nzcnt = 0;
for ( j = 1 ; j <= n ; j++ ){
      matind[nzcnt] = ind_y[0][j];
      matval[nzcnt] = 1.0;
      nzcnt++;
}
status = CPXaddrows(env, lp, 0, 1, nzcnt, rhs,
sense, matbeg, matind, matval, NULL, NULL);
```

```
if ( checkStatus(env, status) ) goto Z;

// aggiunge i vincoli ∑ y_ik - ∑ y_kj = 1 per k =
//1, ..., n && i,j = 0, ... , n
rhs[0] = 1.0;
sense[0] ='E';
for ( int k = 1 ; k <= n ; k++ ){
      nzcnt = 0;
      for ( i = 0 ; i <= n ; i++){
            if ( i == k ) continue;
            matind[nzcnt] = ind_y[i][k];
            matval[nzcnt] = 1.0;
            nzcnt++;
      }
      for ( j = 0 ; j <= n ; j++ ){
            if ( j == k ) continue;
            matind[nzcnt] = ind_y[k][j];
            matval[nzcnt] = -1.0;
            nzcnt++;
      }
      status = CPXaddrows(env, lp, 0, 1, nzcnt,
      rhs, sense, matbeg, matind, matval, NULL,
      NULL);
      if ( checkStatus(env, status) ) goto Z;
}

//aggiunge i vincoli y_i0 = 0 per i = 1, ..., n
rhs[0] = 0;
sense[0] = 'E';
// qui non si usa nzcnt perché ciascun vincolo ha
//un solo elemento
for ( i = 1 ; i <= n ; i++ ){
      matind[0] = ind_y[i][0];
      matval[0] = 1.0;
      status = CPXaddrows(env, lp, 0, 1, 1, rhs,
      sense, matbeg, matind, matval, NULL, NULL);
      if ( checkStatus(env, status) ) goto Z;
}

// aggiunge i vincoli y_0j <= n * x_0j per j = 1,
//..., n
sense[0] = 'L';//
rhs[0] = 0;
// qui non si usa nzcnt perché ciascun vincolo ha
//solo due elementi
for ( j = 1 ; j <= n ; j ++ ){
      matind[0] = ind_y[0][j];
      matval[0] = 1.0;
      matind[1] = ind_x[0][j];
      matval[1] = -n;
      status = CPXaddrows(env, lp, 0, 1, 2, rhs,
      sense, matbeg, matind, matval, NULL, NULL);
      if( checkStatus ( env, status ) ) goto Z;
}

// aggiunge i vincoli y_ij <= (n - 1) * x_ij per i,
//j = 1, ..., n && i != j
rhs[0] = 0;
sense[0] = 'L';
```

```
        // qui non si usa nzcnt perché ciascun vincolo ha
        //solo due elementi
        for ( i = 1 ; i <= n ; i++ ){
              for( j = 1 ; j <= n ; j++ ){
                    if ( i == j ) continue;
                    matind[0] = ind_y[i][j];
                    matval[0] = 1.0;
                    matind[1] = ind_x[i][j];
                    matval[1] = (1 - n);
                    status = CPXaddrows(env, lp, 0, 1, 2,
                    rhs, sense, matbeg, matind, matval,
                    NULL, NULL);
                    if ( checkStatus(env, status) ) goto Z;
              }
        }

        status = CPXwriteprob(env, lp, "TDP.lp", NULL);
        if ( checkStatus(env, status) ) goto Z;

        status = CPXgettime(env, &begin);
        if( checkStatus(env, status) ) goto Z;

        status = CPXmipopt(env, lp);
        if( checkStatus(env, status) ) goto Z;

        status = CPXgettime(env, &end);
        if( checkStatus(env, status) ) goto Z;

        status = CPXgetobjval(env, lp, &objval);
        if ( checkStatus(env, status) ) goto Z;
        printf("\n Optimal solution value %5.0f
        \n\n",objval);

        status = CPXgetbestobjval(env, lp, &bestobjval);
        if ( checkStatus(env, status) ) goto Z;

        status = CPXgetmiprelgap(env, lp, &gap);
        if ( checkStatus(env, status) ) goto Z;

        err = fopen_s(&out, "output.txt", "a");
        if( err != 0) printf("could not create output
        file!");

        status = CPXgetx(env, lp, x, 0, CPXgetnumcols(env,
        lp) - 1);
        if ( checkStatus(env, status) ) goto Z;

        fprintf_s(out, "%s: \n", filename);
        fprintf_s(out, "%s = %f \t", "Best Integer", ob-
        jval);
        fprintf_s(out, "%s = %f \t", "Best Bound",
        bestobjval);
        fprintf_s(out, "%s = %f \t", "Gap", gap);
        fprintf_s(out, "%s = %f \n\n", "Time", end -
        begin);
        fclose(out);

Z:      status = CPXfreeprob(env, &lp);
        if( checkStatus(env, status) ) return;
```

```
      status = CPXcloseCPLEX(&env);
      if ( checkStatus(env, status) ) return;
}
/*


*/
void main ( void ){
      char *sDir = "instances";

      WIN32_FIND_DATA fdFile;
    HANDLE hFind = NULL;

    char sPath[2048];

    //Specify a file mask. *.* = We want everything!
    sprintf(sPath, "%s\\att532.*", sDir);

    if((hFind = FindFirstFile(sPath, &fdFile)) == INVA-
    LID_HANDLE_VALUE)
    {
        printf("Path not found: [%s]\n", sDir);
        return;
    }

    do
    {
        //Find first file will always return "."
        //    and ".." as the first two directories.
        if(strcmp(fdFile.cFileName, ".") != 0
                && strcmp(fdFile.cFileName, "..") != 0)
        {
            //Build up our file path using the passed in
            //[sDir] and the file/foldername we just
            //found:
            sprintf(sPath, "%s\\%s", sDir,
            fdFile.cFileName);

            resolveProblem(sPath);
        }
    }
      //Find the next file.
    while(FindNextFile(hFind, &fdFile));

    FindClose(hFind);

    return;
}
```

Il codice implementativo della struttura dati ausiliaria “vector.cpp” insieme al suo file header “vector.h” è il seguente:

vector.h:

```
#ifndef VECTOR_H__
#define VECTOR_H__

typedef struct vector_ {
	void** data;
	int size;
	int count;
} vector;

void vector_init(vector*);
int vector_count(vector*);
void vector_add(vector*, void*);
void vector_set(vector*, int, void*);
void *vector_get(vector*, int);
void vector_delete(vector*, int);
void vector_insert(vector*, int, void*);
void vector_swap(vector*, int, int);
void vector_invertSegment(vector*, int, int);
void vector_moveSegment(vector*, int, int, int);
void* vector_extract(vector* , int);
int vector_getIndex(vector *, void *);
int vector_contains(vector*, void *);
void vector_free(vector*);

#endif
```

vector.cpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

#include "vector.h"

void vector_init(vector *v){
	v->data = NULL;
	v->size = 0;
	v->count = 0;
}

int vector_count(vector *v)
{
	return v->count;
}

void vector_add(vector *v, void *e)
{
	if (v->size == 0) {
		v->size = 10;
		v->data = (void**) malloc(sizeof(void*) * v-
		>size);
```

```
            memset(v->data, '\0', sizeof(void *) * v-
            >size);
      }

      // condition to increase v->data:
      // last slot exhausted
      if (v->size == v->count) {
            v->size *= 2;
            v->data = (void **)realloc(v->data,
            sizeof(void*) * v->size);
      }

      v->data[v->count] = e;
      v->count++;
}

void vector_set(vector *v, int index, void *e)
{
      if (index >= v->count) {
            return;
      }

      v->data[index] = e;
}

void *vector_get(vector *v, int index)
{
      if (index >= v->count) {
            return NULL;
      }

      return v->data[index];
}

void vector_delete(vector *v, int index)
{
      if (index >= v->count) {
            return;
      }

      int i;
      for( i = index ; i < v->count - 1 ; i++){
            v->data[i] = v->data[i+1];
      }
      v->data[v->count - 1] = NULL;
      v->count--;
}

void vector_insert(vector *v, int index, void* e)
{
      if(index > v->count){
            return;
      }

      if (v->size == 0) {
            v->size = 10;
            v->data = (void**) malloc(sizeof(void*) * v-
            >size);
```

```
            memset(v->data, '\0', sizeof(void *) * v-
            >size);
      }

      // condition to increase v->data:
      // last slot exhausted
      if (v->size == v->count) {
            v->size *= 2;
            v->data = (void **)realloc(v->data,
            sizeof(void*) * v->size);
      }

      int i, j;
      void **newarr = (void**)calloc(v->count * 2,
      sizeof(void*));
      for (i = 0, j = 0; i <= v->count; i++) {
            if(j == index) {
                  newarr[j] = e;
                  j++;
            }
            newarr[j] = v->data[i];
            j++;
      }

      free(v->data);

      v->data = newarr;
      v->count++;
}

void vector_swap(vector* v, int begin, int end)
{
      if(begin >= v->count || end >= v->count){
            return;
      }

      void* tmp = v->data[begin];
      v->data[begin] = v->data[end];
      v->data[end] = tmp;
}

void vector_invertSegment(vector* v, int begin, int end)
{
      if(begin >= v->count || end >= v->count){
            return;
      }

      void * tmp;
      int length = end - begin + 1;
      int i;
      for(i = 0 ; i < length/2 ; i++){
            tmp = v->data[begin + i];
            v->data[begin + i] = v->data[end - i];
            v->data[end - i] = tmp;
      }
}

void vector_moveSegment(vector * v, int start, int dest,
int length)
```

```
{
	if((start >= v->count && dest>= start)|| dest > v-
	>count || (start + length) > v->count){
		return;
	}

	void **newarr = (void**)calloc(v->count *2,
	sizeof(void*) );
	int i;
	for(i = 0 ; i < length ; i++){
		newarr[i] = v->data[start + i];
	}

	for(i = 0 ; i < length ; i++){
		// eliminates an element from the array.
		// consecutive elements in the original array
		//have the same index
		// in the updated array.
		vector_delete(v,start);
	}

	// we need to update the 'dest' index so that it
	points to the same element
	// in the updated array. To do that we need to dec-
	rement it by 'length'.
	// this only applies if we are a segment to the
	left.
	if(dest > start) dest -= length;

	// now we insert the extracted elements to the up-
	dated array
	for(i = 0 ; i < length ; i++){
		vector_insert(v, dest + i, newarr[i]);
	}

	free(newarr);
}

void* vector_extract(vector* v, int index)
{
	if(index >= v->count) {
		return NULL;
	}
	void* e = v->data[index];
	vector_delete(v, index);
	return e;
}

int vector_getIndex(vector* v, void * e)
{
	int i;
	for(i = 0 ; i < v->count ; i++){
		if(e == v->data[i]){
			return i;
		}
	}
	return -1;
}
```

```
int vector_contains(vector* v, void * e)
{
	int i;
	for(i = 0 ; i < v->count ; i++){
		if(e == v->data[i]){
			return 0;
		}
	}
	return -1;
}

void vector_free(vector *v)
{
	free(v->data);
}
```

Infine ecco il codice dell'implementazione degli algoritmi euristici *GVNS-1* e *GVNS-2*, "gvns.cpp":

```
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"
#include <time.h>
#include <memory.h>
#include <assert.h>
#include <math.h>
#include <Windows.h>

#define N 1000

int tipo, n, timmax;
int d[N+1][N+1], r[N+1][N+1], sb[N+1], se[N+1],
delta[N+1];
int temp[N+1], t[N], m, i, j, initialCost, x[N+1],
y[N+1];
double rij, dij;
time_t start, stop;

void readInput(char* filename){
	FILE* inp;
	errno_t err = 0;
	double xd, yd;//, rij, dij;
	float xx[N+1], yy[N+1];
	int tmpij;

	if( ( err = fopen_s(&inp, filename, "r") ) != 0 ) {
		printf("Could not open file! \n");
		return;
	}

	//legge il tipo dell'input, qui si da per scontato
	//che è una matrice
	//quindi si legge il tipo e lo si butta via
	fscanf_s(inp, "%i", &tipo);
```

```
	//legge il numero di nodi n
	fscanf_s (inp, "%i", &n);
	if( n > N ) {
		printf("N is too small!");
		fclose(inp);
		return;
	}
	--n; // mi da il numero effettivo dei clienti

	// ricostruisce la matrice delle distanze a partire
	//dall'input

	if(tipo == 1){		// Euclidean problem
		for(i = 0 ; i <= n ; i++){
			fscanf_s(inp, "%f", &xx[i]);
			fscanf_s(inp, "%f", &yy[i]);
		}
		for(i = 0 ; i <= n ; i++){
			for(j = 0 ; j <= n ; j++){
				xd = xx[i] - xx[j];
				yd = yy[i] - yy[j];
				dij = int (sqrt( xd*xd + yd*yd) +
0.5);
				d[i][j] = dij;
			}
		}
	}
	else if(tipo == 2){	// lower matrix problem
		for (j = 0; j <= n ; j++){
			for(i = 0; i < j; i++){
				fscanf_s(inp, "%i", &d[i][j]);
				d[j][i] = d[i][j];
			}
			fscanf_s(inp,"%i",&d[j][j]);
		}
	}
	else if(tipo == 3){//Att or Pseudo euclidean problem
		for(i = 0 ; i <= n ; i ++){
			fscanf_s(inp, "%i", &x[i]);
			fscanf_s(inp, "%i", &y[i]);
		}
		for(i = 0 ; i <= n ; i++){
			for(j = 0 ; j <= n ; j++){
				xd = x[i] - x[j];
				yd = y[i] - y[j];
				rij = sqrt( (xd*xd + yd*yd)/10);
				dij = int(rij + 0.5);
				if(dij < rij) dij++;
				d[i][j] = dij;
			}
		}
	}
	else if(tipo == 4){	// Full matrix problem
		for(i = 0 ; i <= n ; i++){
			for(j = 0 ; j <= n ; j++){
				fscanf_s(inp, "%i", &d[i][j]);
			}
		}
	}
```

```
else if(tipo == 5){           // Upper row problem
      for(i = 0 ; i <= n ; i ++){
            for(j = 0 ; j <= n ; j ++){
                  d[i][j] = 0;
            }
      }

      for(i = 0 ; i < n ; i ++){
            for(j = i + 1 ; j <= n ; j ++){
                  fscanf_s(inp, "%i", &tmpij);
                  d[i][j] = d[j][i] = tmpij;
            }
      }
}
else if(tipo == 6){           // Geo problem

      double latitude[N+1], longitude[N+1];
      float dx[N+1], dy[N+1];
      double const PI = 3.141592, RRR = 6378.388;
      double deg, mmin, q1, q2, q3;

      for( i =  0; i <= n ; i ++){
            fscanf_s(inp, "%f", &dx[i]);
            fscanf_s(inp, "%f", &dy[i]);

            deg = (int) dx[i];
            mmin = dx[i] - deg;
            latitude[i] = PI * (deg + 5.0 * mmin/
            3.0) / 180.0;

            deg  = (int) dy[i];
            mmin = dy[i] - deg;
            longitude[i] = PI * (deg + 5.0 * mmin/
            3.0) / 180.0;
      }

      for(i = 0 ; i <= n ; i ++){
            for(j = 0 ; j <= n ; j++){
                  if( i == j ){
                        d[i][j] = 0;
                        continue;
                  }
                  q1 = cos(longitude[i] - longi-
                  tude[j]);
                  q2 = cos(latitude[i] - lati-
                  tude[j]);
                  q3 = cos(latitude[i] + lati-
                  tude[j]);

                  d[i][j] = int(RRR * acos( 0.5 * (
                  (1.0 + q1) * q2 - (1.0 - q1) * q3
                  ) ) + 1.0);
            }
      }
}

// faccio la rank sulle distanze per ciascun nodo
for(int i = 0 ; i <= n ; i++){
      int o = 0;
```

```
            for(int l = 0 ; l <= n ; l++){
                  if(l == i ) continue;
                  temp[o] = d[i][l];
                  r[i][o] = l;
                  o++;
            }

            for(int j = 0 ; j < n ; j++){
                  int q = j;
                  for(int k = j + 1; k < n ; k++){
                        if(temp[j] > temp[k]){
                              m = temp[j];
                              temp[j] = temp[k];
                              temp[k] = m;
                              q = r[i][k];
                              r[i][k] = r[i][j];
                              r[i][j] = q;
                        }
                  }
            }
      }
      fclose(inp);
}

void GreedyR(int arr[], int q){

      int k = 1, length = n, tlen = 0, random, cr, min;
      vector s;

      srand(1);
      vector_init(&s);

      for(i = 1 ; i <= n ; i++){
            vector_add(&s, (int *) i);
      }
      arr[0] = 0;

      while(length > 0){
            min = q;
            if(q > n - k){
                  random = rand() % vector_count(&s);
                  arr[k] = (int)vector_extract(&s, ran-
                  dom);
                  goto A;
            }
            tlen = 0;

            for( i = 0, j = 0 ; i < n && j < min ; i++){
                  cr = arr[k-1];
                  if(vector_contains(&s, (int *)
                  r[cr][i]) == 0 &&  r[cr][i] != 0){
                        t[j] = r[cr][i];
                        tlen++;
                        j++;
                  }
            }

            random = rand() % tlen;
```

```
            arr[k] = t[random];

            vector_delete(&s, vector_getIndex(&s, (int *)
            arr[k]));
A:          k++;
            length --;
      }
      vector_free(&s);
}

void shake(int arr[], int k){

      vector a;

      vector_init(&a);

      for(i = 1 ; i <= n ; i++){
            vector_add(&a, (int *) arr[i]);
      }

      while(k > 0){
            i = rand() % n ;
            j = rand() % n  ;

            if(i != j){
                  int tmp = (int)vector_extract(&a, i);
                  if(i < j) {
                        vector_insert(&a, j-1, (int
                        *)tmp);
                  }
                  else{
                        vector_insert(&a, j , (int *)
                        tmp);
                  }
                  k--;
            }
      }

      for(i = 1 ; i <= n ; i++){
            arr[i] = (int)vector_get(&a, i - 1);
      }

      vector_free(&a);
}

int calculateCost(int arr[]){

      int cost = 0;
      int mod = n+1;
      for(i = 0 ; i <= n ; i ++){
            cost += (n + 1 - i) *
            (d[arr[i]][arr[(i+1)%mod]]);
      }
      return cost;
}

void calculateSb(int arr[])
{
```

```
	int i,j, tmp = 0;

	memset(sb, 0, sizeof(int) * (N + 1));
	int mod = n + 1;
	for(i = 0 ; i <= n ; i++){
		tmp = 0;
		for(j = 0 ; j <= i ; j++){
			tmp += (n + 1 - j) *
			(d[arr[j]][arr[(j+1)%mod]]);
		}
		sb[i] = tmp;
	}
}

void calculateSe(int arr[])
{
	int i,j, tmp = 0;

	memset(se, 0, sizeof(int) * (N + 1));
	int mod = n + 1;
	for(i = 0 ; i <= n ; i++){
		tmp = 0;
		for(j = i ; j <= n ; j++){
			tmp += (n + 1 - j) *
			(d[arr[j]][arr[(j+1)%mod]]);
		}
		se[i] = tmp;
	}
	se[n+1] = 0;
}

void calculateDelta(int arr[])
{
	int i, j, tmp = 0;

	memset(delta, 0, sizeof(int) * (N + 1));
	int mod = n + 1;
	for(i = 0 ; i <= n ; i++){
		tmp =  0;
		for(j = 0 ; j <= i ; j++){
			tmp += d[arr[j]][arr[(j+1)%mod]];
		}
		delta[i] = tmp;
	}
}

void updateSbSe(int arr[], int indx1, int indx2)
{
	int i, j;
	int mod = n + 1;
	for(i = indx1 - 1 ; i <= n ; i++){
		sb[i] = sb[i - 1] + (n + 1 - i) *
		(d[arr[i]][arr[(i+1)%mod]]);
	}

	for(j = indx2 ; j >= 0 ; j--){
		se[j] = se[(j + 1)] + (n + 1 - j) *
		(d[arr[j]][arr[(j + 1)%mod]]);
	}
```

```
}

void oneOpt(int arr[])
{
	int i, j, tmp1, tmp2, min;
	int *tmparr;

	min = 0;
	tmparr = (int *) calloc(N+1, sizeof(int));

	for(j = 0 ; j <= n ; j++){
			tmparr[j] = arr[j];
		}

	for(i = 0 ; i <= n - 2; i++){

		int mod = n+1;
		tmp2 = (n+1 - i) * (d[arr[i]][arr[i+2]] -
		d[arr[i]][arr[i+1]]) + (n+1 - i - 2) *
		(d[arr[i+1]][arr[(i+3)%mod]] -
		d[arr[i+2]][arr[(i+3)%mod]]);

		if(tmp2 < min){
			min = tmp2;
			for(j = 0 ; j <= n ; j++){
				tmparr[j] = arr[j];
			}
			tmp1 = arr[i + 1];
			tmparr[i + 1] = arr[i + 2];
			tmparr[i + 2] = tmp1;
		}
	}

	for( j = 0 ; j <= n ; j++){
		arr[j] = tmparr[j];
	}

	free(tmparr);
}

void twoOpt(int arr[], int lmax)
{
	int i, j, k, begin, end, tmp1, tmp2, length,
	arrcost, min;
	int a, b;
	int *tmparr;

	min = calculateCost(arr);
	arrcost = min;
	a = 2;
	b = n-1;

	tmparr = (int *) malloc(sizeof(int) * (N+1));

	calculateSb(arr);
	calculateSe(arr);
	calculateDelta(arr);

	for(j = 0 ; j <= n ; j++){
```

```
                tmparr[j] = arr[j];
        }

    for(begin = 1 ; begin < n - 1 ; begin++){
        for(end = begin + 1 ; end <= n ; end++){
            b = end;
            int mod = n+1;

            tmp2 = 2 * sb[begin - 1] +
                2 * se[end + 1] +
                (2 * (n+1) - begin - end) *
                ( delta[end - 1] - delta[begin])+
                (n+1 - begin) *
                (d[arr[begin]][arr[begin+1]]+
                d[arr[begin]][arr[end]]) +
                (n+1 - end) *
                (d[arr[begin+1]][arr[(end+1)%mod]
                ] +
                d[arr[end]][arr[(end+1)%mod]]);

            if(tmp2 < arrcost + min){
                a = begin + 1;
                i = 0;
                while(i <= begin){
                    tmparr[i] = arr[i];
                    i++;
                }
                for(j = end ;j >= begin+1 ; j--){
                    tmparr[i] = arr[j];
                    i++;
                }
                for(j = end + 1 ; j <= n ; j++) {
                    tmparr[j] = arr[j];
                }
                min = tmp2 - arrcost;
            }
        }
    }

    for( j = 0 ; j <= n ; j++){
        arr[j] = tmparr[j];
    }

    free(tmparr);
}

void moveForward(int arr[], int lmax)
{
    int i, j, k, l, tmp, min;
    int *tmparr, *tmpx;
    vector v;

    min = 0;
    tmparr =(int *) malloc(sizeof(int) * (N+1));
    tmpx = (int *) malloc(sizeof(int) * (N+1));
    for(i = 0 ; i <= n ; i++){
        tmparr[i] = arr[i];
    }
```

```
	calculateDelta(arr);
	int mod = n+1;
	for(k = 1 ; k <= lmax ; k++){
		for( i = 0 ; i < n - k ; i ++){
			for( j = i + k + 1; j <= n ; j++){

				tmp = k * ( delta[j - 1] -
				delta[i + k]) - (j - i - k) *
				(delta[i + k - 1] - delta[i]) +
				(n +1- i) *
				(d[arr[i]][arr[i+k+1]] -
				d[arr[i]][arr[i+1]])+ (n +1- j) *
				(d[arr[i+k]][arr[(j+1)%mod]] -
				d[arr[j]][arr[(j+1)%mod]])+
				(n+1-j+k) * (d[arr[j]][arr[i+1]])
				- (n+1-i-k) *
				(d[arr[i+k]][arr[i+k+1]]);


				if(tmp < min){
					vector_init(&v);
					for(l = 0 ; l <= n ; l++){

						vector_add(&v,(void *)
						arr[l]);
					}
					min = tmp;
					vector_moveSegment(&v, i+1,
					j+1, k);
					for(l = 0 ; l <= n ; l++){
						tmparr[l] = (int)
						vector_get(&v, l);
					}
					vector_free(&v);
				}
			}
		}
	}

	for( j = 0 ; j <= n ; j++){
		arr[j] = tmparr[j];
	}

	free(tmparr);
	free(tmpx);
}

void moveBackward(int arr[], int lmax)
{
	int i, j, k, l, tmp, min;
	int *tmparr, *tmpx;
	vector v;

	min = 0;
	tmparr = (int *) malloc(sizeof(int) * (N+1));
	for(i = 0 ; i <= n ; i++){
		tmparr[i] = arr[i];
	}
```

```
	calculateDelta(arr);
	int mod = n+1;
	for(k = 1 ; k <= lmax ; k++){
		for( i = 0 ; i <= n - k ; i ++){
			for( j = i + 1 ; j <= n - k; j++){
				tmp = -k * ( delta[j - 1] -
				delta[i]) + (j - i) *
				(delta[j + k -1] - delta[j])+
				(n + 1 - i) *
				(d[arr[i]][arr[j+1]] -
				d[arr[i]][arr[i+1]])+
				(n + 1 - j - k) *
				(d[arr[j]][arr[(j+k+1)%mod]] -
				d[arr[j+k]][arr[(j+k+1)%mod]])+
				(n + 1-i-k) *
				(d[arr[j+k]][arr[i+1]]) -
				(n+1-j) *
				(d[arr[j]][arr[j+1]]);

				if(tmp < min){
					vector_init(&v);
					for(l = 0 ; l <= n ; l++){
						vector_add(&v,(void *)
						arr[l]);
					}

					min = tmp;
					vector_moveSegment(&v, j+1,
					i+1, k);
					for(l = 0 ; l <= n ; l++){
						tmparr[l] = (int)
						vector_get(&v, l);
					}
					vector_free(&v);
				}
			}
		}
	}

	for( j = 0 ; j <= n ; j++){
		arr[j] = tmparr[j];
	}

	free(tmparr);
}


void seqVND(int arr[], int lmax)
{
	int l, cost, tmpcost;
	int * tmpx;

	cost = calculateCost(arr);
	tmpx = (int *) malloc(sizeof(int) * (N+1));
	for( j = 0 ; j <= n ; j++){
		tmpx[j] = arr[j];
	}

	l = 1;
```

```
	while(l <= 4){
		switch (l)
		{
		case 1:
			oneOpt(tmpx);
			break;

		case 2:
			twoOpt(tmpx, lmax);
			break;

		case 3:
			moveForward(tmpx, lmax);
			break;

		case 4:
			moveBackward(tmpx, lmax);
			break;
		}

		tmpcost = calculateCost(tmpx) ;
		if(tmpcost < cost){
			cost = tmpcost;
			for(i = 0 ; i <= n ; i++){
				arr[i] = tmpx[i];
			}
			l = 1;
		}
		else{
			l++;
		}
	}

	free(tmpx);
}

void mixVND( int arr[], int lmax)
{
	int improve, i, j, k, tim, tmpcost, tmp2, arrcost;
	int *tmparr;

	improve = 1;
	tmparr = (int *) malloc(sizeof(int) * (N + 1));
	arrcost = initialCost;

	for( j = 0 ; j <= n ; j++){
		tmparr[j] = arr[j];
	}
	seqVND(tmparr, lmax);
	tmpcost = calculateCost(tmparr);
	if(arrcost > tmpcost){
		for(j = 0 ; j <= n ; j++){
			arr[j] = tmparr[j];
		}
		arrcost = tmpcost;
	}

	while(improve == 1){
		improve = 0;
```

```
		for(I = 1 ; i <= n-7 && improve == 0 ; i++){
			for(k=i+4; k<=n-3 && improve == 0;k++){

				for( j = 0 ; j <= n ; j++){
					tmparr[j] = arr[j];
				}

				tmp2 = tmparr[k+1];
				tmparr[k+1] = tmparr[i+1];
				tmparr[i+1] = tmp2;

				tmp2 = tmparr[i+3];
				tmparr[i+3] = tmparr[k+3];
				tmparr[k+3] = tmp2;

				seqVND(tmparr, lmax);
				tmpcost = calculateCost(tmparr);
				if(arrcost > tmpcost){
					for(j = 0 ; j <= n ; j++){
						arr[j] = tmparr[j];
					}
					arrcost = tmpcost;
					improve = 1;
				}
				time(&stop);
				if(difftime(stop, start)>timmax){
					improve = 1;
				}
			}
		}
		if(difftime(stop, start)>timmax){
			improve = 0;
		}
	}
	free(tmparr);
}

void gvns(int arr[], int kmax, int tmax)
{
	int * tmparr, k, tim, cost;

	tmparr = (int *) malloc(sizeof(int) * (N+1));

	GreedyR(tmparr, 10);
	timmax = tmax;
	for(int i = 0 ; i <= n ; i++){
		arr[i] = tmparr[i];
	}

	initialCost = calculateCost(tmparr);
	tim = 0;
	time(&start);
	k = 1;
	while(tim <= tmax){
		k = 1;
		while(k <= kmax){
			shake(tmparr, k);
			mixVND(tmparr, 4);//o seqVND(tmparr,4);
```

```
                        cost = calculateCost(tmparr);
                        if(cost < initialCost){
                              time(&stop);
                              tim = difftime(stop, start);
                              printf("found a new solution of
                              cost = %i, in: %i seconds\n",
                              cost, tim);
                              for(int i = 0 ; i <= n ; i++){
                                    arr[i] = tmparr[i];
                                    initialCost = cost;
                                    printf("%i ", arr[i]);
                              }
                              printf("\n");
                              k = 1;
                        }
                        else{
                              for(int i = 0 ; i <= n ; i++){
                                    tmparr[i] = arr[i];
                              }
                              k++;
                        }
                        time(&stop);
                        tim = difftime(stop, start);

                  }
            }
            free(tmparr);
            printf("stopping after %i seconds\n\n", tim);
}

void main(){
      char* filename = "istanze\\lin318.tsp";
      int k, *arr, *tmparr, rcost, cost, tmp;
      FILE * out;
      errno_t err = 0;
      vector v;

      if( (err = fopen_s(&out, "output.txt", "w"))!= 0 ){
            printf("Could not open file! \n");
            return;
      }

      readInput(filename);
      arr = (int *) calloc(N+1, sizeof(int) );
      tmparr = (int *) calloc( N+1, sizeof(int) );
      gvns(arr, 10, 600);
      calculateDelta(arr);
      cost = calculateCost(arr);
      fprintf_s(out,"Instance: %s\n", filename);
      fprintf_s(out, "optimal solution cost = %i\n",
      cost);
      fprintf_s(out, "optimal solution cost, while keep-
      ing in mind the return cost, is: %i\n\n", cost);
      printf("optimal solution cost, omitting the return
      cost, is: %i\n\n", cost - delta[n] );
      fflush(out);

      free(arr);
      free(tmparr);
```

```
        fclose(out);
}
```