ABSTRACT

Il presente progetto riguarda la progettazione di un software in grado di risolvere istanze del problema del Commesso Viaggiatore applicando differenti algoritmi risolutori. L'obiettivo di questo testo è quello di descrivere le tecniche utilizzate e di confrontare i risultati ottenuti in termini di efficienza e bontà della soluzione prodotta. Verrà fornita una descrizione degli strumenti e l'ambiente di sviluppo utilizzati e sarà analizzato il codice di programmazione realizzato; non mancheranno paragrafi dedicati ad approfondire concetti teorici senza i quali la comprensione del codice potrebbe risultare meno chiara.

INTRODUZIONE

Questa capitolo introduttivo è dedicato alla storia, alle applicazioni e alle correnti sfide riguardanti uno dei più importanti problemi che la disciplina di Ricerca Operativa si trova ad affrontare, ossia il problema del commesso viaggiatore(Travelling Salesman Problem-TSP). Il nome deriva dalla sua più tipica rappresentazione: data una rete di città, connesse tramite delle strade, si vuole trovare il percorso di minore distanza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza. Per quanto detto, risulta naturale modellare il TSP come un grafo pesato i cui nodi modellizzano le città relative al problema in questione mentre i possibili collegamenti tra le località sono modellati con gli archi del grafo i cui pesi possono rappresentare,per esempio, la distanza esistente fra la coppia di nodi collegati dall' arco. Chiaramente è possibile assegnare i pesi in modo arbitrario secondo le nostre esigenze, ad esempio si potrebbe anche tenere conto dei tempi di percorrenza o di eventuali pedaggi presenti nei singoli percorsi. Come è facile immaginare, il TSP può essere quindi utilizzato per una infinità di problemi pratici ma anche teorici.
Il problema del commesso viaggiatore riveste un ruolo notevole nell' ambito di problemi di logistica distributiva, detti anche problemi di routing. Questi riguardano l'organizzazione di sistemi di distribuzione di beni e servizi. Esempi di problemi di questo genere sono la movimentazione di pezzi o semilavorati tra reparti di produzione, la raccolta e distribuzione di materiali, la distribuzione di merci da centri di produzione a centri di distribuzione. Sebbene le applicazioni nel contesto dei trasporti siano le più naturali per il TSP, la semplicità del modello ha portato a molte applicazioni interessanti in altre aree. Un esempio può essere la programmazione di una macchina per eseguire fori in un circuito. In questo caso i fori da forare sono le città e il costo del viaggio è il tempo necessario per spostare la testa del trapano da un foro all'altro. Il problema del commesso viaggiatore risulta essere NP-hard: questo significa che, al momento, non è noto in letteratura un algoritmo che lo risolva in tempo polinomiale. Poichè esiste sempre una istanza per cui il tempo di risoluzione cresce esponenzialmente non è sempre possibile utilizzare algoritmi esatti per risolvere il TPS. Risulta quindi necessario fornire algoritmi euristici, in grado di risolvere in modo efficace istanze con un numero elevato di nodi in tempi ragionevoli.
Problemi matematici riconducibili al TSP furono trattati nell' Ottocento dal matematico irlandese Sir William Rowan Hamilton e dal matematico Britannico Thomas Penyngton. Nel 1857, a Dublino, Rowan Hamilton descrisse un gioco, detto Icosian game, a una riunione della British Association for

the Advancement of Science. Il gioco consisteva nel trovare un percorso che toccasse tutti i vertici di un icosaedro, passando lungo gli spigoli, ma senza mai percorrere due volte lo stesso spigolo. L'icosaedro ha 12 vertici, 30 spigoli e 20 facce identiche a forma di triangolo equilatero. Il gioco, venduto alla ditta J. Jacques and Sons per 25 sterline, fu brevettato a Londra nel 1859, ma vendette pochissimo. Questo problema è un TSP nel quale gli archi che collegano vertici adiacenti, e quindi corrispondono a spigoli dell'icosaedro, sono consentiti e gli altri no (si può pensare che richiedano moltissimo tempo e quindi vadano sicuramente scartati), per tale ragione si tratta di un caso molto particolare di TSP. La forma generale del TSP fu invece studiata solo negli anni Venti e Trenta del ventesimo secolo dal matematico ed economista Karl Menger. Tuttavia, per molto tempo non si ebbe altra idea che quella di generare e valutare tutte le soluzioni, il che mantenne il problema praticamente insolubile. Il numero totale dei differenti percorsi possibili attraverso le n città è facile da calcolare: data una città di partenza, ci sono a disposizione $(n-1)$ scelte per la seconda città, $(n-2)$ per la terza e cosvia. Il totale delle possibili scelte tra le quali cercare il percorso migliore in termini di costo è dunque $(n-1)!$, ma dato che il problema ha simmetria, questo numero va diviso a metà . Insomma, date n città, ci sono $\frac{(n-1)!}{2}$ percorsi che le collegano.
Solo nel 1954, George Dantzig, Ray Fulkerson e Selmer Johnson proposero un metodo più raffinato per risolvere il TSP su un campione di $n = 49$ città: queste rappresentavano le capitali degli Stati Uniti e il costo del percorso era calcolato in base alle distanze stradali.
Nel 1962, Procter and Gamble bandì un concorso per 33 città, nel 1977 fu bandito un concorso che collegasse le 120 principali città della Germania Federale e la vittoria andò a Martin Grötschel oggi Presidente del Konrad-Zuse-Zentrum für Informarionstechnik Berlin(ZIB) e docente presso la Technische Universität Berlin(TUB).
Nel 1987 Padberg e Rinaldi riuscirono a completare il giro degli Stati Uniti attraverso 532 città. Nello stesso periodo Groetschel e Holland trovarono il TSP ottimale per il giro del mondo che passava per 666 mete importanti. Nel 2001, Applegate, Bixby, Chvátal, and Cook trovarono la soluzione esatta a un problema di 15.112 città tedesche, usando il metodo cutting plane, originariamente proposto nel 1954 da George Dantzig, Delbert Ray Fulkerson e Selmer Johnson. Il calcolo fu eseguito da una rete di 110 processori della Rice University e della Princeton University. Il tempo di elaborazione totale fu equivalente a 22,6 anni su un singolo processore Alpha a 500 MHz. Sempre Applegate, Bixby, Chvtal, Cook, e Helsgaun trovarono nel Maggio del 2004 il percorso ottimale di 24,978 città della Svezia. Nel marzo 2005, il TSP riguardante la visita di tutti i 33.810 punti in una scheda di circuito fu risolto usando CONCORDE: fu trovato un percorso di 66.048.945 unità, e provato che non poteva esisterne uno migliore. L'esecuzione richiese approssimativamente 15,7 anni CPU. Ai giorni nostri il risolutore Concorde per il problema del commesso viaggiatore è utilizzato per ottenere soluzioni ottime su tutte le 110 istanze della libreria TSPLIB; l' istanza con più nodi in assoluto ha 85,900 città.

MODELLO MATEMATICO

Nella sua formalizzazione più generale, il problema del Commesso Viaggiatore consiste nell'individuare un circuito hamiltoniano di costo minimo per un dato grafo orientato $G = (V, A)$, dove $V = \{v_1, \ldots, v_n\}$ è un insieme di n nodi e $A = \{(i, j) : i, j \in V\}$ è un insieme di m archi[1].

[1] Chiaramente sia n che m sono interi positivi

Senza perdita di generalità, si suppone che il grafo G sia completo e che il costo associato all'arco $[i, j]$, che indicheremo con c_ij, sia non negativo. Si osserva che aver imposto $c\_ij \geq 0$ non è limitativo poichè è sempre possibile sommare a tutti i costi una costante sufficientemente elevata che li renda positivi senza alterarne l'ordinamento delle soluzioni. A differenza di quanto detto in precedenza, per tutto il proseguimento della tesi supporremmo il grafo G non orientato: tale scelta deriva dal fatto che c_ij nel nostro lavoro rappresenta sempre la distanza (tipicamente euclidea) fra i vertici i e j si ha che:

$$c_{ij} = c_{ji}$$

ossia il costo associato ad un arco non dipende dalla direzione dell'arco stesso. Quando il grafo è non orientato la famiglia di coppie non ordinate di elementi di V, ossia l'insieme degli archi, viene indicato con E.
Definito il problema forniamo ora una sua possibile formulazione in termini di PLI. Introducendo le seguenti variabili decisionali:

$$x_e = \begin{cases} 1 & \text{se il lato } e \in E \text{ viene scelto nel circuito ottimo} \\ 0 & \text{altrimenti} \end{cases}$$

si ottiene il problema:

$$\min \underbrace{\sum_{e \in E} c_e x_e}_{\text{\_costo circuito}} \tag{1}$$

$$\underbrace{\sum_{e \in \delta(V)} x_e = 2}_{\text{due lati incidenti in } v} \quad , \forall v \in V \tag{2}$$

$$0 \leq x_e \leq 1 \text{ intera} \ , \forall e \in E \tag{3}$$

L' insieme di vincoli definiti dalla (2) vengono chiamati vincoli di grado e impongono che in ogni vertice incidano esattamente due lati. In questa forma il modello è compatto dato che il numero di vincoli è polinomiale rispetto alla dimensione dell'istanza ma non è completo poichè è sprovvisto dei vincoli di subtour che impediscono soluzioni il cui grafo risulta non connesso.

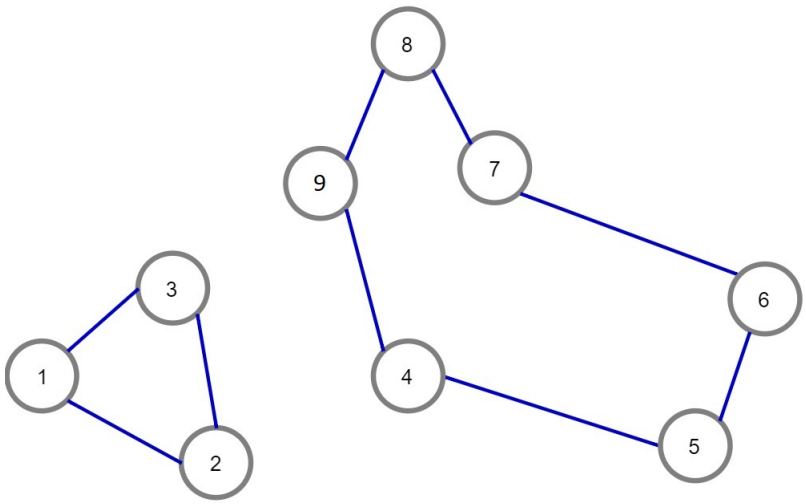

Figura 1: Soluzione con due subtour

Una possibile formulazione per l'eliminazione dei subtour, detta appunto **subtour elimination**, risulta essere:

$$\sum_{e \in E(S)} x_e \geq 1 \ , \ \forall S \subsetneq V : \ 1 \in S \ , \ |S| \geq 2 \tag{4}$$

Il vincolo (4) indica che se si considera un sottoinsieme $S \subsetneq V$, che includa il vertice numerato con il simbolo 1, allora il taglio di G indotto da S:

$$\delta(S) = \{[i,j] \in E : i \in S, j \notin S\}$$

deve contenere almeno un lato appartenente ad E: poichè tutti i subtour violano tale vincolo la soluzione ottima non potrà contenerne al suo interno. Essendo il numero di questi vincoli pari ai sottoinsiemi S distinti, il numero di tali vincoli risulta esponenziale rispetto alla dimensione dell'istanza. In particolare il valore di S, dato un numero n di nodi, è 2^n: questo perchè associando un bit ad ogni vertice (il cui valore definisce se appartiene o meno al sottoinsieme) un qualsiasi sottoinsieme risulta identificato da una sequenza di n bit è quindi possibile definirne 2^n distinti. In realtà avendo noi imposto che il vertice 1 appartenga sempre ad ogni S e che $S \subsetneq V$, si ha che il numero di vincoli di subtour risulti pari a $2^{n-1} - 1$.
Una seconda formulazione equivalente per esprimere i vincoli di subtour è la seguente:

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \ , \ \forall S \subsetneq V \ , \ |S| \geq 2 \tag{5}$$

La gestione di un numero esponenziale di vincoli implica in genere tempi di risoluzione troppo elevati. Nella pratica però non è necessario utilizzare tutti i vincoli di subtour elimination, è sufficiente considerarne un numero molto più ridotto. Non potendo conoscere in anticipo quali siano quelli

essenziali sarà nostro compito progettare un opportuno separatore: ossia una funzione che fornita in ingresso una soluzione x* ottima per il modello corrente generi tutti i vincoli violati.

FILE DI INPUT

Le istanze del problema del Commesso Viaggiatore fornite in input al programma sono state selezionata da un libreria presente al seguente indirizzo web:

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

Ogni istanza è memorizzata in un file di testo in un formato ben preciso, è stato quindi possibile progettare un opportuno parser che automaticamente riesca a estrapolare le informazioni contenute e popolare le strutture dati da noi create[2].

STRUTTURA DEL PROGETTO

I file che compongono il programma realizzato sono stati organizzati nel modo seguente; all' interno della cartella radice, da noi chiamata TSPCsharp, si sono create le seguenti sottocartelle:

- **Src** contenente il progetto di Visual Studio;
- **Data** include le istanze del problema del commesso viaggiatore appartenenti alla TSPlib;
- **Concorde** contenente i file sorgenti in linguaggio C del programma Concorde[3] la cui trattazione è rimandata al Capitolo X.

Il software sviluppato è composto da dieci classi, riportiamo di seguito il nominativo di ognuna di esse:

- Instances
- ItemList
- PathGenetic
- PathStandard
- Point
- Program

[2]Nessun altro tipo di input è supportato

[3]Concorde è un software freeware sviluppato da **William Cook** per la risoluzione di problemi TSP

- Tabu
- TSP
- TSPLazyConsCallback
- Utility

Per le classi Point, Instances, Program, TSP e Utility verrà fornita una descrizione in questo capitolo, le rimanenti classi verranno presentate nei capitoli successivi poichè una loro trattazione risulterebbe in questo momento prematura.

CLASSE POINT

La classe Point è stata realizzate al fine di memorizzare le coordinate in due dimensioni di un singolo nodo n, a tal fine sono presenti due variabili private, accessibili attraverso i propri metodi get e set, di tipo **double** chiamate rispettivamente **x** e **y**. Il costruttore della classe non fa altro che ricevere in input i valori da assegnare a queste ultime. La classe presenta inoltre un ulteriore metodo pubblico e statico chiamato **Distance** che permette il calcolo della distanza tra due nodi:

```
public static double Distance(Point p1, Point p2, string pointType)
```

- **p1**: Rappresenta il primo nodo;
- p2: Rappresenta il secondo nodo;
- **pointType**: Rappresenta il modo con cui il costo relativo al lato che congiunge p1 con p2 viene calcolato; i valori che questo parametro può assumere sono i seguenti:
  - EUC_2D
  - ATT
  - MAN_2D
  - GEO
  - MAX_2D
  - CEIL_2D

A titolo di esempio riportiamo il codice eseguito nel caso in cui pointType risulti uguale a EUC_2D dove il costo del lato deve risultare pari alla distanza euclidea dei due nodi.

```
double xD = p1.X - p2.X;
double yD = p1.Y - p2.Y;

if (pointType == "EUC\_2D")
{
    return (int)(Math.Sqrt(xD * xD + yD * yD) + 0.5);
}
else if ...
```

Per quanto riguarda gli altri metodi di calcolo della distanza rimandiamo il lettore alla visione del codice.

CLASSE INSTANCE

La classe Instance è stata creata per memorizzare tutti i dati caratterizzanti l'istanza del problema del Commesso Viaggiatore. La tabella sottostante fornisce un elenco di variabili ed array definite all'interno della classe assieme ad una loro breve descrizione.

Tipo di dato	**Nome**	**Descrizione**
int	nNodes	Rappresenta il numero di nodi dell' istanza del problema del Commesso Viaggiatore.
Point[ ]	coord	Vettore di Point contenente le coordinate di tutti i vertici del grafo.
string	edgeType	Definisce la modalità con cui calcolare la distanza fra due nodi.
double	timeLimit	Definisce la quantità massima di tempo che il programma dispone per il calcolo della soluzione.
double	inputFile	Rappresenta il nome del file di input contenente l'istanza del problema del Commesso Viaggiatore.
double	tStart	Rappresenta i secondi trascorsi dall'attivazione del cronometro al reale inizio delle operazioni di calcolo per la risoluzione del problema

double	xBest	Rappresenta il costo della soluzione ottima restituira da cplex.
double	tBest	Contiene la qantità di tempo impiegata per il calcolo della soluzione ottima.
double	bestSol	Rappresenta la soluzione ottima ritornata da Cplex.
double	bestLb	Rappresenta il miglior lower bound attualmente calcolato.
double	bestLb	Rappresenta il miglior lower bound attualmente calcolato.
double	bestLb	Rappresenta il miglior lower bound attualmente calcolato.

xMin e yMin ... mi servivano per gnuplot ma dato che ho messo l'autoscale credo non servano più. per xBest ok
L' unico metodo appartenente a questa classe, esclusi i vari getter e setter, è **Print**, la cui firma risulta essere:

```
static public void Print(Instance inst)
```

dove:

- **inst**: oggetto della classe Instance contente tutti dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' tente;

Tale metodo stampa a video le coordinate di tutti i nodi memorizzati dentro **inst**. Viene di seguito riportato il codice:

```
for (int i = 0; i < inst.NNodes; i++)
    Console.WriteLine("Point #" + (i + 1) + "= (" + inst.Coord[i].X + ","
       + inst.Coord[i].Y + ")");
```

CLASSE PROGRAM

La classe Program contiene il metodo Main che, come noto, rappresenta il punto di inizio del programma: attraverso le funzionalità di Visual Studio esso riceve in input l'array **argv** di stringhe contenente i parametri di input forniti dall'utente come il nome del file contenente l'input ed il time limit per la sua risoluzione. Appartengono a questa classe anche i metodi **ParseInst** e **Populate**: rispettivamente forniscono il parser per **argv** ed il parser del file di ingresso indicato con conseguente inizializzazione delle coordinate dei nodi. Firma e implementazione di tali metodi è rimandata al successivo capitolo.
All'interno del metodo **Main** vengono eseguite in ordine le seguenti attività:

- Creata una istanza della classe **Instance** ed invocati i due metodi precedentemente nominati.
- Creato un oggetto della classe **Stopwatch**. Questa classe è fornita direttamente da Visual Studio appartenente al Namespace **System.Diagnostics** è fornisce le funzionalità di un cronometro compatibile al multithreading.
- Invocato il metodo **TSPOpt** della classe **TSP** passandogli come parametri i due oggetti di tipo Instance e Stopwatch.
- In caso di risultato positivo (una soluzione del problema è stata trovata) viene mostrato a video il risultato ottenuto ed il tempo di calcolo trascorso.
- Viene effettuata una pulizia dei file creatisi durante l'esecuzione del problema.

In tutto il nostro progetto si è cercato di utilizzare il minor numero possibile di variabili globali, in particolare solo all'interno di questa classe ne sono state definite due di seguito descritte:

Tipo di dato	**Nome**	**Valore**	**Descrizione**
int	VERBOSE	5	Regola quanto output il programma mostra a video: si è scelto di condizionare l' esecuzione di molte righe di codice che producevano una stampa a video in base al valore assunto da questa variabile. Si è deciso di ristringere il suo valore da 1 a 9, quando assume il valore 9 viene riportato a video il maggior numero possibile di stampe.
int	TICKS_PER_SECOND	1000	Cplex utilizza i così detti ticks come unità di misura per il tempo di calcolo, questa costante indica quanti ne trascorrono in un secondo.

CLASSE TSP

La classe **TSP** è stata pensata come il cuore del programma in quanto lo scheletro di tutti i metodi di risoluzione implementatisi trova al suo interno. Contiene un unico metodo pubblico che rappresenta quindi l'unico entry point per utilizzare questa classe: **TSPOpt**.

```
static public bool TSPOpt(Instance instance, Stopwatch clock)
```

Come già specificato nella descrizione della classe Program, TSPOpt è invocato dal metodo Main e pertanto maggiori dettagli riguardanti i suoi parametri di ingresso possono essere trovati nella sezione precedente.
TSPOpt si preoccupa di istanziare i vari elementi utilizzati da tutti i metodi di risoluzione[4], fornire all'utente un'interfaccia grafica che gli permetta di scegliere quale di questi ultimi voglia utilizzare e di conseguenza invoca il metodo privato della classe associato alla scelta effettuata.
Entrando nello specifico per quanto riguarda gli elementi inizializzati troviamo un oggetto della classe **Cplex** che come già accennato in precedenza ci permetterà di stabilire una connessione con il programma Cplex ed utilizzarlo per la risoluzione del modello matematico, ed un oggetto **Process** che sostanzialmente viene da noi utilizzato per inizializzare e comunicare con il programma **GNUPlot**[5].

CLASSE UTILITY

La classe Utility può essere considerata come una libreria: contiene al suo interno solamente metodi **statici** che si è deciso di raggruppare al suo interno per rendere il codice il più compatto e leggibile possibile.

INTERPRETAZIONE FILE DI INPUT

Lo sviluppo del programma è iniziato realizzando una opportuna funzione per interpretare correttamente i parametri di ingresso forniti dall'utente. Oltre al nome del file di testo contenente i dati relativi all'istanza del problema che si vuole risolvere, all'utente è richiesto di fornite un time limit(espresso in secondi) e di scegliere con quale algoritmo risolvere l'istanza da esso fornita. Si è deciso di ricevere da riga di comando il nome del file e il time limit; per quanto riguarda la scelta dell'algoritmo risolutore ed eventuali parametri da esso richiesti si è preferito realizzare una semplice interfaccia grafica per favorire l'utente. Visual Studio, all'interno delle proprietà del progetto, permette di definire una stringa come parametro di ingresso per il programma. Questa viene automaticamente separata in sottostringhe utilizzando come separatore il carattere di spazio e fornito in ingresso al metodo Main. Allo stato attutale è gestita solamente la possibilità di fornire in ingresso il nome del file contenente i dati ed il timelimit per la ricerca della soluzione. Per ottenere una migliore organizzazione e chiarezza per il nostro lavoro è stato deciso di utilizzare questa regola per la costruzione della stringa di ingresso: ogni parametro inseritovi deve essere preceduto da una parola chiave che lo identifica il cui primo carattere deve essere '-'. Questa tecnica si rileverà utile

[4] Fatta eccezione per l'UserCutCallBack che è gestita esternamente da una DLL
[5] Per maggiori dettagli si veda la sezione dedicata a GNUPlot

anche in futuro nel caso si decidesse di ampliare la lista di parametri di ingresso.
La funzione che interpreta correttamente gli argomenti forniti in input dalla riga di comando è stata chiamata ParseInst ed ha la seguente intestazione:

```
static void ParseInst(Instance inst, string[] input)
```

- **inst**: rappresenta il riferimento all'istanza della classe Instance dichiarata nel metodo Main, i valori letti vengono memorizzati al suo interno.
- **input**: rappresenta un vettore contenente i parametri di input forniti da riga di comando dall'utente.

Il metodo è composto da un semplice ciclo for che scandisce il vettore **input** cercando una parola chiave, se trovata la stringa successiva viene memorizzata correttamente dentro **inst**:

```
for (int i = 0; i < input.Length; i++)
{
   if (input[i] == "-file")
   {
     //Expecting that the next value is the file name
       inst.InputFile = input[++i];
       continue;
    }
    if (input[i] == "-timelimit")
    {
      //Expecting that the next value is the time limit in seconds
       inst.TimeLimit = Convert.ToDouble(input[++i]);
       continue;
    }
}
```

Nel caso in cui l' utente non fornisca il nome del file di input oppure il time limit viene lanciata una eccezione:

```
if (inst.InputFile == null || inst.TimeLimit == 0)
     throw new Exception("File input name and/or timelimit are missing");
```

METODO POPULATE

Il metodo Populate è utilizzato per la lettura dei dati contenuti all'interno del file di input e soprattutto alla loro memorizzazione all'interno di un oggetto di tipo **Instance** in modo tale che una volta conclusosi il metodo questo contenga tutte le informazioni necessarie per la creazione del modello matematico.
I file di input presenta una struttura pressoché identica tra loro e cioè una divisione in sezioni identificate da parole chiave. Fatta eccezione per la sezione che descrive le coordinate dei nodi, tutte le altre si sviluppano in una sola riga la cui struttura è del tipo:

<parolaChiave> : < valore>

Di seguito sono riportati i valori che possono essere assunti dalle parole chiavi e il significato del contenuto della relativa sezione:

- NAME:<string>
  - nome con cui l' istanza è nota in letteratura.
- TYPE:<string>
  - indica il tipo dell' istanza. Nel nostro ambito sarà sempre TSP.
- **COMMENT:<string>**
  - include informazioni aggiuntive, solitamente contiene il nome deli gli autori che hanno proposto l' istanza.
- **DIMENSION:<integer>**
  - indica il numero di nodi.
- **EDGE WEIGHT TYPE:<string>**
  - Definisce il modo con cui il costo del lato deve essere calcolato, i possibili valori che può assumere il contenuto di questa sezione sono stati già presentati a pagina 7 durante la descrizione del metodo Distance.
- **NODE COORD SECTION**:
  - Il contenuto di questa sezione si sviluppa in più righe; in ogni riga troviamo nell' ordine:
    * Un numero progressivo intero positivo che comincia da 1 e che identifica il nodo. Osserviamo che anche se in input il primo nodo è numerato a partire da 1, nel vettore Point di inst le coordinate saranno memorizzate a partire dall'indice 0[6].
    * Un numero reale positivo che definisce la coordinata x del nodo.
    * Un numero reale positivo che identifica la coordinata y del nodo.

Il file di testo termina sempre con la stringa **EOF** che indica la fine del file di testo.
Per poter leggere il contenuto di un file è necessario inizializzare una nuova istanza della classe StreamReader passando come parametro al costruttore il percorso ove tale file è collocato.

```
StreamReader sr = new StreamReader("..\\..\\..\\..\\Data\\" +
    inst.InputFile)
```

[6]Tale scelta è per mantenere una conformità con la metrica adottata dal linguaggio C# per l'enumerazione degli elementi dei vettori, nel caso in cui le coordinate vengano visualizzate a video il loro indice viene comunque incrementato di uno

Il metodo ReadLine() della classe StreamReader ritorna, come stringa, il contenuto di una intera riga del file la quale viene memorizzata all'interno di una variabile di tipo string chiamata **line**. Poichè si vuole leggere tutto il contenuto del file, è necessario invocare ReadLine() ciclicamente sull' oggetto **sr** finchè line risulta diversa da null oppure viene incontrata la parola chiave **EOF**.

```
while ((line = sr.ReadLine()) != null)
{
    ...

    //This line signals the end of the file
    if (line.StartsWith("EOF"))
    {
    Instance.Print(inst);
    Console.WriteLine(line);
    //Correct end of the file
    break;
    }

    ...
}
```

Poichè ogni riga inizia con una nota parola chiave, per prelevare il contenuto di una sezione e memorizzarlo in un opportuno campo di inst, è sufficiente confrontare la prima stringa di ogni riga con una delle noti parole chiavi. Per far ciò si è usato il metodo StartWith della classe String, la cui firma è:

```
public bool StartWith(string value)
```

Questo metodo, applicato alla variabile line, determina se la prima stringa di line corrisponde alla stringa value specificata all'atto dell' invocazione del metodo. Nel caso in cui il confronto dia esito positivo, per prelevare il contenuto della sezione è necessario applicare i metodi IndexOf e Remove sempre alla variabile line; l'intestazione di tali metodi è riportata di seguito:

```
public int IndexOf(string value,int startIndex)
```

dove:

- **value**: stringa da cercare.
- **startIndex**: posizione iniziale della ricerca.

```
public string Remove(int startIndex, int count)
```

dove:

- **startIndex**: posizione da cui iniziare l'eliminazione dei caratteri.
- **count**: numero di caratteri da eliminare.

Per quanto detto, risulta immediata la comprensione del codice necessario per prelevare il contenuto della sezione e memorizzarlo dentro un oggetto di tipo **Instance** chiamando il metodo setter adeguato:

```
inst.SetterName = (line.Remove(0, line.IndexOf(:) + 2));
```

Il codice riportato deve chiaramente effettuare un cast per i tipi diversi da string, i metodi necessari sono già disponibili all'interno della classe **Convert** di C#.
Una volta che ci troviamo all'interno della sezione **NODE COORD SECTION** la lettura delle coordinate viene eseguita eseguendo ciclicamente il seguente codice:

```
string[] elements = line.Split(new[]{ ' ' } ,
    StringSplitOptions.RemoveEmptyEntries);

int i = Convert.ToInt32(elements[0]);

inst.Coord[i - 1] = new Point(Convert.ToDouble(elements[1].Replace(".",
    ",")), Convert.ToDouble(elements[2].Replace(".",",")));
```

Il metodo Split della classe String ritorna un array contenente in ogni elemento una sottostringa della stringa a cui tale metodo è applicato. Le sottostringhe vengono estratte dalla stringa in base ai caratteri delimitatori specificati all'atto dell'invocazione del metodo, quest'ultimo ha diversi overload: quello di nostro interesse è riportato di seguito.

```
public string[] Split(char[] separator, StringSplitOptions options)
```

dove:

- **separator**: array i cui elementi definiscono i separatori della stringa. Nel nostro caso è un array con un solo elemento contenente il carattere ' '.
- **options**: A questo parametro possono essere passate solo i seguenti due valori dell' enumerazione StringSplitOptions:
  - **StringSplitOptions.RemoveEmptyEntries**: indica che gli elementi dell'array ritornato non possono essere stringhe vuote. Questo è l'opzione da noi selezionata.
  - **StringSplitOptions.None**: indica che gli elementi dell'array ritornato possono essere stringhe vuote.

Ogni coordinata letta viene tradotta in un oggetto di tipo **Point** il quale è a sua volta memorizzato all'interno del vettore **Coord** dell'oggetto di tipo **Instance** nella posizione indice letta.
Come nota conclusiva specifichiamo che C# utilizza come separatore tra parte intere e parte decimale di un numero il carattere ',' e non il carattere '.' utilizzato per nei file di input. E' quindi necessaria una modifica delle stringhe lette attraverso il metodo non statico della classe string:

```
public string Replace(string oldValue, string newValue)
)
```

dove:

- **oldValue**: stringa sa sostituire;
- **newValue**: stringa con cui sostituire tutte le occorrenze di oldValue.

COSTRUZIONE DEL MODELLO

In questo paragrafo vedremo come è possibile creare da programma un modello matematico attraverso l'uso di alcune routine appartenenti alla libreria di Cplex. Esula dallo scopo di questa tesi fornire al lettore una descrizione del funzionamento di Cplex da iterativo.

COSTRUZIONE MODELLO IN C

Per istanziare un nuovo modello di programmazione lineare è necessario inizializzare un **environment** di Cplex utilizzando la funzione **CPXopenCPLEX** la quale ritorna un puntatore all'environment creato, la firma di tale funzione è:

```
CPXENVptr CPXopenCPLEX(int* status\_p)
```

dove:

- **status_p**: puntatore ad una variabile di tipo intero usato per ritornare un eventuale codice di errore.

Ad un enviroment è possibile associare uno o più modelli attraverso il comando **CPXcreateprob**, la cui intestazione è:

```
CPXLPptr CPXcreateprob(CPXCENVptr env, int * status\_p, const char *
   probname\_str
```

dove:

- **env**: puntatore all'environment sul quale si è deciso di creare il modello;
- **status_p**: puntatore ad una variabile di tipo intero usato per ritornare un eventuale codice di errore;
- **probname_str**: rappresenta un array di caratteri che definisce il nome del modello creato.

Tale funzione ritorna un puntatore al modello creato: questo risulta vuoto poichè privo di funzione obbiettivo, variabili e vincoli.

Procediamo quindi al loro inserimento partendo definendo contemporaneamente le variabili e il loro coefficiente nella funzione obiettivo; è possibile procedere in più modi, quello da noi scelto è di utilizzare la funzione **CPXnewcols** la cui firma è:

```
int CPXnewcols (CPXENVptr env,CPXLPptr lp,int ccnt,double *obj, double
    *lb, double *ub, char *ctype, char **colname);
```

dove:

- **env** : puntatore all'enviroment di Cplex nel quale vuole essere inserito il modello.
- **lp** : puntatore al problema di programmazione lineare.
- **ccnt** : intero che indica il numero delle nuove variabili che vengono aggiunte al problema.
- **obj** : array contenente per ogni variabile il relativo coefficiente
- **lb** : array di lunghezza ccnt contenente il lower bound di ogni variabile aggiunta.
- **ub** : array contenente l'upper bound di ogni variabile aggiunta.
- **ctype** : array di lunghezza ccnt contenente il tipo di ogni variabile. I valori che un elemento di questo array può assumere sono:
  - '**C**': variabile continua
  - '**B**': variabile binaria
  - '**I**': variabile intera
- **colname** : array di lunghezza ccnt contenente puntatori ad array di char, a sua volta ognuno di essi deve contenere il nome della variabile aggiunta al modello.

Per motivi di semplicità non andremo ad inserire tutte le variabili contemporaneamente ma una ad una.

E' giunto quindi il momento di parlare di quali variabili vogliamo aggiungere al nostro modello tenendo presente che il medesimo discorso sarà applicato anche per la parte in C#. Sappiamo che per ogni coppia di nodi (i,j)[7] esiste un unico lato che li collega e che quest'ultimo è privo di direzione. Si presenta quindi la necessità di definire una convenzione per l'assegnazione del nome ai vari lati. La scelta adottata è la seguente: considerando due generici nodi **i** e **j** allora il loro lato sarà chiamato **x(i,j)** se $i < j$ oppure **x(j,i)** se $j < i$[8].
Questa convenzione offre anche un importante spunto per decidere con quale ordine memorizzare i vari parametri delle variabili (nome, coefficiente, lower bound ecc..): date due coppie distinti di nodi (i,j) e (v,w)[9] la posizione di memoria in cui viene memorizzata l'informazione riguardante la prima coppia è **inferiore** rispetto alla seconda se e solo se $(i < v)||(i == v \& j < w)$. In altre parole saranno memorizzate in ordine le informazioni per i nodi (1,2), (1,3), ..., (2,3), (2,4), ..., (n-1,n).

[7] Ricordiamo che i deve essere diverso da j in quanto per i problemi da noi considerati i cappi non sono ammessi
[8] Notiamo che per quanto espresso nella nota precedente non ha senso considerare il caso $i = j$
[9] dove assumiamo $i < j \& v < W$

Una ulteriore considerazione necessaria è la seguente: come mostrato poco fa il metodo **CPXnewcols** si aspetta il passaggio di diversi array mentre noi vorremmo utilizzare semplici variabili. La soluzione è molto semplice e consiste nell'anteporre il carattere & prima di ogni variabile in questo modo stiamo in realtà passando un puntatore alla sua locazione di memoria.

Le operazioni descritte sono state realizzate tramite il seguente codice:

```
double zero = 0.0; // one = 1.0;
char binary = 'B';

char **cname = (char **)calloc(1, sizeof(char *));          // (char **)
   required by cplex...
cname[0] = (char *)calloc(100, sizeof(char));

// add binary var.s y(i,j)

for (int i = 0; i < inst->nNodes; i++)
{
    for (int j = i + 1; j < inst->nNodes; j++)//Mi interessano solo le
       coppie con i<j
    {
        sprintf(cname[0], "x(%d,%d)", i + 1, j + 1);
        double obj = dist(inst->coord[i], inst->coord[j], inst->edgeType);
        double ub = 1.0;
        if (CPXnewcols(env, lp, 1, &obj, &zero, &ub, &binary, cname))
           printError(" ... errato CPXnewcols su x");//Aggiugo una
           variabile al modello
        if (CPXgetnumcols(env, lp) - 1 != xPos(i, j, inst)) printError("
           ... errata posizione per x");//Serve solo per controllare se
           la funzione xPos è corretta
    }
}
```

La funzione chiamata xPos riceve in ingresso un lato (i,j) del grafo e restituisce l'indice della variabile Cplex associata a quest'ultimo. Dato che risulta possibile effettuare errori nella realizzazione di questa funzione, in questo punto del codice è utile effettuare un controllo se il valore ritornato da xPos coincide con quello aspettato,in caso contrario viene sollevata una eccezione. Firma e dettagli implementativi di xPos saranno forniti nel paragrafo successivo in quanto è definita anche in C#.

Una volta definite le variabili è necessario creare i vincoli: per far ciò si è utilizzata la funzione **CPXnewrows**, la cui firma è:

```
int CPXnewrows(CPXCENVptr env, CPXLPptr lp, int rcnt, const double * rhs,
   const char * sense, const double * rngval, char ** rowname)
```

dove:

- **env**: puntatore all'enviroment di Cplex nel quale vuole essere inserito il modello.
- **lp**: puntatore al problema di programmazione lineare.

- **rcnt**: intero che definisce il numero di nuovi vincoli aggiunti al modello.
- **rhs**: array di lunghezza rcnt contenente il termine noto di ogni vincolo.
- **sense**: array di lunghezza rcnt i cui elementi possono assumere i seguenti valori:
  - '**L**': indica che il vincolo è una disuguaglianza il cui segno è $\leq$
  - '**E**': indica che il vincolo è una uguaglianza
  - '**G**': indica che il vincolo è una disuguaglianza il cui segno è $\geq$
  - '**R**' : indica che il vincolo è limitato
- **rngval**: variabile di tipo double contenente il valore 1.0;
- **rowname**: variabile di tipo char che assume il valore costante 'B';

Anche in questo caso anzichè aggiungere tutti i vincoli in una singola iterazione, risulta più semplice aggiungere un vincolo per volta invocando il metodo tante volte quante sono i vincoli da aggiungere.

COSTRUZIONE E RISOLUZIONE DEL MODELLO MATEMATICO IN C#

Per poter creare un modello matematico in Cplex, utilizzando come linguaggio di programmazione C# è necessario creare inizialmente una istanza della classe **Cplex**:

```
Cplex cplex = new Cplex();
```

Per creare il modello si associano, tramite opportune funzioni che descriveremo in questo paragrafo, all'istanza creata la funzione obbiettivo,le variabili e i vincoli del modello.
In C# le variabili del modello sono oggetti il cui tipo deve implementare l'interfaccia **INumVar**. Non è necessario creare da noi una nuova classe infatti ci viene fornito il metodo **NumVar** della classe **Cplex**:

```
public virtual INumVar NumVar(double lb, double ub, NumVarType type,
    string name)
```

dove:

- **lb**: Rappresenta il lower bound della variabile creata;
- **ub**: Rappresenta l' upper bound della variabile creata;
- **type**: Questo campo determina il tipo della variabile, può assumere i seguenti valori:
  - **NumVarType.Int**: Nel caso di variabile intera;
  - **NumVarType.Int**: Nel caso di variabile binaria;
  - **NumVarType.Float**: Nel caso di variabile continua;
- **name**: Nome identificativo della variabile.

Che come si può notare nella firma ha come tipo di ritorno un tipo di oggetto che implementa l'interfaccia da noi desiderata. Vedremo nel seguito della trattazione quanto utili risultano essere le funzionalità offerta de quest'ultima.
Introduciamo ora una seconda interfaccia **ILinearNumExpr** che come si può intuire viene implementata da oggetti che vogliono definire una espressione lineare. Anche in questo caso ci viene incontro la classe **Cplex** attraverso il metodo **LinearNumExpr**:

```
ILinearNumExpr expr = cplex.LinearNumExpr();
```

La variabile **expr** rappresenta quindi una espressione lineare che deve essere definita come:

$$\sum_{i=1}^{n} a_i x_i$$

dove x_i sono variabili di tipo **INumVar** mentre a_i è un coefficiente di tipo **double**. Per aggiungere all'oggetto **expr** una variabile del modello è necessario utilizzare il metodo **AddTerm** la cui intestazione è:

```
void AddTerm(INumVar var,double coef)
```

dove:

- **var**: variabile da aggiungere all'espressione;
- **coeff**: coefficiente della variabile aggiunta all'espressione.

L'implementazione da noi fornita per quanto riguarda la funzione obiettivo è la seguente:

```
//Populating objective function

for (int i = 0; i < instance.NNodes; i++)
{
  //Only links (i,j) with i < j are correct

  for (int j = i + 1; j < instance.NNodes; j++)
  {
      //zPos returns the correct position where to store the variable
         corresponding to the actual link (i,j)

     int position = zPos(i, j, instance.NNodes);

     z[position] = cplex.NumVar(0, 1, NumVarType.Int, "x(" + (i + 1) +
        "," + (j + 1) + ")");

     expr.AddTerm(z[position], Point.Distance(instance.Coord[i],
        instance.Coord[j], instance.EdgeType));
   }
}
```

Espressioni lineari definite in questo modo possono essere utilizzate sia per definire la funzione obiettivo del modello ma anche per i suoi vincoli.

Nel primo caso risulta sufficiente invocare i metodi non statici **AddMinimize** oppure **AddMaximize** della classe **Cplex** che rispettivamente definiscono una funzione obiettivo da minimizzare o da massimizzare, nel nostro caso:

```
cplex.AddMinimize(expr);
```

Per quanto riguarda i vincoli è necessario utilizzare i metodi **AddEq**, **AddLe**, **AddGe** che rispettivamente aggiungono al modello una equazione, una disequazione avente segno $\leq$, una disequazione avente segno $\geq$.
Nel nostro caso poichè ogni vincolo è una equazione riportiamo di seguito la firma della relativa funzione:

```
public virtual IRange AddEq(INumExpr e,double v,string name)
```

dove:

- **e**: Espressione contenente le variabili del vincolo;
- **v**: Termine noto del vincolo;
- **name**: Nome identificativo del vincolo.

Il codice completo diventa quindi:

```
for (int i = 0; i < instance.NNodes; i++)
{
 //Resetting expr
  expr = cplex.LinearNumExpr();

  for (int j = 0; j < instance.NNodes; j++)
  {
   //For each row i only the link (i,j) or (j,i) has coefficent 1
   //xPos return the correct position where link is stored inside the
      vector x

    if (i != j)//No loops with only one node
       expr.AddTerm(x[xPos(i, j, instance.NNodes)], 1);
  }

//Adding to the model the current equation with known term 2 and name
   degree(<current i node>)
cplex.AddEq(expr, 2, "degree(" + (i + 1) + ")");
}
```

Spiegato come è possibile creare un modello C# risulta comprensibile la scelta di realizzare un'opportuna funzione, chiamata **BuilModel** appartenente alla classe **Utility**, che produce il modello matematico del Commesso Viaggiatore risolubile da Cplex:

```
public static INumVar[] BuildModel(Cplex cplex, Instance instance, int
   nEdges)
```

dove:

- **cplex**: oggetto sul quale si definirà il modello matematico(funzione obbiettivo,variabili e vincoli)
- **instance**: oggetto contenente tutti i dati inerenti all'istanza del Commesso Viaggiatore fornita in ingresso dall' utente.
- **nEdges**: Parametro la cui spiegazione è rimandata al capitolo...

Passiamo infine a descrivere i metodi necessari per risolvere il modello, ottenere il costo e la soluzione ottima calcolata da Cplex.

Per risolvere il modello è sufficiente invocare, sull'oggetto di classe Cplex dove è stato definito, il metodo **Solve**:

```
cplex.Solve();
```

Una volta avviata la risoluzione, Cplex fornisce in automatico informazioni sul processo stampate nello standard output da noi definito[10]: inizialmente troviamo le impostazioni di risoluzione selezionate come ad esempio il numero di Thread .., successivamente .. .
Terminata l'operazione il costo della soluzione è memorizzato all'interno della variabile **ObjValue** di tipo **double** del solito oggetto **cplex**:

```
cplex.ObjValue;
```

Naturalmente è anche possibile conoscere il valore assunto da ogni variabile nella soluzione fornitaci da Cplex tramite il metodo **GetValues** della classe **Cplex**:

```
public virtual double GetValues(INumVar[] var)
```

dove:

- **:** rappresenta il vettore contenente tutte le variabile appartenenti al modello.

É presente anche l'analogo metodo per accedere al valore di una sola variabile **GetValue**. Il suo utilizzo è da noi altamente sconsigliato in quanto sperimentalmente è stato da verificato che ciclare quest'ultimo metodo impiega un tempo molto maggiore rispetto al semplice **getValues**.
——————————————— Qui bisogna aprire un capitolo nuovo con una breve introduzione, dire che si passa ora ad esporre i metodi utilizzati per gestire i vincoli di subtour elimination ———————————————

METODO LOOP

Il primo metodo sperimentato prende il nome di **LOOP**. ——————————————— ——- Va messa un pò di storia!!!!!! ———————————————- L'idea alla sua base è molto semplice ed è la seguente: inizialmente il modello fornito non deve contiene alcun vincolo di subtour elimination ed una volta risolto si procedere ad analizzare la soluzione ottima

[10]Se non viene modificato di default risulta essere la classica console del progetto C#

trovata. Se questa presenta dei subtour il modello viene ampliato inserendovi gli appositi vincoli per eliminarli e si procede ad una sua nuova risoluzione. Viene da se che quest'ultimo passo va ripetuto fino a quando la soluzione proposta non risulta accettabile e quindi priva di subtour[11].
É importante far notare che ogni iterazione del loop i vincoli aggiunti nella precedente sono ovviamente mantenuti.
In questo modo siamo sicuri di aver aggiunto al nostro modello solo i vincoli strettamente necessari il che non assicura che essi non siano un numero esponenziale.

Per poter implementare il metodo Loop risulta quindi evidente la necessità di sviluppare un'opportuna funzione in grado di individuare la presenza di subtour all'interno di una generica soluzione proposta e di generarne gli opportuni vincoli per eliminarli.
In letteratura esistono molteplici modi per eseguire tali operazioni, quella da noi adottata si rifà all'algoritmo di Kruskal per trovare un albero a costo minimo in un grafo connesso con lati non orientati[12].
La tecnica da noi adottata è stata quella di creare due metodi chiamati **InitCC** e **UpdateCC**: il primo serve solamente come inizializzazione per le strutture dati utilizzate dal secondo il quale, se invocato una volta per ogni lato appartenente alla soluzione attuale ne trova tutte le componenti connesse indicando anche quali lati sono a loro appartenenti. I dettagli riguardo le loro implementazioni sono visibili nella appendice di questo testo, per ora specifichiamo solamente che al termine dell'utilizzo del metodo **UpdateCC** i seguenti oggetti:

```
List<ILinearNumExpr> rcExpr = new List<ILinearNumExpr>();
List<int> bufferCoeffRC bufferCoeffRC = new List<int>();
```

risultano essere costruiti, in particolare **rcExpr** contiene le espressioni dei subtour elimination mentre invece **bufferCoeffRC** contiene il numero di lati appartenenti ad ogni subtour e quindi il termine noto delle precedenti espressioni[13].
Se all'interno di **rcExpr** è presente una espressione sola significa che la soluzione attuale è valida e quindi ottima per il problema, al contrario si deve procedere all'inserimento dei vincoli con un semplice ciclo for:

```
for (int i = 0; i < rcExpr.Count; i++)
  cplex.AddLe(rcExpr[i], bufferCoeffRC[i] - 1);
```

METODI UpdateCC e InitCC

Come già specificato nella sezione riguardo il metodo **LOOP** questi due metodi di supporto appartenenti alla classe **Utility** hanno il compito di individuare tutte le componenti connesse (che da ora in avanti abbrevieremo con **cc**) di una generica soluzione proposta.

[11] Da qui deriva il nome del metodo in quanto la soluzione consiste in un loop delle stesse operazioni
[12] Nello specifico la parte di nostro interesse è quella che impedisce la formazione di più componenti connesse
[13] Il codice assume che l'espressione di indice i presente all'interno di **rcExpr** abbia il proprio termine noto nella posizione di indice i dentro **bufferCoeffRC**

Prima di passare alla implementazione vera e propria introduciamoli ad alto livello: inizialmente si vuole assumere l'esistenza di n **cc** distinte, ognuna di esse contenente un nodo della soluzione. Questo è il compito della dalla funzione **InitCC**.
Successivamente per ogni lato della soluzione si vuole analizzare a quali **cc** sono assegnati i due nodi che lo caratterizzano. Se queste sono differenti vanno unificate in modo tale che tutti i nodi appartenenti, ad esempio , alla seconda ora appartengano tutti alla prima. Nel caso in cui invece le due **cc** coincidano significa che abbiamo trovato un subtour e il relativo vincolo di eliminazione deve essere definito. Tutte questo è invece compito del metodo **UpdateCC**.
Iniziamo quindi l'analisi del codice necessario. Per prima cosa si necessita di un vettore di interi che contenga all'indice $i - esimo$ l'identificativo della **cc** alla quale appartiene il nodo i[14]. L'inizializzazione di questo vettore viene fornita da **InitCC**:

```
public static void InitCC(int[] cc)
{
    for (int i = 0; i < cc.Length; i++)
    {
        cc[i] = i;
    }
}
```

Passiamo ora al metodo **UpdateCC** che presenta la seguente firma:

```
public static void UpdateCC(Cplex cplex, INumVar[] z,
   List<ILinearNumExpr> rcExpr, List<int> bufferCoeffRC, int[]
   relatedComponents, int i, int j)
```

dove:

- **cplex**: oggetto contenente il modello matematico corrente, eventualmente necessario per la creazione del vincolo di subtour elimination;
- **z**: vettore contenente le variabili del modello, eventualmente necessario per la creazione del vincolo di subtour elimination;
- **rcExpr**: Lista all' interno della quale vengono memorizzate le espressioni contenenti le variabili del vincolo di subtour;
- **bufferCoeffRC**: Lista contenente i termini noti dei vincoli di subtour;
- **relatedComponents**: vettore che definisce per ogni nodo del grafo la relativa componente connessa;
- **i**: Nodo che con il parametro j forma il lato (i,j);
- **j**: Nodo che con il parametro i forma il lato (i,j).

[14]Per semplicità si è deciso di identificare ogni **cc** con un valore intero univoco

La funzione UpdateCC viene invocata dal metodo Loop n volte, alla k-esima invocazione riceve in ingresso il k-esimo lato appartenente alla soluzione ottima del modello corrente. Per verificare se il lato ricevuto genera un subtour nel grafo G=(V,T*), dove T* contiene i precedenti $k-1$ lati controllati, si verifica se i vertici del lato appartengono alla medesima componente connessa. Nel caso in cui i due vertici non appartengono alla medesima componente connessa, è necessario aggiornare le componenti connesse dei vertici per l' invocazione successiva del metodo, viceversa si è individuato un subtour caratterizzato dai nodi aventi come componente connessa la medesima dei nodi i e j.
A livello implementativo si è utilizzato un array di interi chiamato relatedComponents, di dimensione pari al numero di vertici del grafo, come struttura dati necessaria per fotografare le componenti connesse del grafo G=(V,T*); relatedComponents contiene all' indice j la componente connessa del nodo j. La funzione InitCC, invocata ad ogni iterazione del metodo Loop, ha il compito di inizializzare relatedComponents associando ad ogni nodo una componente connessa diversa: in particolare si è scelto di associare al nodo j la componente connessa j. Passiamo ora ad analizzare come è stato nella pratica implementato il metodo UpdateCC, la sua intestazione è la seguente:

```
public static void UpdateCC(Cplex cplex, INumVar[] z,
    List<ILinearNumExpr> rcExpr, List<int> bufferCoeffRC,int[]
    relatedComponents, int i, int j)
```

dove:

- cplex: oggetto contenente il modello matematico corrente;
- z: vettore contenente le variabili del modello;
- rcExpr: Lista all' interno della quale vengono memorizzate le espressioni contenenti le variabili del vincolo di subtour;
- bufferCoeffRC: Lista contenente i termini noti dei vincoli di subtour;
- relatedComponents: vettore che definisce per ongi nodo del grafo la relativa componente connessa;
- i: Nodo che con il parametro j forma il lato [i,j];
- j: Nodo che con il parametro i forma il lato [i,j].

Il caso in cui non si crei un subtour è gestito molto semplicemente in questo modo:

```
if (relatedComponents[i] != relatedComponents[j])
{
    for (int k = 0; k < relatedComponents.Length; k++)
    {
        if ((k != j) && (relatedComponents[k] == relatedComponents[j]))
        {
            //Same as Kruskal
            relatedComponents[k] = relatedComponents[i];
        }
```

```
    }
    //Finally also the vallue relative to the Point i are updated
    relatedComponents[j] = relatedComponents[i];
}
```

Dove per convenzione si è deciso di inglobare la **cc** del nodo j in quella del nodo i.
Il secondo caso è invece gestito nel seguente modo:

```
else
{
    ILinearNumExpr expr = cplex.LinearNumExpr();

    //cnt stores the # of nodes of the current related components
    int cnt = 0;

    for (int h = 0; h < relatedComponents.Length; h++)
    {
        //Only nodes of the current related components are considered
        if (relatedComponents[h] == relatedComponents[i])
        {
            //Each link involving the node with index h is analized
            for (int k = h + 1; k < relatedComponents.Length; k++)
            {
                //Testing if the link is valid
                if (relatedComponents[k] == relatedComponents[i])
                {
                    //Adding the link to the expression with coefficient 1
                    expr.AddTerm(z[zPos(h, k, relatedComponents.Length)],
                       1);
                }
            }
            cnt++;
        }
    }
    //Adding the objects to the buffers
    rcExpr.Add(expr);
    bufferCoeffRC.Add(cnt);
}
```

Ripetere il metodo **UpdateCC** una ed una sola volta per ogni lato appartenente alla soluzione corrente ci assicura che le due liste **rcExpr** e **bufferCoeffRC** contengano tutti i dati per implementare i subtour elimination desiderati.

METODO LOOP CON PRIMA FASE EURISTICA

Il metodo Loop, indipendente dalla implementazione che si decide di utilizzare, vuole essere un algoritmo esatto. In altre parole è necessario assicurarsi che il risultato finale da esso prodotto sia **sempre** il migliore possibile. Come vedremo nei paragrafi successivi, sono stati pensati molteplici

algoritmi, detti euristici, che al contrario cercano solamente di avvicinarsi al risultato ottimo limitando al contempo i loro tempi di esecuzione. É infatti quest'ultimo fattore a risultare cruciale per molti problemi di programmazione lineare, a maggior ragione per quelli che, come il *commesso viaggiatore*, vogliono studiare situazioni **np-difficili**[15]. Per quanto appena esposto si è pensato di progettare una variante del metodo **Loop** caratterizzata da una fase iniziale **euristica** i cui risultati vengono poi sfruttati da una seconda fase finale **esatta**. Durante la sua progettazione ci si accorge fin da subito che, come per ogni algoritmo euristico, non esistono specifici paletti che se fissati assicurano al **100%** il raggiungimento dei propri obiettivi. Nel nostro caso ciò che desideriamo è chiaramente una fase *euristica* più veloce di quella *esatta* ma che produca anche risultati utili a quest'ultima. Abbiamo quindi deciso che, all'interno della nostra applicazione, sia l'utente stesso a poter settare alcuni parametri che rendono le due fasi più o meno differenti tra loro. In questo modo, basandosi sulle proprie esperienze e test, è possibile ottenere i risultati migliori per qualsiasi istanza del problema che si desidera risolvere. Entriamo ora nel dettaglio delle due fasi chiarendo fin da subito che tutte e due devono sempre fornire soluzioni **valide** per il problema che andiamo a risolvere. In realtà ciò che è interessante discutere riguarda quasi solamente la fase *euristica* in quanto quella *esatta* è in tutto e per tutto il classico metodo **Loop** già esposto in precedenza[16]. Esistono innumerevoli modi per rendere euristico il metodo **Loop**, possiamo suddividerli in tre grandi categorie: della prima fanno parte le tecniche che rendono la risoluzione stessa da parte di **Cplex** euristica, nella seconda ricadono i metodi che introducono nuovi vincoli al modello matematico ed infine la terza categoria è una semplice combinazione delle due precedenti. All'interno del nostro progetto sono state sviluppate tre varianti della fase *euristica*, una per ogni categoria appena esposta:

- **prima categoria**: durante la risoluzione del problema attraverso la tecnica del **Branch&Cut**, Cplex, oltre al banale calcolo della soluzione ottima per ogni nodo dell'albero decisionale, sfrutta internamente algoritmi euristici per agevolare il processo. Durante quest'ultimo si hanno quindi a disposizione due parametri, il primo è il costo della soluzione euristica migliore (C_{eu}), mentre il secondo è il classico costo **lower bound**[17] (L_b). É importate far notare che questi valori mutano mano a mano che si procede alla costruzione dell'albero decisione, in particolare C_{eu} cresce mentre L_b scende fino a che, teoricamente, non coincidano. La distanza **relativa** tra i due valori viene costantemente monitorata da Cplex e, se questa scende sotto la soglia minima del suo parametro interno **EpGap**, il processo di risoluzione viene considerato terminato e la miglior soluzione valida trovata viene restituita. Maggiori dettagli riguarda *EpGap* sono forniti nel paragrafo ad esso dedicato LINK!!!!!!!, per ora ci basta dire che se di default è settato ad un valore vicino allo 0, la sua variazione è proprio ciò che viene utilizzato nel nostro programma all'interno della fase *euristica* del metodo *Loop*. Risulta estremamente intuitivo e facilmente verificabile che per una specifica istanza non è possibile a priori determinare quanto velocemente verrà raggiunto un certo gap durante la fase di *Branch&Cut*. Eǵuindi consigliabile eseguire inizialmente il normale metodo **Loop**, analizzare l'output for-

[15] In letteratura è noto infatti che i problemi appartenenti a quest'ultima categoria sono caratterizzati da tempi di risoluzione, al caso peggiore, esponenziali rispetto al numero di variabili che li caratterizzano. Nel caso in cui siano definiti nel tipico linguaggio della programmazione lineare, la caratteristica appena esposta è riscontrabile da un numero di vincoli anch'esso esponenziale.

[16] Ciò che varia è solamente che il modello matematico iniziale dato in pasto alla fase *esatta* presenta già dei vincoli di **subtour elimination** individuati dalla fase *euristica* dove però il modello matematico di partenza presenta già alcuni vincoli di *subtour elimination*.

[17] Nel nostro caso sarà il costo della migliore soluzione intera trovato.

nito da Cplex durante la risoluzione ed osservare per quale gap soluzioni successive dell'albero decisionale non producono più miglioramenti sostanziali e
o richiedono tempi di esecuzione eccessivi. Per quanto detto il valore di **EpGap** viene lasciato a discrezione dell'utente;

- **seconda categoria**: come precedentemente indicato ad inizio paragrafo, i tempi di risoluzione risultano esponenziali rispetto al numero di variabili che caratterizzano il problema in questione. Limitarne il numero ha pertanto un impatto notevole nella risoluzione del modello matematico ed a tale scopo è possibile decidere di settare a priori il valore assunto da alcune variabili. Tanto migliore risulta essere la previsione così introdotta, migliori saranno i tempi di risoluzione ottenibili sia per la fase *euristica* che quella *esatta* del metodo *Loop*. La scelta da noi effettuata è di lasciare selezionabili per ogni nodo gli m collegamenti a costo minore che lo vedono come un loro vertice. Il numero m è lasciato selezionabile dall'utente e da risultati sperimentali non è consigliabile che si discosti troppo dal valore 10;

- **terza categoria**: vengono semplicemente combinate le due precedenti;

Nei due paragrafi successivi vengono mostrati alcuni dettagli, tra cui quelli realizzativi, per l'utilizzo delle varianti euristiche esposte del metodo **Loop**.

EpGap

EpGap risulta essere un parametro interno di Cplex, il cui valore di default è $1e^{-06}$. Indicando con **bestnode** il miglior valore della funzione obiettivo calcolata ad un nodo dell'albero decisionale attraverso metodi euristici propri di Cplex, e con **bestInteger** il costo della miglior soluzione intera trovata fino a quel momento, ossia il costo dell'incumbent, qualora la seguente quantità:

$$|bestNode - bestInteger|/(1e - 10 + |bestInteger|)$$

risulti inferiore ad *EpGap* il solver si arresta.
Il settaggio del parametro in questione è molto semplice, è sufficiente invocare il metodo **SetParam** sull'oggetto della classe **Cplex** che si sta utilizzando come riportato di seguito:

```
cplex.SetParam(Cplex.DoubleParam.EpGap, newValue);
```

Essendo *EpGap* un valore di distanza **relativo** e non **assoluto**, la variabile **newValue** deve essere compresa tra i valori 0[18] e 1[19].
Completato il settaggio di *EpGap* è sufficiente procedere con il normale metodo risolutivo **Loop**. Al termine di quest'ultimo otteniamo una soluzione euristica ma soprattutto un insieme di vincoli di *subtour elimination*. Entriamo quindi nella fase *esatta* dell'algoritmo riportando al valore di default *EpGap* e ripetiamo il metodo risolutivo **Loop** sul modello matematico originale ampliato dai nuovi vincoli appena citati.

[18] La risoluzione termina fornendo sempre la soluzione ottima.
[19] La risoluzione termina alla prima soluzione ammissibile individuata.

Figura X: Output

TITOLO DA CAMBIARE

Nel paragrafo LINK!!!!!!!!!!!!!!!, è stato presentato il metodo **BuildModel** della classe **Utility** il cui compito è di creare il modello matematico, privo dei vincoli di subtour elimination, risolubile da Cplex. Durante la presentazione della sua firma, era stato lasciato in sospeso l'utilizzo del parametro *nEdges* in quanto allora del tutto prematura. Come specificato nel paragrafo LINK!!!!!!!!!! una variante euristica del metodo **Loop**, prevede di utilizzare solamente un numero massimo m di lati incidenti in ogni nodo del problema in questione del commesso viaggiatore. Tale parametro è inoltre richiesto all'utente per i motivi già specificati e viene comunicato al metodo **BuildModel** proprio grazie alla variabile di ingresso **nEdges**[20].
A livello implementativo, per rendere inutilizzabili certi collegamenti, è sufficiente non inserirli nel modello matematico oppure farlo ma fissandone sia il **lower** che l'**upper bound** a 0. Quest'ultima opzione è quella più comoda da utilizzare in quanto nella successiva fase *esatta* dell'algoritmo il modello matematico deve comunque aver disponibili al suo interno tutte le variabili. L'individuazione algoritmica di quali collegamenti debbano essere abilitati risulta l'operazione più complessa da un punto di vista computazione. Come vedremo nel corso di questa tesi, diversi algoritmi necessitano di conoscere per ogni nodo l'ordinamento completo dei lati ad esso incidenti basato sul loro costo e per tanto si è definita una unica funzione **BuildSL** adibita a tale scopo[21]. la quale I dettagli riguardanti al metodo **BuildSL** sono riportati al seguente paragrafo della appendice LINK!!!!!!!!!!!!!!!, in questo contesto ci basta indicare che essa restituisce una lista di vettori di interi, chiamata **listArray**, in cui $i-esimo$ elemento contiene, in ordine crescente, la sequenza dei restanti $n-1$ vertici basandosi sulla loro distanza rispetto al nodo i. In altre parole, una volta invocato *BuildSL*, i lati il cui **upper bound** deve essere posto pari ad 1 sono individuabili dagli estremi

$$0, (listArray[0])[0]$$

,

$$0, (listArray[0])[1]$$

, ...,

$$0, (listArray[0])[m]$$

, ...,

$$i, (listArray[i])[0]$$

, ...,

$$i, (listArray[i])[m]$$

, ...,

$$n-1, (listArray[n-1])[0]$$

[20]Come dettaglio implementativo specifichiamo inoltre che per convenzione si è deciso di settare $nEdges = -1$ nel caso in cui si voglia coistruire un modello con tutti i lati possibili abilitati

[21]In realtà il costo computazionale per trovare gli m lati meno costosi incidenti in un nodo, ripetuto per tutti gli n nodi disponibili, è il medesimo del metodo *BuildSL* e quindi l'utilizzo di quest'ultimo non porta alcuno svantaggio.

, ...,

$$n-1, (listArray[n-1])[m]$$

. Di seguito è riportato per completezza il codice all'interno della funzione **BuildModel** che nel caso di $nEdges \geqslant 1$ sfrutta le informazioni presenti in *listArray* per il settaggio delle variabili del modello matematico:

```
if (nEdges > 0)
{
    List<int>[] listArray = BuildSLComplete(instance);

    for (int i = 0; i < instance.NNodes; i++)
    {
        for (int j = 0; j < nEdges; j++)
        {
            int position = xPos(i, listArray[i][j], instance.NNodes);

            x[position].UB = 1;
        }
    }
}
```

Dove chiaramente in precedenza era stato necessario inizializzare tutte le variabili contenute in **x** come:

```
ILinearNumExpr expr = cplex.LinearNumExpr();


//Populating objective function
for (int i = 0; i < instance.NNodes; i++)
{
    for (int j = i + 1; j < instance.NNodes; j++)
    {
        ////xPos return the correct position where to store the variable
           corresponding to the actual link (i,i)
        int position = xPos(i, j, instance.NNodes);

        if (nEdges > 0)
            x[position] = cplex.NumVar(0, 0, NumVarType.Bool, "x(" + (i +
               1) + "," + (j + 1) + ")");
        else
            ...

        expr.AddTerm(x[position], Point.Distance(instance.Coord[i],
           instance.Coord[j], instance.EdgeType));
        }
    }
}
```

Terminata la costruzione del modello matematico si procede alla sua risoluzione attraverso il classico metodo **Loop**. Al suo termine, otteniamo una soluzione euristica e soprattutto una lista di vincoli di

subtour elimination. Per passare alla risoluzione esatta del problema in analisi, manteniamo questi ultimi e settiamo l'**upper bound** di ogni variabile ad 1. Questa ultima operazione è ottenibile invocando il metdo **ResetVariables** appartenente alla classe Utility:

```
public static void ResetVariables(INumVar[] x)
{
    for (int i = 0; i < x.Length; i++)
        x[i].UB = 1;
}
```

Dove chiaramente **x** è il vettore contenente i riferimenti alle variabili utilizzate dal modello matematico da noi definito.

CALLBACK

In questa sezione esponiamo una tecnica alternativa per l'inserimento delle espressioni di subtour elimination all'interno di un sistema. Ciò che varia rispetto al metodo **Loop** presentato in precedenza è il **momento** in cui tali espressioni vengono definite.
L'idea è di sfruttare il fatto che Cplex come metodo di risoluzione per i problemi di **PLI** utilizza la tecnica del **Branch&Cut**[22]. Viene inoltre offerta la possibilità di conoscere la soluzione trovata, sia essa frazionaria o intera, per ogni nodo dell'albero decisione ma soprattuto la possibilità di ampliare il modello matematico come meglio crediamo.
Quello che vogliamo fare risulta a questo punto molto chiaro: se alla analisi della soluzione di un nodo sono presenti subtour il modello matematico deve essere modificato per eliminarli.
A livello pratico Cplex permette l'implementazione distinta di callback che vengono eseguite nel momento in cui viene trovata una soluzione intera oppure frazionaria[23]: nel primo caso è necessario implementare una **"lazy constraint callback"** mentre nel secondo caso una **"user cut callback"**.
Da notare che in realtà solo soluzioni valide per i criteri di fathoming possono far scattare una callback, ciò non avviene ad esempio se il valore della soluzione di un nodo risulta maggiore rispetto a quello dell'incumbent[24].
In generale i tagli possono essere definiti **locali** o **globali**: mentre i primi hanno validità esclusiva all'interno del sottoalbero avente come radice il nodo per la quale sono stati generati, i secondi hanno validità per tutti i nodi dell'albero decisionale e vengono memorizzati in una struttura globale detta **pool di tagli**. Cplex inoltre fornisce la possibilità di definire un taglio **purgeable** o meno: nel primo caso significa che può essere rimosso in un secondo momento poiché ritenuto inefficace. Durante il proseguo della tesi i tagli saranno da considerarsi sempre globali e non purgeable.

[22] Da ora in avanti sarà abbreviato con la sigla **B&C**

[23] In informatica una callback è una funzione definita dall'utente che viene eseguita in automatico dal sistema ogniqualvolta scatta un particolare evento.

[24] Per incumbent si intende il valore migliore trovato fino a questo momento relativo ad un soluzione accettabile.

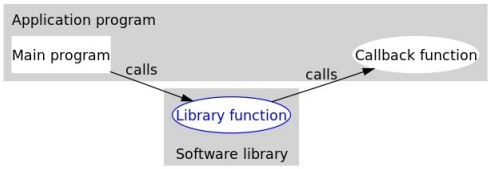

Figura 3: Soluzione frazionaria

Prima di procedere con l'esposizione dei dattagli riguardanti l'implementare di tali procedure, è possibile effettuare le seguenti considerazioni:

- La soluzione ottima fornita da Cplex risulta per costruzione priva di subtour: non è quindi più necessario, al contrario del metodo **Loop**, lanciare molteplici risoluzioni. A livello pratico è sufficiente invocare solo una volta il metodo **Cplex.Solve()**.
- Maggiore è il numero di vincoli che andiamo ad inserire, maggiore diventa il tempo di risoluzione per i vari nodi successivi dell'albero decisionale.
- Il numero di nodi che forniscono soluzioni frazionarie risulta di molto superiore rispetto a quelli con soluzione intera. Tenendo conto di quanto detto al punto precedente, non è quindi saggio andare ad analizzare tutte le soluzioni frazionarie ma solo una loro minima percentuale. Senza questo accorgimento si andrebbe inevitabilmente ad inserire innumerevoli vincoli superflui per l'ottenenimento della soluzione ottima con conseguenti tempi di risoluzione eccessivamente elevati.
- I moderni processori hanno a disposizione molteplici core sia reali che virtuali e quindi sfruttare tecniche di multi-threading. In particolare Cplex permette di settare il numero di thread utilizzabili, in modo tale che ognuno di essi si occupi dalla risoluzione di un nodo dell'albero decisionale: in questo modo, in linea teorica, si drovrebbe ottenere un boost delle prestazioni con conseguente riduzione dei tempi di calcolo. D'altro canto, come per qualsiasi applicazione informatica, l'utilizzo del multi-threading risulta rischioso in quanto l'accesso contemporaneo ai medesimi dati può portare ad una loro inconsistenza. Nel nostro caso può capitare che più callback eseguite contemporaneamente vadano a modificare variabili condivise andando così incontro ad eccezioni o anomalie tali da non garantire più la correttezza della soluzione prodotta da Cplex.
  Per evitare queste problematiche, i progettisti di Cplex hanno preferito settare il numero di thread al valore **1** dopo l' installazione di una callback. É quindi nostro compito modificare tale parametro così da renderlo pari al numero di processori virtuali a nostra disposizione e di conseguenza assicurarci che le callback risultino **thread-safe**. Maggiori dettagli sono riportati nei successivi paragrafi.

LAZYCONSTRAINT CALLBACK C#

Per poter utilizzare una **lazy constraint callback** in **C#** Cplex fornisce all'interno delle proprie librerie la classe astratta **LazyConstraintCallback** che a sua volta estende **ControlCallback**. É

quindi necessario creare una propria classe che estenda quest'ultima, nel nostro caso è stato deciso di chiamarla **TSPLazyConsCallback**, in questo modo è necessario definire al suo interno il metodo **Main** che verrà invocato automaticamente dal sistema ogniqualvolta scatta la callback[25].
Una volta terminato questo processo l'installazione della callback viene eseguita nel seguente modo:

```
cplex.Use(new TSPLazyConsCallback(...));
```

Dove, come al solito, cplex è l'instanza della classe **Cplex** sulla quale definiamo il modello matematico privo dei vincoli di subtour elimination.
Mostriamo ora la firma del costruttore della classe **TSPLazyConsCallback** riportando una breve descrizione dei parametri di ingresso:

```
public TSPLazyConsCallback(Cplex cplex, INumVar[] z, Instance instance,
    Process process, bool BlockPrint)
```

- **cplex**: necessario per l'individuazione dei vincoli di subtour elimination, contiene i dati del modello matematico utilizzato;
- **z**: identico al punto precedente, contiene i riferimenti alla variabili del modello matematico;
- **instance**: necessario nel caso in cui si desideri stampare attraverso GNUPlot le soluzioni intere che hanno fatto scattare la callback;
- **process**: identico al punto precedente;
- **BlockPrint**: è il parametro booleano che determina se procedere o meno con le stampe delle soluzioni intere (se **true** si procede con la stampa);

Come accennato nel paragrafo precedente, l'installazione di una callback setta automaticamente il numero di thread ad uno. Per modificare tale valore, ponendolo pari al numero logico di cores messi a disposizione dal processo in uso, è sufficiente eseguire la seguente riga di codice:

```
cplex.SetParam(Cplex.Param.Threads, cplex.GetNumCores());
```

Come sarà possibile vedere più avanti, la tecnica da noi utilizzata per l'individuazione di subtour risulta thread-safe in quanto non vengono utilizzate variabili condivise da più threads se non nella sola modalità di lettura. L'aggiunta di eventuali tagli, d'altro canto, viene gestita in modo automatico da Cplex assicurandoci, anche in questo caso, una procedura thread-safe. Un discorso apparte deve invece essere fatto nel caso in cui la variabile **BlockPrint** descritta in precedenza sia stata posta a **true**. Come era logico aspettarsi, abbiamo verificato che spesso la procedura di stampa attraverso GNUPlot di una qualsiasi soluzione richiede un tempo di esecuzione maggiore rispetto la frequenza con cui le callback sono effettuate. Ricordando inoltre che, il metodo da noi utilizzato per comunicare a GNUPlot le coordinate cartesiane dei punti del grafo cartesiano prevede la scrittura di queste ultime in un apposito file di testo, è stato necessario individuare un modo

[25] Rocordiamo che tutti i metodi astratti presenti all'interno di una classe astratta devono essere obbligatoriamente definiti da tutte le classi che estendono quest'ultima

per evitare problemi riguardanti il multi-threading: utilizzare sempre lo stesso file di testo causa infatti errori nella stampa dei grafi, in particolare la lettura delle coordinate da parte di GNUPlot risulta troppo lenta e durante questo processo più thread rischiano di modificare il file con le proprie coordinate.
Per evitare questo problema è stato quindi necessario stampare le coordinate prodotte dai vari nodi dell'albero decisionale in differenti files. A tal proposito la tecnica da noi scelta è stata quella di inserire nel nome di questi ultimi anche l'id numerico del nodo a loro associato che viene fornito dirretamente da Cplex:

```
string nodeId = GetNodeId().ToString();

...

string fileName = instance.InputFile + "_" + nodeId;
```

La funzione **GetNodeId** risulta disponibile in quanto ereditata dalla classe **ControlCallback**.
Dopo i dovuti chiarimenti riguardanti il multi-threading passiamo ora a descrivere nei dettagli come è stata realizzato il metodo Main della classe TSPLazyConsCallback. Prima di tutto per verificare l'eventuale presenza di subtour bisogna naturalmente accedere alla soluzione fornita per il nodo dell'albero decisionale in questione: a tal fine si possono utilizzare i metodi **GetValues** e **GetValue**, ereditati entrambi dalla classe ControlCallback, che ricevono in input rispettivamente un vettore di riferimenti per variabili del modello matematico e un singolo riferimento ad una variabile di quest'ultimo. Dopo pochi test ci si accorge immediatamente che invocare più volte il metodo **GetValue**, ad esempio dentro un ciclo for, risulta molto più oneroso in termini temporali rispetto una singola evocazione del metodo **GetValues**: si può quindi dedurre che è molto più dispendioso effettuare multiplici accessi all'interfaccia fornita da Cplex rispetto alla quantità di dati che ad essa richiediamo.
Per quanto appena detto la nostra scelta è ricaduta nel metodo **GetValues** che restituisce un vettori di **double** contenente il valore dell variabili (il cui riferimento è ricevuto come ingresso) nella soluzione corrente del modello matematico. Da notare che anche in questo caso anche se ci aspettiamo tutti valori interi, in particolare pari a 0 oppure 1, è possibile che ci siano in realtà discostamenti infinitesimi pertanto quando controlliamo il valore di una variabile verifichiamo semplicemente se è maggiore o minore del valore 0,5.
I metodi utilizzati per l'individuazione di eventuali subtour e la eventuale stampa del grafo attraverso GNUPlot sono identici a quelli utilizzati per il metodo **Loop** pertanto non sono qui riportarti.
Una volta ottenute tutte le informazioni riguardanti i subtour, al contrario di quanto viene fatto nel metodo Loop non è richiesto di ampliare direttamente il modello con nuovi vincoli ma, come era già stato accennato in precedenza, deve essere popolato il pool di tagli associato al modello matematico:

```
IRange[] cuts = new IRange[ccExprLC.Count];

//if cuts.Length is 1 the graph has only one tour then cuts aren't needed
if (cuts.Length > 1)
{
    for (int i = 0; i < cuts.Length; i++)
    {
```

```
            cuts[i] = cplex.Le(ccExprLC[i], bufferCoeffCCLC[i] - 1);
            Add(cuts[i], 1);
        }
    }
```

Dove:

- **cuts**: è un vettore di **IRange** che sono la struttura di dati base fornita da Cplex per memorizzare espressioni lineari;
- **ccExprLC[i]**: analogamente per quanto avviene nel metodo Loop, contiene i dati delle variabili dell'i-esimo taglio memorizzati come **ILinearNumExpr** (struttura dati fornita da Cplex);
- **bufferCoeffCCLC[i]**: analogamente per quanto avviene nel metodo Loop, contiene il numero di variabili che definiscono l'i-esimo taglio;
- **cplex.Le**: è la funzione definita da Cplex che restituisce una espressione lineare che impone le variabili, ricevute come primo parametro, minori oppure uguali del secondo parametro ricevuto;
- **Add**: è la funzione eraditata dalla classe **LazyConstraintCallback** che permette di aggiungere un taglio **globale**. Come parametri riceve quindi il taglio stesso ed un valore intero che indica a Cplex come debba gestire quest'ultimo:
  - **0**: il taglio è aggiunto al pool in maniera permanente;
  - **1**: il taglio è definito come **purgeable** quindi eliminabile nel caso in cui non risulti più efficiente;
  - **2**: il taglio viene trattato come se fosse stato generato da Cplex, quindi ad esempio prima di essere aggiunto al pool viene analizzata la sua efficacia e di conseguenza l'operazione va quindi a buon fino o meno;

CONCORDE

L'utilizzo di una **user cut callback** risulta molto più complicato rispetto a quanto appena visto per la **lazy constraint callback**: dato che tali callback scattano nel momento in cui viene trovata una soluzione frazionaria, l'individuazione di eventuali subtour non può essere eseguita con le tecniche esposte fino ad ora.
Per effetturare tale operazione si necessita di un separatore che, ricevendo in ingresso la soluzione x* con almeno una componente frazionaria, fornisca in uscita un insieme $S \subsetneq V$, $|S| \geq 2$ tale per cui:

$$\sum_{(i,j)\in E(S)} x^*_{i,j} \nleqslant \left|\widehat{S}\right| - 1 \tag{6}$$

Tale sottoinsieme non è facilmente individuabile come nel caso di soluzione intere in cui era sufficiente individuare le componenti connesse. Per esempio in Figura 3 si è riportato il supporto di una soluzione x* frazionaria ove i lati colorati di rosso, blu e grigio indicano che la corrispondente variabile assumme rispettivamente i valori 1, 0.5 , 1.5.

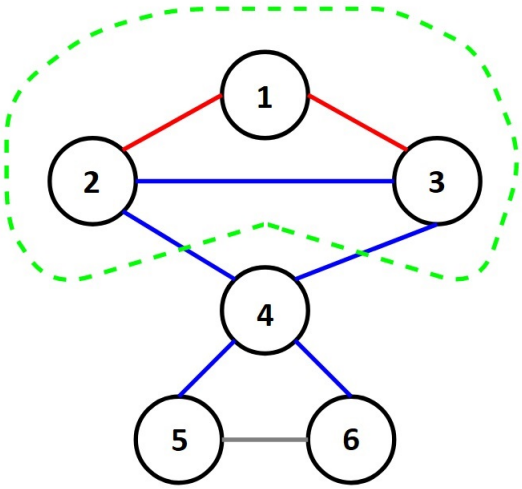

Figura 3: Soluzione frazionaria

Tale grafo risulta connesso; tuttavia è presente un sottoinsieme S = 1,2,3 per cui vale (6).
Una seconda formulazione equivalente alla (6) risulta essere la seguente:

$$\sum_{(i,j)\in\delta(S)} x^*_{i,j} \ngeq 2 \tag{7}$$

Si osserva che il primo membro di (7) può essere visto come la capacità di una sezione di una rete di flusso se si interpretano le x* come le capacità della rete. E' possibile calcolare una sezione di capacità minima risolvendo un problema di max flow che sappiamo essere di programmazione lineare e quindi risolubile attraverso un algoritmo polinomiale.
Poichè però la sezione di capacità minima dipende dal nodo sorgente s e dal nodo di destinazione t, si devono in realtà risolvere **n-1** problemi di max flow: per tale ragione è stato preferito utilizzare una porzione del software **Concorde** che offre, attraverso le proprie librerie, la possibilità di risolvere tale problema con tempi di esecuzione molto brevi ed allo stesso tempo di alleggerire il nostro carico di lavoro che in ogni caso non avrebbe prodotto risultati migliori.
Concorde è un software, sviluppano in linguaggio **C** da David Applegate, Robert E. Bixby, Vašek Chvátal, e William J. Cook, specializzato nella risoluzione ottimizzata di istanze del problema del commesso viaggiatore. Per fini accademici la distribuzione e l'utilizzo è fornita in modo gratuito e le librerei possono essere scaricate direttamente dal seguente indirizzo:

http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm

Come accennato poco fa il linguaggio utilizzato da Concorde non è **C#** bensì **C** pertanto una implementazione diretta del software non è possibile. La soluzione da noi adottata è la seguente: abbiamo creato un nuovo progetto Visual Studio in linguaggio **C**/**C++** ed al suo interno abbiamo creato un codice che soddisfacesse unicamente alla funzionalità ambedue i tipi di callback proposti

interfacciandosi alle librerie di Concorde per trovare i vincoli di subtour elimination ed aggiungerli al modello matematico. Successivamente il tutto è stato impachettato all'interno di una **DLL** compatibile con il linguaggio **C#**. Il processo di creazione della DLL è stato spiegato nel seguente paragrafo LINK!!!!!!!!!!!!!!!!!!!, d'ora in avanti quindi ci focalizzeremo unicamente nel contenuto della libreria dinamica da noi creata.

Per poter utilizzare Concorde in ambiente Windows è necessario importare ogni singolo file .c e .h che appartiene alla distribuzione: nel nostro caso però solo una minima parte delle sue funzionalità è di nostro interesse e pertanto solamente i seguenti file sono stati da noi utilizzati:

- **allocrus.c**
- **connect.c**
- **cut_st.c**
- **mincut.c**
- **shrink.c**
- **sortrus.c**
- **urandom.c**
- **cut.h**
- **machdefs.h**
- **macrorus.h**
- **util.h**
- **end**

Dove affinché il programma compili correttamente, è necessario effettuare le seguenti modifiche:

- All'interno dei file **allocrus.c** e **util.h** è necessario importare tramite il comando **import** l'header **malloc.h**.
- All' interno del file **machdefs.h** è necessario eliminare l'inclusione di **config.h**.

LAZYCONSTRAINTCALLBACK IN C

In questa mostreremo solamente i dettagli per l'installazione e l'utilizzo delle **lazyconstraint callback** in linguaggio **C** in quanto un discorso più ampio è stato precedentemente in LINK!!!!!!!!!. L'installazione di questo tipo di callback avviene attraverso l'invocazione della routine **CPXsetlazyconstraintcallbackfunc** la cui firma è:

```
CPXsetlazyconstraintcallbackfunc(CPXENVptr
    env,int(*)(CALLBACK\_CUT\_ARGS) lazyconcallback, void * cbhandle)
```

Dove:

- **env**: Espressione contenente una combinazione lineare delle variabili del vincolo; SIAMO SICURI!!!!???!??!
- **lazyconcallback**: Rappresenta il nome, attribuito dal programmatore esterno, della funzione che viene invocata da Cplex qualora la soluzione del rilassamento continuo di un nodo abbia valore intero ed inferiore all'incumbent. Nel nostro caso il nome assunto è **myLazyCallBack**;
- **cbhandle**: Puntatore ad una struttura dati passata dall'utente contenente le informazioni che devono essere visibili all'interno della callback *myLazyCallBack*. Come parametro si è passato il puntatore all'istanza;

In modo del tutto analogo a quanto fatto per **C#**, per impostare il numero di thread pari ai core virtuali offerti dalla macchina in utilizzo si sono utilizzati i metodi **CPXsetintparam** e **CPXgetnumcores**:

```
CPXgetnumcores(env, int * nCore);
CPXsetintparam(env, CPXPARAM\_Threads, nCore);
```

Passiamo ora a descrivere la funzione *myLazyCallBack* che identifica i subtour ed aggiunge i relativi vincoli al modello, ricordando che la sua firma deve rispettare specifici parametri definiti da Cplex stesso[26]:

```
static int CPXPUBLIC myLazyCallBack(CPXCENVptr env, void *cbdata, int
    wherefrom, void *cbhandle, int *useraction\_p)
```

Dove:

- **env**: rappresenta l'istanza dell' enviroment con il quale stiamo lavorando. DIVERSO DA SOPRA!!!!!!!!
- **cbdata**: come accennato poco fa questo parametro è quello specificato dall'utente durante l'installazione della callback, non essendo noto a priori il tipo di dato che l'utente desidera ricevere si utilizza **void**;
- **wherefrom**: definisce da che punto del *B&C* è stata invocata la funzione, ai fini pratici tale parametro, per la lazy callback è risultato irrilevante;
- **cbhandle**: puntatore a dati privati utilizzato da Cplex;
- **useraction_p**: puntatore ad un intero utilizzato dall'utente per comunicare a Cplex diverse informazioni. Tale parametro può assumere i seguenti tre valori:

[26] Essendo tali funzioni invocate automaticamente da Cplex i tipi di parametri che esse ricevono sono stati definiti a priori e non risultano modificabili

- **0**: avente come costante simbolica CPX_CALLBACK_DEFAULT comunica a Cplex che fino a quel punto la callback non ha aggiunto tagli al modello;
- **1**: avente come costante simbolica CPX_CALLBACK_FAI impone a Cplex di uscire dall'ottimizzazione;
- **2**: avente come costante simbolica CPX_CALLBACK_SET comunica a Cplex che sono stati aggiunti tagli;

La prima operazione da compiere consiste nell effettuare un cast al puntatore **cbhandle** il cui tipo è noto solo al programmatore che ha installato la callback: nel nostro caso il puntatore è di tipo **instance** per cui:

```
instance *inst = (instance*)cbhandle;
```

Successivamente è necessario assegnare al parametro ***useraction_p** il valore **CPX_CALLBACK_DEFAULT**
Per ottenere la soluzione del rilassamento continuo è necessario utilizzare il metodo **CPXgetcallbacknodex** avente come intestazione:

```
int CPXgetcallbacknodex(CPXCENVptr env, void * cbdata, int wherefrom,
    double * x, int begin, int end)
```

Dove:

- Per quanto riguarda env, cbdata, wherefrom vale la descrizione vista per il metodo myLazyCallBack;
- **x**: array che al termine del metodo conterrà la soluzione intera del rilassamento continuo;
- **begin**: indica l'indice della prima variabile di cui si vuole conoscere il valore;
- **end**: indica l'indice dell'ultima variabile di cui si vuole conoscere il valore;

Nel nostro caso dato che vogliamo conoscere tutte le variabili, assegniamo i valori 0 e $[n*(n-1)/2]\check{}1$ rispettivamente ai parametri **begin** ed **end**.
All'atto dell'invocazione del metodo *CPXgetcallbacknodex* non viene passato come parametro l'array **bestLb** contenuto in *inst* ma viene creato un opportuno array chiamato **xstar**:

```
double *xstar = (double*)malloc(inst->nCols * sizeof(double));
```

Questa operazione risulta necessaria al fine di realizzare un codice che risulti thread-safety: poiché *inst* è un puntatore accessibile da tutti i thread esiste il rischio di accessi multipli sia in modalità di lettura popolando così *bestLb* con valori appartenti a soluzioni differenti. A questo punto entra in gioco Concorde per l'individuazione e l'introduzione dei vincoli di subtour elination, dato che il metodo da utilizzare è il medesimo che vedremo per le *usercut callback* rimandiamo al paragrafo seguente per maggiori dettagli. Una volta completato tale passaggio non rimane altro che impostare il parametro ***useraction_p** al valore **CPX_CALLBACK_SET** al fine di comunicare a Cplex che sono stati aggiunti tagli.

USERCUT CALLBACK IN C

Come già anticipato nei precedenti paragrafi questo tipo di callback sono utilizzate per gestire soluzioni frazionarie ottenute per i vari nodidell'albero decisionale durante una risoluzione di tipo *B&C* per problemi di programmazione lineare da parte di Cplex. A livello concettuale sono del tutto simili a quanto visto nel paragrafo precedente per le *lazy callback* in linguaggio **C**, è raccomandata una lettura del paragrafo precedente a loro dedicato in quanto di seguito saranno esposti estensivamente solamente i dettagli riguardanti la gestione dei tagli.[27].
L'installazione delle callback avviene tramite la funzione **CPXsetusercutcallbackfunc**:

```
int CPXsetusercutcallbackfunc (CPXENVptr env, int(*)(CALLBACK\_CUT\_ARGS)
    lazyconcallback, void * cbhandle)
```

La funzione invocata da Cplex in corrispondenza di una soluzione frazionaria è stata da noi chiamata **myUserCutCallBack** la cui firma, che anche in questo caso viene imposta dai progettisti di Cplex, risulta essere:

```
int CPXPUBLIC myUserCutCallBack(CPXCENVptr env, void *cbdata, int
    wherefrom, void *cbhandle, int *useraction\_p)
```

Cplex, una volta calcolata una soluzione frazionara, genera in automatico dei propri tagli[28]. Quando il parametro *wherefrom* risulta pari a **CPX_CALLBACK_MIP_CUT_LAST** significa che l'iterazione successiva da parte di Cplex consisterebbe nell'operazione di branching sul nodo in questione: solo in questa condizione risulta conveniente generare i propri vincoli caratteristici del problema che si sta risolvendo. Qualora il parametro *wherefrom* assuma invece altri valori, si effettua una semplice **return 0** senza eseguire alcuna operazione, altrimenti come già discusso per le lazy, è necessario recuperare il puntate all'istanza. Riprendendo quanto detto nel paragrafo LINK PARAGRAFO CALLBACK||||| è sconsigliato aggiungere *manualmente* ad ogni nodo dell'albero decisionale dei tagli in quanto il loro numero complessivo risulterebbe troppo elevato andando quindi a **peggiorare** le prestazioni di Cplex. Per tale ragione, dopo alcuni test e secondo le linee guida discusse durante il corso, si è deciso che solamente con una probabilità del 10% la *usercut callback* da noi definita entra in gioco. Dato che l'id numerico assegnato ai nodi dell'albero decisionale, ottenuto attraverso al funzione **CPXgetcallbacknodeinfo**, non ha alcuna relazione diretta alla probabilità che venga generata una soluzione intera oppure frazionaria, è sufficiente effettuare una operazione di modulo dieci a tale valore: nel caso in cui il risultato sia pari a zero si procede con il calcolo dei tagli. Successivamente, invocando la nota funzione **CPXgetcallbacknodex** si ottiene la soluzione frazionaria. Prima di procedere con la parte principale di questo metodo, per ragioni di chiarezza riportiamo il codice che esegue quanto finora descritto:

```
*useraction\_p = CPX\_CALLBACK\_DEFAULT;

int nodecount = 0;
```

[27]Notiamo in realtà che l'implementazione delle *usercut callback* avviene sempre in concomitanza all'implementazione delle *lazy callback* per tanto il settaggio del numero di thread è necessario solamente una volta
[28]Ad esempio taglio di *Gomory*

```
CPXgetcallbacknodeinfo(env, cbdata, wherefrom, 0,
   CPX\_CALLBACK\_INFO\_NODE\_DEPTH, &nodecount);

if (wherefrom == CPX\_CALLBACK\_MIP\_CUT\_LAST)
{
    instance *inst = (instance*)cbhandle;

    double *xstar = (double*)malloc(inst->nCols * sizeof(double));

     if ((nodecount % 10) != 0)
        return 0;

    if (CPXgetcallbacknodex(env, cbdata, wherefrom, xstar, 0, inst->nCols
       - 1))
    {
        free(comps);
        free(compscount);
        free(xstar);
        free(elist);
        return 1;
    }
}
```

Da questo momento inizieremo ad utilizzare le funzionalità offerta da *Concorde*, tutte le funzioni il cuo nome inizia con la sigla **"CC"** sono importate da quest'ultimo. L'aggiunta di eventuali vincoli di *subtour elimination* avviene tramite l'invocazione in un primo momento della funzione **CCcut_connect_components** la quale identifica le componenti connesse della soluzione ricevuta come parametro indipendentemente dal fatto che sia intera o frazionaria.
Di seguito sono riportati nel dettaglio tutti i parametri che tale funzione vuole ricevere in ingresso, si osserva che mentre i primi 4 costituiscono l'effettivo input della funzione i rimanenti 3 sono in realtà settata al suo interno e quindi possono essere visti come parametri di output:

- **ncount**: rappresenta il numero di nodi del grafo;
- **econut**: rappresenta il numero di lati del grafo, ossia ncount*(ncount-1)/2;
- ***elist**: vettore di dimensione $2 * econut$, contiene al suo interno tutti i lati del grafo caratterizzati dai nodi sul quale esso incide memorizzati in locazioni consecutive dell'array, è stato da noi realizzato nel seguente modo:

```
    int loader = 0;
    for (int i = 0; i < inst->nNodes;  i++)
    {
        for (int j = i + 1; j < inst->nNodes;  j++)
        {
            elist[loader++] = i;
            elist[loader++] = j;
        }
    }
```

- ***x**: soluzione per la quale si desiderano individuare le componenti connesse;
- ***ncomp**: rappresenta il numero di componenti connesse;
- ****compscount**: vettore di vettori contenenti il numero di nodi per ciascuna componente connessa, è strutturato in modo che compscount[i] contenga il numero di nodi presente nell'i-esima componente connessa;
- ****comps**: vettore di vettori contenenti gli indici dei nodi presenti all'interno delle componenti;

Nonostante non risulti necessario, al fine di rendere il codice maggiormente leggibile, si è deciso di assegnare sia alle variabili che ai puntatori il medesimo nome che assumono all'interno di *CCcut_connect_components*.

```
int *compscount = (int*)malloc(inst->nMaxCuts * sizeof(int));
int *comps = (int*)malloc(inst->nNodes * sizeof(int));
int nLati = ((inst->nNodes - 1)*inst->nNodes / 2);
int *elist = (int*)malloc((nLati * 2) * sizeof(int));
int ncomp;
```

La chiamata alla funzione risulta quindi essere:

```
if (CCcut\_connect\_components(inst->nNodes, nLati, elist, xstar, &ncomp,
   &compscount, &comps))
    printError(" error in CCcut\_connect\_components() inside
       fractcutusercallback");
```

Al suo termine, in modo del tutto trasparente, otteniamo le tre variabili **ncomp**, **compscount** e **comps** che forniscono tutte le informazioni necessarie all'aggiunta dei tagli all'interno dell'apposito pool fornito da Cplex. Completiamo quest'ultima operazione tramite la routine fornita da Cplex **CPXcutcallbackadd** la cui firma è:

```
CPXcutcallbackadd(CPXCENVptr env,void * cbdata,int  wherefrom,int  nzcnt,
   double  rhs,int sense,int cutind,double const *  cutval, int
   purgeable);
```

Dove:

- **env,cbdata,wherefrom**: parametri noti già discussi nella callback myLazyCallback;
- **nzcnt**: numero di coefficienti diversi da zero del vincolo;
- **rhs**: definisce il termine noto del vincolo;
- **sense**: può assumere i seguenti valori:
  - **cutind**: array di *nzcnt* elementi contenenti gli indici delle variabili presenti nel vincolo;
  - **cutval**: array di *nzcnt* elementi contenenti i corrispondenti valori dei coefficienti;
  - **purgeable**: valore intero che specifica come Cplex deve trattare il taglio:

* **CPX_USECUT_FORCE**: il taglio una volta aggiunto al rilassamento non può essere più rimosso;
* **CPX_USECUT_PURGE**: il taglio è aggiunto al rilassamento ma può essere eliminato in un secondo momento se giudicato inefficiente;
* **CPX_USECUT_FILTER**: il taglio deve essere trattato come se generato da Cplex il quale prima di aggiungerlo al rilassamento lo analizza e può quindi decidere di abortire l'operazione di aggiunta. nel rilassamento(per esempio è già presente un taglio più efficiente);

Per aggiungere un taglio per ogni componente connessa è necessario popolare i vettori **cutval**, **cutind** e la variabile **nzcnt** opportunamente sfruttando le informazioni fornite da *Concorde*. Per stabilire quali nodi appartengono alla t-esima componente connessa si sono dichiarate due variabili intere **k1** e **k2** che contengono sistematicamente l'indice del **primo** e dell'**ultimo** nodo tra quelli appartenenti alla t-esima componente connessa memorizzata in *comps*. Si osserva che *k2* è inizializzato al valore -1 in quanto gli indici di un qualsiasi vettore partono da 0.

```
if (ncomp > 1)
    {
        int k1 = 0;
        int k2 = -1;

        for (int c = 0; c < ncomp; c++)
        {
            int dimIndexValue = compscount[c] * (compscount[c] - 1) /
               2;
            int *cutind = (int*)malloc(dimIndexValue * sizeof(int));
            double *cutval = (double*)malloc(dimIndexValue *
               sizeof(double));
            int nzcnt = 0;


            k2 += compscount[c];

            for (int i = k1; i < k2; i++)
            {
                for (int j = i + 1; j <= k2; j++)
                {
                    cutval[nzcnt] = 1.0;
                    cutind[nzcnt] = xPos(comps[i], comps[j], inst);
                    nzcnt++;
                }
            }

            k1 = k2 + 1;

            CPXcutcallbackadd(env, cbdata, wherefrom, nzcnt,
               compscount[c] - 1, 'L', cutind, cutval,
               CPX\_USECUT\_FORCE);
```

```
            *useraction\_p = CPX\_CALLBACK\_SET;
            free(cutind);
            free(cutval);
        }

        free(elist);
        free(comps);
        free(compscount);
        free(xstar);


        return 0;
    }
```

Nel caso in cui la soluzione presenti una sola componente connessa, come in Fig. X, invocando la funzione **CCcut_violated_cuts** di *Concorde* è possibile individuare gli insiemi S che soddisfino la disuguaglianza (7): noto S risulta poi banale inserire il relativo vincolo di subtour. In particolare *CCcut_violated_cuts* è una funzione in grado di individuare sezioni di capacità inferiori ad una certa soglia. Descriviamo quindi i 7 parametri che tale funzione riceve in input:

- **int ncount**, **int ecount**, **int *elist**: il loro significato è già stato descritto per la funzione *CCcut_connect_components*;
- **dlen**: vettore contenente la capacità di ogni lato;
- **cutoff**:[Questo è il termine noto della disequazione f2, non ho capito perchè devo togliere a 2 un EPSILON, così è scritto nel pdf condiviso dal prof che si chiama RO2_TSPutilities]
- **(*(doit_fn)(double, int, int *, void *)** : è una funzione creata da noi che risulta essere una vera e propria callback: ogniqualvolta *Concorde* individua un insieme S cercato tale funzione viene invocata. Al suo interno, grazie ai parametri forniti[29] è nostro compito procedere all'ampliamento del pool di tagli di *Cplex*.
- **pass_param**: puntatore ad una struttura dati contenente variabili e puntatori che devono essere accessibili all'interno della callback;

Nel nostro caso l'invocazione di tale metodo avviene nel seguente modo:

```
CCcut_violated_cuts(inst->nNodes, inst->nCols, elist, xstar, 2.0 -
    cutThreshold, doitFuncConcorde, (void*)&in)
```

Dove occorre solamente far notare che la funzione callback da noi definita prende il nome **doitFuncConcorde** mentre l'ultimo parametro è una **struct** da noi creata contenente al suo interno tutte le informazioni occorrenti per invocare il metodo **CPXcutcallbackadd**, descritto in precedenza, all'interno della callback.

[29] Maggiori dettagli riguardo i quattro parametri di ingresso saranno forniti a breve durante la descrizione di come l'implementazione di tale funzione è stata da noi realizzata.

```
typedef struct
{
    instance *inst;
    CPXCENVptr env;
    void *cbdata;
    int wherefrom;
    int *useraction\_p;
} inputCC;
```

Per concludere questo paragrafo non rimane altro che parlare più in dettagli riguardo la realizzazione della callback **doitFuncConcorde**, per prima cosa forniamo la sua firma:

```
int doitFuncConcorde(double cutValue, int cutcount, int *cut, void
   *inParam)
```

Dove:

- **cutValue**: rappresenta il valore del taglio;
- **cutcount**: rappresenta il numero dei nodi;
- **cut**: array contenente l'indice associato ai nodi;
- **inParam**: struttura dati appena descritta;

Dopo aver effettuato la classica operazione di recupero del puntatore alla struttura dati fornita in ingresso alla callback

```
inputCC *in = (inputCC*)inParam;
```

possiamo procedere con l'aggiunta del taglio attraverso *CPXcutcallbackadd*: si sono così definiti due array di interi **cutind** e **cutval** contenenti rispettivamente gli indici delle variabili che costituiscono il taglio ed il relativo coefficiente. Si è inoltre dichiarata una variabile intera **nzcnt** che contiene il numero di variabili caratterizzanti il taglio:

```
int dimIndexValue = inst->nNodes * (inst->nNodes - 1) / 2;
int *cutind = (int*)malloc(dimIndexValue * sizeof(int));
double *cutval = (double*)malloc(dimIndexValue * sizeof(double));
int nzcnt = 0;

    for (int i = 0; i < cutcount - 1; i++)
    {
        for (int j = i + 1; j <= cutcount - 1; j++)
        {
            int n1 = cut[i];
            int n2 = cut[j];
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
            METTI QUA IL COMMENTO A xPos che avevi fatto... il codice va
               commentato !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            cutind[nzcnt] = xPos(n1, n2, inst);
            cutval[nzcnt] = 1.0;
            nzcnt++;
        }
    }
CPXcutcallbackadd(in->env, in->cbdata, in->wherefrom, nzcnt, cutcount -
   1, 'L', cutind, cutval, CPX\_USECUT\_FORCE);

*in->useraction\_p = CPX\_CALLBACK\_SET;

free(cutind);
free(cutval);

return 0;
```

INTRODUZIONE ALGORITMI EURISTICI

Fino a questo momento sono stati presentati algoritmi che, alla loro naturale terminazione, garantiscono di risolvere una generica istanza del problema del commesso viaggiatore in modo esatto, ovvero restituendo sempre un ottimo globale come soluzione. L'applicazione di metodi esatti non è sempre possibile per due motivi principali: il primo è che si necessita di un programma di calcolo molto potente ed in genere costoso come può essere **Cplex**, in secondo luogo, quando si procede all'analisi di problemi **NP-hard**, indipendentemente da quali accorgimenti introduciamo, non è mai garantito di trovare la soluzione migliore in tempi relativamente brevi. Per tali ragioni, nelle applicazioni reali, capita spesso che l'unica strada percorribile sia il ricorrere a metodi *non esatti*, che, come abbiamo già potuto vedere nelle varianti del metodo **Loop** presentate, prendono il nome di algoritmi euristici: la soluzione che offrono sarà sempre ammissibile ma non viene garantità la sua ottimilità, al contrario nella maggior parte dei casi, soprattutto per istanze complesse, questa non viene quasi mai raggiunta. É inoltre tenere ben presente che esistono molteplici algoritmi euristici più o meno potenti, essendo inoltre tecniche non esatte, è possibile trovarne infinite varianti per ognuno di essi. In generale, come meglio vedremo nel seguito del testo, possiamo classificarli in **costruttivi**, **migliorativi**, **metaeuristici** e **matheuristics**. Quanto esposto fino ad ora ci porta intuitivamente a pensare che la bontà di un metodo proposto può subire enormi variazioni in base a quali istanze su cui viene applicato.

Gli euristici costruttivi determinano (costruiscono) una soluzione ammissibile per passi partendo inizialmente da una soluzione vuota. Ad ogni iterazione selezionano in modo iterativo nuovi elementi da aggiungere alla soluzione parziale corrente sino ad arrivare ad una soluzione completa (criterio di espansione). Gli euristici migliorativi, invece, a partire da una soluzione ammissibile, cercano possibili soluzioni ammissibili migliori in termini di funzione obbiettivo apportando "piccole" modifiche alla soluzione data. Spesso algoritmi che appartengono a queste due classi di euristici vengono utilizzati in sequenza: si ricorre ad un euristico costruttivo per generare una soluzione ammissi-

bile e successivamente si tenta di migliorarla attraverso l' utilizzo di un euristico di miglioramento. Poiché la soluzione prodotta dai metodi di miglioramento spesso risulta essere un ottimo locale, si concluderà il capitolo con la presentazione di algoritmi metaeuristici i quali sono stati sviluppati per "sfuggire" in maniera "intelligente" dai minimi locali al fine di calcolare soluzioni migliori. TOGLIERE!!!

ALGORITMI COSTRUTTIVI GREEDY

Gli algoritmi costruttivi hanno la caratteristica di determinare una soluzione ammissibile partendo da una *vuota*. Quest'ultima, durante tutto il corso dell'agoritmo, seguendo il criterio di espansione, viene continuamente aggiornata ed ampliata attraverso l'introduzione di nuove componenti fintanto che non diviene completa e quindi ammissibile. Una sottocategoria molto importante degli algoritmi costruttivi viene definita come **greedy**: la soluzione euristica al problema è ottenuta attraverso una sequenza *finita* di decisioni "localmente ottime". In altre parole, attraverso una struttura ricorsiva, ad ogni sua iterazione, la soluzione parziale viene aggiornata con l'aggiunta dell'elemento migliore disponibile in quel momento. La correttezza di queste operazione non deve però mai essere verificata runtime dell'algoritmo ma solamente a livello teorico durante la sua fase di progettazione. |'E proprio questa caratteristica che motiva il nome *greedy* e soprattutto rende tali algoritmi estremamente velociti.

```
Procedure Greedy(E, F, S):
  begin
    S := ∅; Q := E;
    repeat
      e := Best(Q); Q := Q \ {e};
      if S ∪ {e} ∈ F
        then S := S ∪ {e}
    until Q = ∅;
  end.
```

Figura X: Algoritmo Greedy

Nella procedura, **S** è l'insieme degli elementi di **E** che sono stati inseriti nella soluzione parziale corrente mentre **Q** è l'insieme degli elementi appartenenti ad **E** ancora da esaminare. La procedura fa uso della sottoprocedura **Best** la quale fornisce il miglior elemento di **E** tra quelli ancora in **Q** sulla base di un prefissato criterio euristico. Come esempio della categoria di algoritmi euristici costruttivi greedy presentiamo nel paragrafo seguente la tecnica del **nearest neighbour**.

ALGORITMO NEAREST NEIGHBOUR

L'algoritmo **nearest neighbour** è stato uno dei primi algoritmi utilizzati per risolvere istanze del

problema del commesso viaggiatore. La sua dimostrazione di correttezza non viene qui riportata in quanto non risulta di interesse ai fine del nostro progetto ed è inoltre ampiamente discussa in letteratura. Limitiamoci quindi ad una descrizione del concetto fondamentale alla base dell'algoritmo: dato un qualsiasi percorso (soluzione) parziale, il modo migliore per ampliarlo è banalmente attraverso l'aggiunta di un nuovo arco a costo **minore** avente come estremi uno dei due nodi liberi[30] del percorso stesso mentre il secondo sia uno ancora disponibile[31].

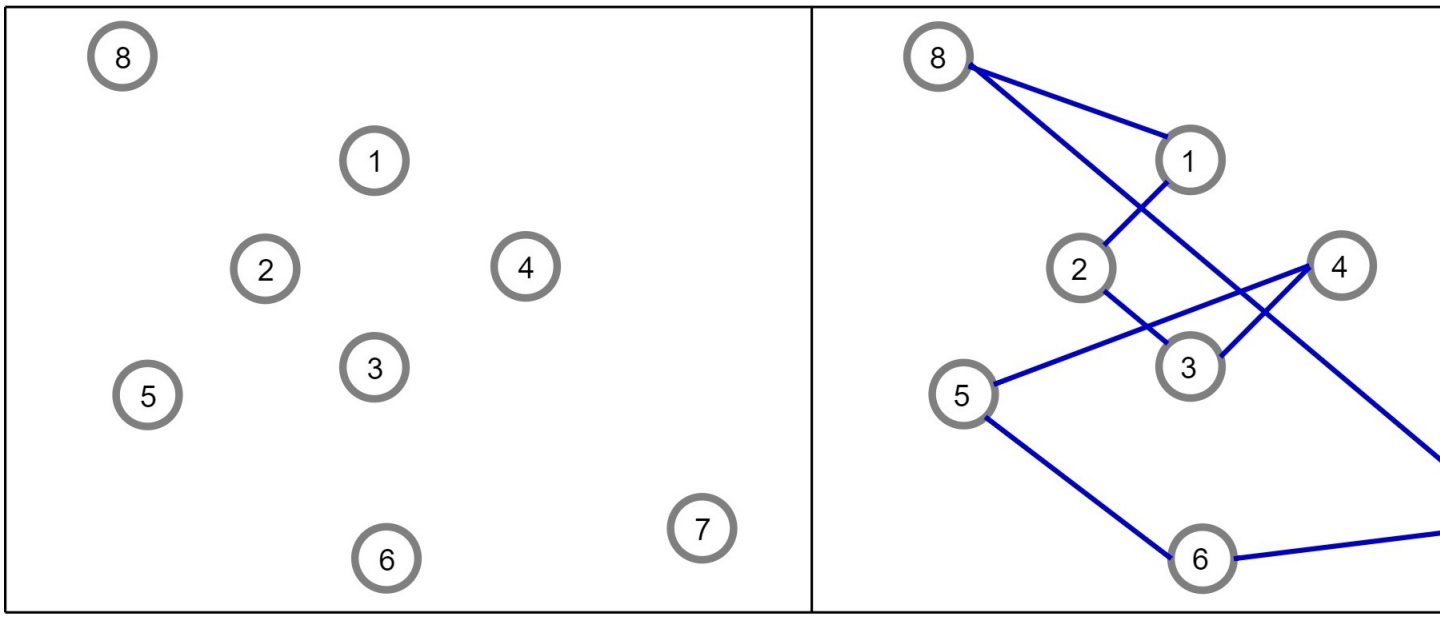

Figura X:Esempio di algoritmo Nearest Neighbour

Entriamo ora più nel dettaglio nella realizzazione di questo algoritmo descrivendone i passaggi fondamentali. Noteremo che sono state fatte delle scelte di programmazione che in un primo momento potrebbero apparire inmotivate se considerate limitatamente a qanto visto fino ad ora. L'algoritmo preso in analisi produce una soluzione valida molto velocemente ma che risulta essere scadente per quanto riguarda il suo costo, nel nostro progetto di conseguenza il *nearest neighbour* viene unicamente utilizzato per fornire una o più soluzioni di partenza per altri algoritmi euristici. Risulta pertanto di maggior importanza che al suo interno sia introdotta una certa randomicità nelle scelte che effettua in modo tale che multipli utilizzi sulla stessa instanza del TSP producano soluzioni tra loro scorrelate. Algoritmi che presentano tale caratteristica possono essere trovati in letteratura con la dicitura **GRASP**[32]

- Innanzi tutto, partendo da una soluzione vuota è necessaria una operazione preliminare prima di innescare la ricorsività dell'algoritmo. In altre parole dobbiamo fornire un nodo iniziale il cui lato incidente a costo minore diventa il la prima vera componente della soluzione che quindi da vuota diviene parziale. La scelta di quale debba essere il nodo di partenza avviene in modo casuale;

[30]Nodi attraversati del percorso parziale ma aventi un solo lato incidente.
[31]Non ancora attraversato dal circuito parziale.
[32]Greedy Randomly Adaptive Search Procedure.

- Come sarà più chiaro in seguito, l'algoritmo *nearest neighbour* non produce mai soluzioni soddisfacenti ed i solamente utilizzato come punto di partenza per algoritmi più complessi. Questi ultimi necessitano in genere di una struttura dati che veda il circuito prodotto come un percorso orientato. Seguendo questa linea di pensiero, supponendo che l'ultima iterazione abbia introdotto il nodo j nel percorso parzile, il lato successivo che andremo a selezionare dovrà sempre essere incidente in j. In questo modo risulta molto più semplice tenere traccia della soluzione come un vero e proprio percorso orientato, essendo questo utilizzato solamente come input per altri metodo più complessi la nostra scelta non risulta in alcun modo limitante;

- Per introdurre un ulteriore livello di casualità nell'algoritmo, la scelta di quale lato entra a far parte della soluzione parziale non ricade sempre in quello a costo minore. In particolare dopo varie prove si è deciso che quest'ultima opzione avviene con una percentuale del 90%, mentre con il 9% la scelta ricade nel secondo miglior lato e con il restate 1% nel terzo miglior lato;

- Spesso può capitare che il lato designito per essere aggiunto al percorso causerebbe la presenza di un cappio al suo interno. Naturalmente la soluzione finale risulterebbe non valida e quindi tale situazione deve essere evitata. Banalmente, nel caso in cui il lato in questione sia la $i-esima$ scelta migliore, questo viene sostituito dalla $(i+1)-esima$ miglior scelta. Naturalmente il tutto viene ripetuto iterativamente fino a che non si trova un lato accettabile;

Di seguito è riportata la realizzazione del codice tenendo presente che è richiesta in input una struttura dati che permetta di ottenere l'ordine crescente, per ogni nodo, dei lati in esso incidenti basandosi chiaramente sul loro costo. Tale informazione è fornita dal metodo **BuildSLComplete** già descritto brevemente LINK!!!!!!!!!!!!! (loop euristico) ed in dettaglio nell'apposito paragrafo della appendice LINK!!!!!!!!!!!!!!!!!!.

```
public static PathGenetic NearestNeighbour(Instance instance, Random rnd,
    List<int>[] listArray)
{
// heuristicSolution is the path of the current heuristic solution to
    generate
     int[] heuristicSolution = new int[instance.NNodes];
     double distHeuristic = 0;

     int currentIndex = rnd.Next(instance.NNodes);
     int startindex = currentIndex;

     bool[] availableIndexes = new bool[instance.NNodes];

     availableIndexes[currentIndex] = true;

     for (int i = 0; i < instance.NNodes - 1; i++)
     {
         bool found = false;

         int plus = RndPlus(rnd);

         int nextIndex = listArray[currentIndex][0 + plus];
```

```
        do
        {
            if (availableIndexes[nextIndex] == false)
            {
                heuristicSolution[currentIndex] = nextIndex;
                distHeuristic +=
                   Point.Distance(instance.Coord[currentIndex],
                   instance.Coord[nextIndex], instance.EdgeType);
                availableIndexes[nextIndex] = true;
                currentIndex = nextIndex;
                found = true;
            }
            else
            {
                plus++;
                if (plus >= instance.NNodes - 1)
                {
                    nextIndex = listArray[currentIndex][0];
                    plus = 0;
                }
                else
                    nextIndex = listArray[currentIndex][0 + plus];
            }

        } while (!found);
    }

    heuristicSolution[currentIndex] = startindex;
    distHeuristic += Point.Distance(instance.Coord[currentIndex],
       instance.Coord[startindex], instance.EdgeType);

    return new PathGenetic(heuristicSolution, distHeuristic);
}
```

ALGORITMI MIGLIORATIVI

Gli algoritmi euristici migliorativi si basano su un'idea estremamente semplice ed intuitiva: data una soluzione ammissibile **x**, relativa ad un problema di ottimizzazione, viene esaminato se attraverso minime variazioni questa risulta migliorabile in termini di funzione obiettivo. In gergo più tecnico si parla di ricercare soluzioni *vicine* a quella attuale ma migliorative. Per poter definire il concetto di "vicinanza" è necessario discutere quello di **mossa**. Questa è una operazione di modifica (caratteristica dell'algoritmo migliorativo) che viene eseguita su **x** e che ha come conseguenza la generazione di un **insieme** di soluzioni ammissibili le quali costituiscono un intorno di **x**, indicato con **N(x)**. Si parla allora di **y** vicina ad **x** se e solo se differiscono tra loro per una sola mossa e quindi $y \in N(x)$. Una volta definito $N(x)$ questo viene esplorato secondo due possibili strategie che sono **first improvement** e **steepest descent**. Nel primo caso l'esplorazione dell'intorno termina non appena si trova una soluzione migliore di quella corrente. Nel secondo caso, invece, l'esplorazione è completa

e viene trovato il miglioramento più consistente.
Qualora esista una soluzione **y** migliore di **x**, il procedimento viene iterato esplorando N(y); viceversa l'algoritmo si arresta. Giunti a questo punto il risultato finale può essere una soluzione *localmente ottima* oppure, più raramente, globalmente ottima[33]. Poiché da un punto di vista matematico il processo di ricerca analizza, ad ogni interazione, un intorno della soluzione corrente, gli algoritmi migliorativi vengono anche chiamati *algoritmi di ricerca locale*.
Tranne alcuni casi particolari in cui la funzione obiettivo ha determinate caratteristiche di convessità, nella maggior parte dei problemi reali questa presenta un grande numero di minimi locali che spesso si discostano totalmente dell'ottimo globale. In effetti, una delle fortunate eccezioni è il metodo del simplesso per la programmazione lineare che si pone alla base degli studi in questo settore: esso fornisce sia un metodo per analizzare in un numero finito di passi tutti gli ottimi locali del problema e soprattutto se questi sono anche globali.
Tra i più famosi algoritmi migliorativi applicabili al problema del commesso viaggiatore troviamo i **K-Opt** dove **K** è in genere un numero intero superiore a 2. Procediamo quindi ad una loro descrizione generale seguita da una implementazione particolare della tecnica del **2-Opt**.

ALGORITMO K-OPT

Gli algoritmi **K-Opt**, sigla inglese per K-Ottimalità, fanno parte della categoria degli algoritmi migliorativi e sono caratterizzati da una mossa, applicata ad un circuito hamiltoniano, consistente nello **scambio** di **K** archi con altrettanti non facenti parte del percorso producendone uno *vicino*, migliore e chiaramente valido. Come specificato nel paragrafo precedente **K** può assumere qualsiasi valore superiore a 2[34] ma in genere è proprio $K = 2$ l'unica variante realmente utilizzata. Risulta infatti facilmente verificabile che più il suo valore è alto, più salgono sia la complessità computazione che di scrittura
progettazione dell'algoritmo[35] perdendo così i vantaggi offerti dall'utilizzo di tecniche euristiche. Concentriamoci quindi unicamente nella variante $2-Opt$: presa una qualsiasi coppia di archi distinti $([i,j];[h,k])$[36] è possibile sostituirli correttamente con una sola delle combinazioni $([i,k];[h,j])$ e $([i,h];[j,k])$. Una di queste due, infatti, trasforma la soluzione in una seconda contenente due subtour. A discapito quindi di un concetto molto basilare, l'algoritmo $2-opt$[37] introduce la difficoltà di verificare quale delle due possibili sostituzioni è valida. L'approccio migliore in questi casi è di introdurre un fittizio ordinamento nel circuito attraverso una struttura dati di supporto apposita. Come è possibile infatti vedere dalla figura X, se gli archi sono ordinati, e quindi lo è anche il circuito, la difficoltà appena discussa è facilmente risolvibile:

[33] Da notare che un ottimo è globale se lo è anche localmente

[34] Chiaramente **K** non può superare il numero di archi che costituiscono la soluzione.

[35] Vedremo in seguito che questa affermazione è valida solo per algoritmi che applicano direttamente la definizione di K-Ottimalità. Esistono infatti metodi che la ottengono indirettamente se sono caratterizzati da tempi di esecuzione molto buoni.

[36] Notiamo che questi non possono mai essere presi consecutivi e cioè con un vertice in comune in quanto non ci sarebbe modo di produrre una soluzione *vicina* valida.

[37] In generale lo stesso discorso può applicarsi anche a tutto il resto della famiglia di algoritmi.

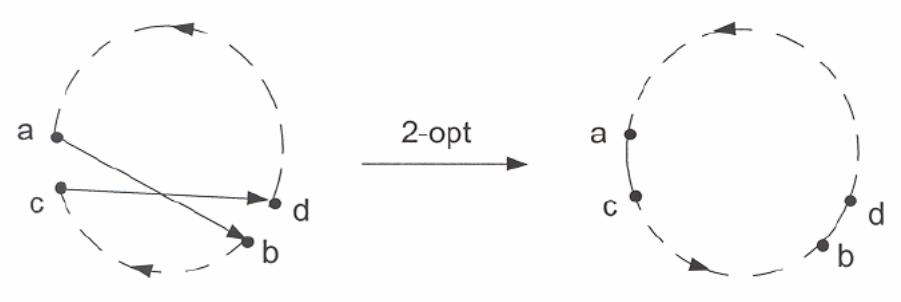

Figura X:Esempio di azione nell'algoritmo 2-opt

Viene da sé che applicare l'approccio $2-Opt$ a qualsiasi coppia di archi non garantisce un miglioramento del costo della soluzione. Questo però avviene nel 100% dei casi se i due archi in questione visualmente si incrociano come mostrato nella figura X, tale affermazione è facilmente verificabile matematicamente. Nel complesso, nel caso in cui si voglia trovare l'operazione di due ottimalità migliore[38] è necessario confrontare tutte le coppie possibili ottenendo quindi una complessità computazione pari a $O(n^2)$ rispetto al numero di nodi mentre nel caso di un generico valore di K si passa ovviamente a $O(n^3)$. Nel caso del *TSP*, alcuni esperimenti svolti versoo la fine degli anni '50 hanno mostrato che il passaggio da due ottimalità a tre ottimalità porta un miglioramento sensibile della qualità della soluzione trovata, che giustifica pienamente l'aumento di carico computazionale ma non il costo di scrittura
progettazione dell'algoritmo se paragonato ad altri metodi euristici che mostreremo nel seguito del testo. Miglioramenti praticamente trascurabili si hanno invece per $K > 3$. Concludiamo le nozioni riguardanti i metodi migliorativi di K ottimalità nel seguente paragrafo dove vengono mostrati i dettagli implementativi dell'algoritmo $2-Opt$ da noi progettato

METODO TwoOpt

Il metodo TwoOpt implementa l' euristico algoritmo $2-Opt$ applicato al problema del commesso viaggiatore la cui discussione teorica è stata presentata nel paragrafo precedente.

```
public static void TwoOpt(Instance instance, PathStandard pathG)
```

Dove:

- **instance**: oggetto dove sono memorizzati i dati relativi alla istanza del problema TSP;
- **pathG**: rappresenta il percorso sul quale l'algoritmo viene eseguito;

La classe **PathStandard**, descritta in dettagli in questo capitolo della appendice LINK!!!!!!!!!!!!!!!!, permette di memorizzare una soluzione del problema come un percorso ordinato. Molto semplicemente al suo interno si mantiene aggiornato un vettore di interi di nome **path** dove all'indice $i-esimo$ troviamo il nodo successivo da visitare, seguendo un il percorso attuale, trovandosi nel vertice di indice i. Di seguito è riportato il contenuto del metodo **TwoOpt** che nel nostro caso utilizza la tecnica *first improvement*, già discussa nei paragrafi precedenti.

[38]Cioè trovare la soluzione vicina a costo più basso

```
int indexStart = 0;
int cnt = 0;
bool found = false;

do
{
   found = false;
   int a = indexStart;
   int b = pathG.path[a];
   int c = pathG.path[b];
   int d = pathG.path[c];

   for (int i = 0; i < instance.NNodes - 3; i++)
   {
      double distAC = Point.Distance(instance.Coord[a],
         instance.Coord[c], instance.EdgeType);
      double distBD = Point.Distance(instance.Coord[b],
         instance.Coord[d], instance.EdgeType);
      double distAD = Point.Distance(instance.Coord[a],
         instance.Coord[d], instance.EdgeType);
      double distBC = Point.Distance(instance.Coord[b],
         instance.Coord[c], instance.EdgeType);

      double distTotABCD = Point.Distance(instance.Coord[a],
         instance.Coord[b], instance.EdgeType) +
      Point.Distance(instance.Coord[c], instance.Coord[d],
         instance.EdgeType);

      if (distAC + distBD < distTotABCD)
      {
            Utility.SwapRoute(c, b, pathG);
            pathG.path[a] = c;
            pathG.path[b] = d;
            pathG.cost = pathG.cost - distTotABCD + distAC + distBD;
            indexStart = 0;
            cnt = 0;
            found = true;
            //"break" = "first improvement" technique
            break;
      }

      c = d;
      d = pathG.path[c];
    }

    if (!found)
    {
        indexStart = b;
        cnt++;
    }
```

```
} while (cnt < instance.NNodes);
```

É utile fare infine le seguenti annotazioni: le assegnazioni degli indici possono ad un primo sguardo sembrare errate in quanto non tutte le possibili coppie di lati vengono analizzate, in realtà questo non avviene solamente per quelli tra loro consecutivi. Infine, dato che come vedremo a breve la tecnica della due ottimalità è utilizzata in un contesto più ampio, il percorso modificato viene poi riutilizzato e quindi è necessario mantenere aggiornato anche il suo ordinamento fittizio. Questa funzionalità è offerta dal metodo **SwapRoute** descritto nel dettaglio nella appendice LINK!!!!!!!.

MULTISTART

La tecnica del *multi start* è un modo molto semplice per combinare i due algoritmi euristici proposti, *nearest neighbour* e *2-Opt*, sfruttando appieno i loro punti di forza. L'idea alla base è la seguente: *nearest neighbour*, secondo l'implementazione presentata LINK!!!!!!!, offre la possibilità di produrre molto velocemente un numero soluzioni valide al problema TSP in esame tutto diverse tra loro; Il suo principale svantaggio è che queste presentano risultati molto scadenti per quanto riguarda la funzione obiettivo da minimizzare. A questo punto è logico pensare di introdurre la tecnica del *2-Opt*, l'algoritmo infatti necessità di una soluzione iniziale su cui essere applicato e produce velocemente risultati accettabili indipendentemente dalla bontà del punto di partenza. Nel complesso quindi, l'algoritmo *multi start* ripeto la combinazione appena proposta memorizzando solamente la soluzione migliore torvata durante il processo. Il tutto naturalmente fino allo scadere del classico timelimit ricevuto in ingresso dalla applicazione. Riportiamo quindi di seguito il codice commentato senza fornire ulteriori indicazioni in quanto la sua comprensione dovrebbe a questo punto risultare facile:

```
//It stores the current best path found
PathStandard incumbentSol = new PathStandard();
//It stores the latest path found
PathStandard heuristicSol;
//Used by nearest neighbour, it orders the links accident in a generic
   node based on their cost
List<int>[] listArray = Utility.BuildSLComplete(instance);

//At least one time the combo nearest neighbour and 2-Opt is used to
   produce a valide solution
do
{
//Using the nearest neighbour technique
heuristicSol = Utility.NearestNeighbour(instance, rnd, listArray);

//Using the 2-Opt technique on the nearest neighbour solution produced
TwoOpt(instance, heuristicSol);

//Confronting the best solution so far with the latest
if (incumbentSol.cost > heuristicSol.cost)
{
```

```
incumbentSol = heuristicSol;

Console.WriteLine("Incumbed changed");
}
else
Console.WriteLine("Incumbed not changed");

} while (clock.ElapsedMilliseconds / 1000.0 < instance.TimeLimit);
   //Cicle is repeated until the time limit is over
```

GENERAZIONE NUMERI CASUALI - SEMERANDOM DA SPOSTARE!!!!!!!!!!!!!!! INIZIO O APPENDICE

Per generare un numero casuale è sufficiente istanziare la classe Random ed invocare sull' istanza creata il metodo **Next** o **NextDouble**. Per esempio nel caso in cui si voglia generare un numero casuale intero tra 1 e 99 è necessario scrivere le seguenti righe di codice:

```
Random random = new Random();
int nun = random.Next(1,100);
```

I numeri random sono generati, a partire da un valore d' inizializzazione chiamato **seme**, da un algoritmo matematico. Per sua natura l' algoritmo è deterministico: se si fornisce in input lo stesso seme genererà sempre la medesima sequenza di numeri. Per tale ragione il valore del seme viene derivato dall'orologio di sistema all' atto della creazione dell' istanza della classe Random qualora si utilizza il costruttore di default. In questo modo, non essendo predicibile il valore del seme, la sequenza di numeri generati dall' algoritmo risulta essere sistematicamente casuale.
L' orologio di sistema non sempre risulta un buon valore da utilizzare per settare il seme. Si supponga di avere la necessità di creare due oggetti diversi della classe Random: qualora quest' ultimi siano creati uno di seguito all' altro avranno entrambi il medesimo seme poichè l' orologio di sistema risulta lo stesso nel lasso di tempo che il processore impiega ad eseguire le due istruzioni.
Per constatare ciò si è realizzato il seguente programma:

```
Random rdn1 = new Random();
Random rdn2 = new Random();

for (int i = 0; i < 10; i++)
{
    Console.WriteLine(rdn1.Next(1, 10) + "-" + rdn2.Next(1, 10));
}
Console.ReadLine();
```

Il cui output, come previsto, risulta mostrato in Fig C.

Figura X: Output

Per ovviare a questa problematica la classe Random dispone di un secondo costruttore che riceve come parametro un intero che setta il valore del seme: sarà a questo punto compito del programmatore passare valori casuali e differenti all' atto della creazione delle due istanze della classe Random.

METAEURISTICI

TABU

VNS

MATHEURISTICS

HARD FIXING

WARM-START

PREPROCESSING

IMPLEMENTAZIONE HARD FIXING

RINS

POLISHING

LOCAL BRANCHING

IMPLEMENTAZIONE LOCAL BRANCHING

LOCAL BRANCH SIMMETRICO

ALGORITMI GENETICI

Gli Algoritmi Genetici (AG), proposti nel 1975 da J.H. Holland, sono un modello computazionale idealizzato dall'evoluzione naturale darwinista. L'aggettivo "genetico" deriva dal fatto che il modello evolutivo darwiniano trova spiegazioni nella branca della biologia detta genetica e dal fatto che tali algoritmi attuano meccaniche concettualmente simili a quelli dei processi biochimici scoperti da questa scienza. I principi fondamentali che consentono la nascita e lo sviluppo di un processo evolutivo che porta all'evoluzione di una specie sono la **selezione naturale** e la **varietà del genotipo**[39] della popolazione. La selezione naturale è il meccanismo grazie al quale si ha un progressivo e cumulativo aumento della frequenza degli individui aventi caratteristiche ottimali per l'ambiente in cui essi vivono poiché solo quelli che meglio si adattano ad un certo habitat riescono a sopravvivere e a riprodursi. I meccanismi generatori della variazione del genotipo della popolazione sono sostanzialmente due:

[39]Il termine *genotipo* indica la costituzione genetica di un organismo o di un gruppo di individui

- Un processo di **riproduzione** nel quale gli individui, detti genitori, si accoppiano producendo di nuovi, detti figli, il cui patrimonio genetico risulta pertanto una combinazione di quello dei genitori;
- Un processo di **mutazione** che colpisce i figli i quali subiscono una modifica del patrimonio genetico ereditato dai genitori per effetto dell'ambiente che li circonda;

I cambiamenti che si verificano da una generazione all'altra risultano essere molto piccoli ma, dato che sopravvivono soprattutto quelli positivi, un loro accumulo porta nel tempo a grandi cambiamenti. La ricerca parte da una popolazione iniziale di individui, detti cromosomi, che rappresentano ipotetiche soluzioni al problema dato. Ogni individuo della popolazione viene codificato da un vettore[40] i cui elementi contengono simboli appartenenti ad un alfabeto finito, detti geni. Ad ogni soluzione è associato un valore determinato da una funzione chiamata **Fitness** il cui scopo è di determinare la bontà di un individuo nel risolvere il problema in questione. Così come nella natura solamente gli individui che meglio si adattano all' ambiente sono in grado di sopravvivere e riprodursi, anche negli algoritmi genetici le soluzioni migliori sono quelle che hanno la maggiore probabilità di trasmettere i propri geni alle generazioni future. Come vedremo in seguito sono fondamentalmente tre le caratteristiche determinanti per un algoritmo genetico: determinare quale funzione di fitness si andrà ad utilizzare, partendo dalla attuale generazione decidere come creare un pull di possibili candidati per quella successiva ed infine come selezionare tra questi ultimi quelli che sopravviveranno. Essendo la definizione delle funzione di fitness direttamente dipendente da quale tipo di problema si desidera studiare, concludiamo questa introduzione elencando solamente quali operatori genetici è possibile applicare per definire le restanti due caratteristiche di un algoritmo.

OPERATORI GENETICI

In questo paragrafo vengono trattati i principali operatori genetici applicabili ai cromosomi. Per ogni operatore vengono inoltre descritte le principali varianti che si possono trovare in letteratura.

OPERATORE DI CROSSOVER

Il crossover è una metafora della riproduzione in cui il materiale genetico dei discendenti è una combinazione di quello dei genitori. Di seguito sono indicati alcuni dei metodi più comuni per creare un *figlio* partendo da due *genitori*, le instanze così ottenute vanno a far parte di quelle candidate alla sopravvivenza per la generazione successiva:

- **Crossover ad un punto**: date due soluzioni si tagliano i loro vettori di codifica in un punto casuale o predefinito per ottenere due teste $\{H_a, H_b\}$ e due code$\{T_a, T_b\}$, si possono costruire quindi altrettante soluzioni distinte combinando la testa di un genitore con la coda dell'altro $S_1 = H_a \cup T_b, S_2 = H_b \cup T_1$;

[40] Oltre alla codifica vettoriale in letteratura è possibile trovare anche quella ad albero. Tuttavia essa viene utilizzata per codificare gli individui della popolazione nell'ambito della programmazione genetica (che è?????????).

- **Crossover a due punti**: date due soluzioni si tagliano i loro vettori di codifica in due punti predefiniti o casuali al fine di ottenere una coppia di teste$\{H_a, H_b\}$, parti centrale $\{I_a, I_b\}$ ed code $\{T_a, T_b\}$. Le due soluzioni sono ottenute scambiando le due parti centrali nei genitori $S_1 = H_a \cup I_b \cup T_a, S_2 = H_b \cup I_a \cup T_b$;
- **Crossover uniforme**: consiste nello scambiare casualmente elementi tra le soluzioni candidate all'evoluzione;
- **Crossover aritmetico**: consiste nell'utilizzare un'operazione aritmetica per creare la nuova soluzione, ad esempio eseguendo una *XOR* o una *AND* tra elementi dei genitori se interpretati come una sequenza binaria;

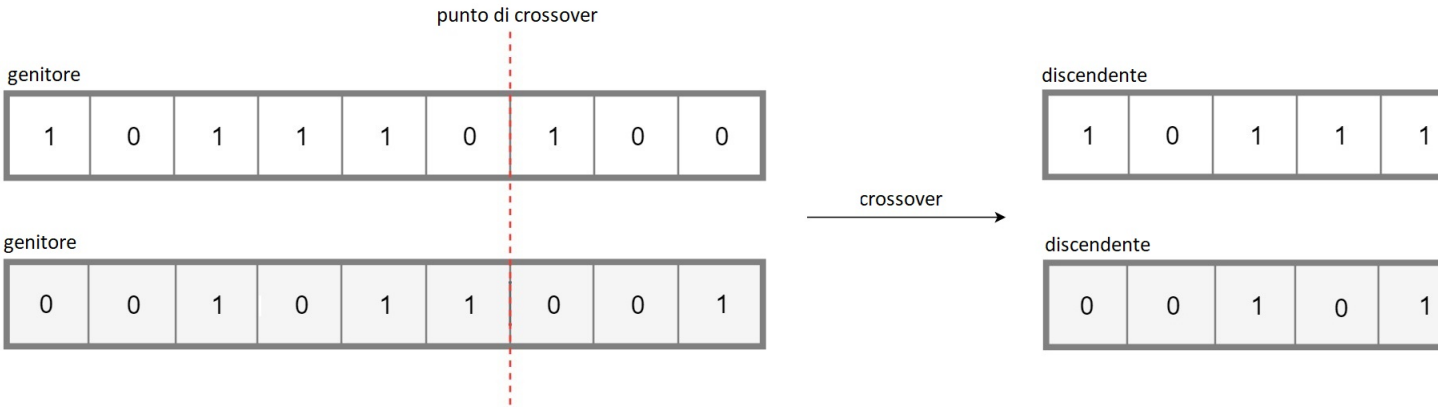

Figura 3: Esempio di operatore di crossover

OPERATORI DI SELEZIONE

A causa di complessi fenomeni di interazione non lineare, non è sempre vero che da due soluzioni promettenti ne nasca una terza *migliore* né che da due soluzioni con valori di fitness basso venga generato un figlio *peggiore*. Pertanto non è statisticamente conveniente utilizzare i soli elementi con valori di fitness elevata sia durante la scelta dei genitori che durante la scelta di quali elementi faranno parte della generazione successiva. Per quanto riguarda quest'ultimo caso, oltre al semplice valore di fitness, vengono prese in considerazione particolari tecniche di *selezione*. Le più comuni sono:

- **Selezione a roulette**: la probabilità che una soluzione venga scelta per far parte della successiva generazione è direttamente proporzionale al valore restituito dalla funzione di fitness. Immaginiamo quindi di avere a disposizione una roulette la cui ruota viene divisa in sezioni tutte assegnate ai vari candidati, la loro grandezza è quindi proporzionale all'idoneità dell'individuo. La selezione è banalamente ottenuta con molteplici rotazioni della roulette tenendo conto che un individuo non può essere selezionato più volte. Questa tecnica presenta dei problemi nel caso in cui le sezioni della ruota risultino tra loro eccessivamente sbilanciate

in ampiezza, le soluzioni peggiori vengono selezionate troppo raramente e questo per quanto già esposto non è necessariamente un bene;

- **Selezione di Boltzmann**: le soluzioni vengono scelte con un grado di probabilità che, agli inizi dell'algoritmo, favorisce *l'esplorazione* mentre più avanti tende a stabilizzarsi. Questa tecnica ritiene utile, in un primo momento, consentire agli individui meno idonei di riprodursi quasi quanto quelli migliori, e far procedere lentamente la selezione così da mantenere una certa diversità all'interno della popolazione. In seguito si rafforza la selezione per favorire maggiormente gli individui ad alta idoneità, presumendo che la fase iniziale, con grande diversità e poca selezione, abbia consentito alla popolazione di individuare la zona giusta nello spazio di ricerca;
- **Selezione a torneo**: da un pool di possibili soluzioni, nel caso più comune vengono scelti in modo del tutto casuale sia due individui che un numero $c \in [0, 1]$. Se quest'ultimo risulta minore di un parametro $k \in [0, 1]$ fissato, si seleziona il più idoneo tra i due candidati, altrimenti la scelta ricade sul peggiore. Naturalmente si procede fino a quando non ho tutti gli elementi per la generazione successiva.

Non esiste in assoluto un metodo migliore tra quelli proposti, molto dipende direttamente da come questi sono implementati e soprattutto sia dalla dimensione del problema che dalla quantità di vincoli imposti: ad esempio, nel caso in cui sia richiesto di trovare nel minor tempo possibile una **buona** soluzione è sconsigliato utilizzare la selezione di Boltzmann.

MUTAZIONE

L' operatore di mutazione prevede che in funzione di una prefissata e usualmente piccola probabilità $p_{mutation}$, il valore di un bit del figlio venga cambiato: questo serve per simulare quanto avviene in natura dove, anche se raramente, è possibile che vi sia una variazione del genotipo durante l' evoluzione di un essere vivente. La figura x.y illustra un esempio di mutazione.

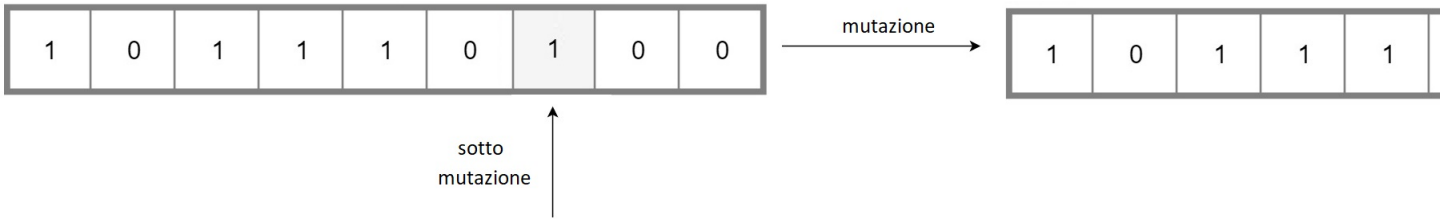

Figura 3: Esempio di mutazione

ALGORITMO GENETICO REALZIZZATO PER TSP

Gli AG risolvono un determinato problema generando sempre nuove **popolazioni** di soluzioni dove in genere troviamo una fitness media piuttosto bassa, giungendo solamente dopo diverse generazioni

a valori più elevati. Per poter applicare un algoritmo genetico, occorre anzitutto codificare numericamente le soluzioni e individuare una opportuna funzione di fitness. La codifica vettoriale dei cromosomi più adatta per i problemi di TSP risulta essere un vettore di interi dove ogni elemento identifica in maniera univoca una delle città da visitare mentre il suo posizionamento identifica l'ordine di visita. La funzione di fitness realizzata riceve in ingresso una soluzione **ammissibile**[41] e restituisce un valore reale pari al reciproco del suo costo PERCHE'?????? LO DICIAMO DOPO???????. La prima generazione viene ottenuta attraverso il metodo **NearestNeightborGenetic** dell classe **Utility**. Come facilmente intuibile genera soluzioni che tendono a collegare nodi tra loro vicini, per maggiori dettagli si consulti l'apposita sezione ad esso dedicata nella appendice LINKK!!!!!!!!!!!!!!!!!. La generazione di un *figlio* a partire da due *genitori* avviene attraverso il crossover ad un punto già presentato dove però solamente una delle due soluzioni ottenute entra a far parte del pool di candidati per la successiva generazione. Il crossover viene fornito dal metodo **GenerateChild**, sempre appartenente alla classe **Utility**, che viene descritto nell'apposito paragrafo LINK!!!!!!!, mentre i restanti condidati alla nuova generazione sono gli elementi stessi della generazione precedente (motivo?!?!?!). Come operatore di selezione, si è deciso di utilizzare la *selezione a roulette* dato che le soluzioni hanno lo stesso ordine di grandezza per quanto riguarda i loro costi QUI NON E' CHIARO CHE SIGNIFICA—INTENDI CHE LE FETTE CHE VENGONO FUORI NON SONO TROPPO DIVERSE??????. Infine la mutazione avviene con probabilità p_{mutex} pari all'1%. Quanto descritto viene per la maggior parte gestito attraverso il metodo NOME!!!!!!!!!!!!! con l'ausilio della classe **pathgenetic**, procediamo quindi alla loro esposizione.

CLASSE PATHGENETIC

La classe **PathGenetic** utilizzata per memorizzare i dati di una soluzione generica, estende **PathStandard** già discussa nel paragrafo X.Y. aggiungendo due campi utili solamente per gli algoritmi genetici: il primo di tipo *double* memorizza la fitness associata alla soluzione, il secondo di tipo intero identifica il circuito all'atto dell'estrazione dei percorsi che formeranno la generazione successiva[42]. La classe è dotata del metodo privato **CalculateFitness** il quale semplicemente setta la variabile fitness come descritto in precedenza:

```
private void CalculateFitness()
{
     fitness = 1 / cost;
}
```

La variabile **cost** e l'array **path** sono ereditati da PathStandard e vengono settati utilizzando uno dei tre costruttori a disposizione

```
public PathGenetic(int[] path, double cost) : base()
{
     this.path = path;
```

[41] Per qualsiasi generazione non sono quindi accettabili elementi non validi per il problema in questione. La funzione XXXX LINK!!! è stata realizzata a a tale scopo.

[42] I suddetti parametri prendono nome **fittness** e **nRoulette**

```
        this.cost = cost;
        CalculateFitness();
        nRoulette = -1;
}

public PathGenetic(int[] path, Instance inst) : base(path, inst)
{
    CalculateFitness();
    nRoulette = -1;
}


public PathGenetic(): base()
{
    fitness = -1;
    nRoulette = -1;
}

ANDREBBE UN COMMENTINO SU OGNUNO
```

FUNZIONE GENETICALGORITHM

Questa classe ha il compito di amministrare tutto tutte le fasi dell'algoritmo genetico. Per prima cosa si procede con una dichiariazione ed inizializzazione delle varie strutture dati necessarie. Tra queste troviamo due liste di *PathGenetic* chiamate **OriginallyPopulated** e **ChildPoulation** che, durante tutto il processo, contengono rispettivamente l'insieme dei circuiti hamiltoniani che compongono la generazione $i - esima$ ed i figli da loro generati. La prima generazione viene ottenuta attraverso il metodo **NearestNeightborGenetic**, discusso in maggiore dettagli nella appendice LINK!!!!!, che, ricevendo la lista completa per ogni nodo di quali sono ad esso più vicini[43], costruisce le varie instanze tendendo a collegare i nodi più vicini tra loro[44]. Durante la generazione dei nuovi figli, poiché l'indice in cui è memorizzato un circuito all'interno di *OriginallyPopulated*, dovuto in genere all'estrazione della rouellette, è casuale[45] si è deciso di accoppiare circuiti memorizzati in celle adiacenti; la casualità di OriginallyPopulated consente di combinare fra loro soluzioni buone con altre meno buone e ciò statisticamente risulta particolarmente vantaggioso. Una volta prodotti i figli è necessario procedere con la creazione della nuova generazione padre: questo viene eseguito dal metodo **NextPopulation** descritto in seguito. Per identificare il miglior circuito della generazione corrente si è realizzato il metodo **BestSolution**, qualora il valore ottenuto risulti minore dell'incumbent[46] quest'ultimo viene aggiornato. L'algoritmo termina quanto scade il time limit fornito dall' utente. Riportiamo di seguito il codice realizzato:

[43] Funzione già descritta qui LINK!!!!!!!!!!!!!!.

[44] Il numero di componenti per ogni generazione viene chiesto in input all'utente e passato come parametro di ingresso alla funzione *GeneticAlgorithm*.

[45] Con casuale si intende che non è presente alcuna forma di correlazione fra l'indice e il costo della soluzione.

[46] Con incumbent si intende la miglio soluzione fin ora calcolata.

```
PathGenetic incumbentSol = new PathGenetic();
PathGenetic currentBestPath = null;

List<PathGenetic> OriginallyPopulated = new List<PathGenetic>();
List<PathGenetic> ChildPoulation = new List<PathGenetic>();

List<int>[] listArray = Utility.BuildSLComplete(instance);

//Generate the first population
for (int i = 0; i < sizePopulation; i++)
OriginallyPopulated.Add(Utility.NearestNeightborGenetic(instance, rnd,
   true, listArray));
do
{
    //Generate the children
    for (int i = 0; i < sizePopulation; i++)
    {
        if (i % 2 != 0)
            ChildPoulation.Add(Utility.GenerateChild(instance, rnd,
               OriginallyPopulated[i], OriginallyPopulated[i - 1],
               listArray));
    }

    OriginallyPopulated = Utility.NextPopulation(instance,
       sizePopulation, OriginallyPopulated, ChildPoulation);

    //currentBestPath contains the best path of the current population
    currentBestPath = Utility.BestSolution(OriginallyPopulated,
       incumbentSol);

    if (currentBestPath.cost < incumbentSol.cost)
    {
        incumbentSol = (PathGenetic)currentBestPath.Clone();
        Utility.PrintGeneticSolution(instance, process, incumbentSol);
    }

    // We empty the list that contain the child
    ChildPoulation.RemoveRange(0, ChildPoulation.Count);

} while (clock.ElapsedMilliseconds / 1000.0 < instance.TimeLimit);

Console.WriteLine("Best distance found within the timelit is: " +
   incumbentSol.cost);
```

NEARESTNEIGHTBORGENETIC

Il corrente metodo è simile alla funzione NearestNeight discussa nel paragrafo X.Y ma con due differenze significative:

- La sequenza degli elementi nell array che codifica il percorso creato dal metodo indicano l'ordine con cui il circuito visita i nodi. Si ricorda invece che il metodo NearestNeight produce percorsi codificati in array in cui alla generica posizione **i** è collocato il nodo successivo al nodo **i**. Tale modifica è dettata solamente da una agevolazione nell'utilizzo successivo di queste informazioni da parte dell'algoritmo genetico nella creazione della prima generazione.
- Poiché un algoritmo genetico è tanto migliore quanto gli individui che formano la popolazione di partenza hanno caratteristiche dissimili fra di loro, si è fatto in modo che i circuiti fossero il più possibili diversi gli uni dagli altri.

L' intestazione del metodo risulta essere:

```
public static PathGenetic NearestNeightborGenetic(Instance instance,
    Random rnd, bool rndStartPoint, List<int>[] listArray)
```

Dove:

- **instance**: oggetto della classe *Instance* contenente tutti i dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' utente;
- **rnd**: istanza della classe *Random* precedentemente inizializzato con un seme random diverso per ogni iterazione del programma;
- **rndStartPoint**: variabile booleana che determina se il nodo di partenza sul quale viene applicato l' algoritmo nearest neightbor risulta essere casuale (in tal caso assume il valore true) oppure sia il nodo di default 0;
- **listArray**: lista in cui all'indice **i** è presente un vettore di dimensione instance.Nnodes al cui interno sono, in ordine crescente rispetto alla distanza assunta dal nodo **i**, presenti gli indici associati ai nodi del grafo.

Il circuito prodotto dal metodo viene memorizzato all'interno del vettore **heuristicSolution** avente una dimensione pari al numero di nodi del grafo. Poiché si vogliono soluzioni che siano il più possibile dissimili tra loro, è consigliabile fare in modo che *rndStartPoint* sia posta a *true*[47]. Il vettore **VisitedNodes** di tipo bool è un vettore di supporto che memorizza all'indice **i** il valore logico true se il nodo **i** è già stato visitato, false altrimenti.

```
// heuristicSolution is the path of the current heuristic solution
    generate
    int[] heuristicSolution = new int[instance.NNodes];

    bool[] VisitedNodes = new bool[instance.NNodes];

    int firstNode = 0;

    //rndStartPoint define if the starting point is random or always the
       node 0
```

[47]Da notare che tale parametro non è settata runtime ma solamente via hardcode.

```
    if (rndStartPoint)
        firstNode = rnd.Next(0, instance.NNodes);

    heuristicSolution[0] = firstNode;
    VisitedNodes[firstNode] = true;
```

Una volta definito il nodo di partenza i restanti nodi, che se visitati rispettando il loro ordine compongono un circuito hamiltoniano, son ottenuti attraverso un ciclo *for*: alla generica iterazione **i** del ciclo, sfruttando la struttura dati listArray e la funzione **RndGenetic** si memorizza all'interno della variabile **nextNode** il nodo successivo visitato dal percorso sempre che questo sia ancora disponibile. Per verificarne la disponibilità si utilizza l'array *VisitedNodes*, qualora non sia possibile utilizzare tale nodo si passa al successivo più vicino.
(la variabile contatore de ciclo for è inizializzata al valore 1, quindi il heuristicSolution[i-1] è memorizzato l' ultimo nodo visitato) QUESTO VA COME COMMENTO NEL CODICE

```
for (int i = 1; i < instance.NNodes; i++)
{
     bool found = false;
     int candPos = RndGenetic(rnd);
     int nextNode = listArray[heuristicSolution[i - 1]][candPos];
      do
      {
         //We control that the selected node has never been visited
          if (VisitedNodes[nextNode] == false)
          {
              VisitedNodes[nextNode] = true;
              heuristicSolution[i] = nextNode;
              found = true;
          }
          else
          {
              candPos++;
              if (candPos >= instance.NNodes - 1)
              {
                  nextNode = listArray[heuristicSolution[i - 1]][0];
                  candPos = 0;
              }
              else
                  nextNode = listArray[heuristicSolution[i - 1]][candPos];
          }

     } while (!found);
}
```

GENERATECHILD

Per generare un figlio si è realizzato il metodo **GenerateChild**, appartenente alla classe Utility,

avente la seguente intestazione:

```
public static PathGenetic GenerateChild(Instance instance, Random rnd,
    PathGenetic mother, PathGenetic father, List<int>[] listArray)
```

Dove:

- **instance**: oggetto della classe Instance contenente tutti i dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' utente;
- **rnd**: istanza della classe *Random* precedentemente inizializzato con un seme random diverso per ogni iterazione del programma;
- **father**: circuito hamiltoniamo che sarà accopiato con il parametro mather;
- **mother**: hamiltoniamo che sarà accopiato con il parametro father;
- **listArray**: lista in cui all' indice i è presente un vettore di dimensione instance.Nnodes al cui interno sono, in ordine crescente rispetto alla distanza assunta dal nodo i, presenti gli indici associati ai nodi del grafo.

Come precedentemente accennato il seguente metodo produce un figlio utilizzando l'operatore di crossover a singolo punto.

```
int crossover = (rnd.Next(0, instance.NNodes));

for (int i = 0; i < instance.NNodes; i++)
{
    if (i > crossover)
        pathChild[i] = mother.path[i];
    else
        pathChild[i] = father.path[i];
}
```

Una volta creato il figlio, con una probabilità p = 0.01 viene effettuata su di esso una mutazione utilizzando il metodo **Mutation** LINK?!!?!?.

```
if (rnd.Next(0, 101) == 100)
        Mutation(instance, rnd, pathChild);
```

Il figlio ottenuto quasi certamente non risulta essere un circuito ammissibile, per tale motivo si è progettato il metodo **Repair**. É interessante far notare che quest'ultimo non cerca di modificare i circuiti non modo da abbassarne il più possibile il costo ma al contrario è stato costruito in modo tale da risultare il più veloce possibile: soprattutto per popolazioni numerose e time limit alti tale scelta risulta essenziale. Di conseguenza i figli così prodotti sono tipicamente caratterizzati da un costo che con bassa probabilità risulta migliore rispetto a quello dei genitori e ciò comporta una saturazione dell'algoritmo dopo poche iterazioni (PERCHè?!?!?!?! METTERLO COME FOOTNOTE). Come operazione finale si è quindi deciso di applicare su di essi l'algoritmo **TwoOpt** ma solamente con una probabilità $p = 1/(n/2)$, dove n è il numero dei nodi. La ragione per cui p assume tale valore è

dovuta al fatto che l'operazione di due ottimalità ha complessità computazionale O()[quando faccio il multistart controllo meglio quanto è la sua complessità] ed una sua applicazione più frequente rallenterebbe troppo l'algoritmo.

```
if (ProbabilityTwoOpt(instance, rnd) == 1)
{
      child.path = InterfaceForTwoOpt(child.path);
      TSP.TwoOpt(instance, child);
      child.path = Reverse(child.path);
}
```

Da notare che la funzione per ottenere la due ottimalità è utilizzata da più metodi di risoluzione (multi-start ecc..) e necessita che il percorso della soluzione hamiltoniana sia memorizzato con un apposito formato diverso da quello presentato per l'algoritmo genetico: sono quindi necessarie due semplici interfacce **InterfaceForTwoOpt, Reverse**, i cui tempi di esecuzione sono $O(n)$.

NEXTPOPULATION

La funzione **NextPopulation**, appartenente alla classe **Utility**, consente di definire la nuova generazione scegliendone gli elementi tra la vecchia generazione ed i suoi figli attraverso una estrazione a roulette. La firma di tale funzione risulta essere:

```
public static List<PathGenetic> NextPopulation(Instance instance, int
    sizePopulation, List<PathGenetic> FatherGeneration, List<PathGenetic>
    ChildGeneration)
```

Dove:

- **instance**: oggetto della classe Instance contenente tutti i dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' utente;
- **sizePopulation**: parametro che indica di quanti elementi deve essere la nuova generazione, per come è stato costruito il nostro programma questo parametro non varia mai ed è richiesto una sola volta all'utente;
- **FatherGeneration**: Lista contenente i circuiti che definiscono la generazione corrente;
- **ChildGeneration**: Lista contenente i circuiti figli generati da FatherGeneration utilizzando la funzione GenerateChild.

Per prima cosa uniamo le due liste di circuiti in quanto si è deciso che l'unico metro di giudizio durante la selezione deve essere il valore di fitness attribuito ad ogni soluzione indipendentemente dalla loro provenienza.

```
for (int i = 0; i < ChildGeneration.Count; i++)
    FatherGeneration.Add(ChildGeneration[i]);
```

Passiamo ora a descrivere come è gestita la selezione a roulette, chiaramente esistono molteplici metodi e quello da noi scelto non ha alcun vantaggio significativo rispetto agli altri. L'idea alla base dell'algoritmo è di assegnare un valore univoco ad ogni circuito estraibile e di utilizzare tali valori molteplici volte come caselle della roulette, implementata come una lista di interi. Il numero di inserimenti per ogni valore è direttamente proporzionale alla fitness del circuito a cui è associato. Tutto questo viene per la maggior parte gestito all'interno del metodo **FillRoulette** LINK!!!!!!!!!!!!!!!!!, sempre definito nella classe **Utility**, che restituisce inoltre la grandezza della roulette così creata e poi memorizzato nella variabile **upperExtremity**:

```
List<PathGenetic> nextGeneration = new List<PathGenetic>();
List<int> roulette = new List<int>();
Random rouletteValue = new Random();
int upperExtremity = FillRoulette(roulette, FatherGeneration);
```

La creazione della nuova generazione avviene estraendo valori random non superiori ad *upperExtremity*, questi forniscono gli indici della lista-roulette le cui posizioni indicano indirettamente quale circuito deve far parte della nuova generazione. Naturalmente è possibile estrarre più volte la stessa soluzione, in questo caso la cosa va ignorata ripetendo nuovamente il processo. Nel complesso si dovranno estrarre (*sizePopulation*) circuiti hamiltoniani.

```
List<int> NumbersExtracted = new List<int>();
bool find = false;
int numberExtracted;

for (int i = 0; i < instance.SizePopulation; i++)
{
    do
    {
        find = true;
        numberExtracted = rouletteValue.Next(0, upperExtremity);
        //A path can't be extracted more than one time
        if (NumbersExtracted.Contains(roulette[numberExtracted]) == false)
        {
            find = false;
            NumbersExtracted.Add(roulette[numberExtracted]);
            nextGeneration.Add(FatherGeneration.Find(x => x.NRoulette ==
                roulette[numberExtracted]));
        }
  } while (find);
}
return nextGeneration;
```

APPENDICE

FILL ROULETTE

Il metodo FillRoulette ha il compito di popolare la roulette in modo tale che la selezione sia proporzionale alla fitness. Associa ad ogni circuito un numero intero, chiamato **NRoulette**, progressivo e inserisce all'interno della roulette tale valore un numero di volte proporzionale al valore della fitness del circuito, infine ritorna la dimensione della roulette. La sua firma risulta essere:

```
static int FillRoulette(List<int> roulette, List<PathGenetic>
    CurrentGeneration)
```

- **roulette**: Lista di interi che rappresenta la roulette e che viene popolata dal metodo;
- **CurrentGeneration**: Lista contenente i circuiti candidati a far parte della nuova generazione;

Utilizzando il metodo **Estimate** LINK!!!!!!!!!!!! si ottiene una costante intera che viene memorizzata all'interno della variabile **proportionalityConstant**: moltiplicare questo valore per la fitness di un circuito ci dice quante volte il corrispondente *NRoulette* associato vada inserito nella ruolette. Poiché all'interno della stessa generazione i valori della fitness non variano per ordini di grandezza, tale costante viene per convenzione calcolata utilizzando il circuito memorizzato all'indice 0 in *CurrentGeneration*.

```
int sizeRoulette = 0;

int proportionalityConstant = Estimate(CurrentGeneration[0].Fitness);

for (int i = 0; i < CurrentGeneration.Count; i++)
{
    int prob = (int)(CurrentGeneration[i].Fitness *
        proportionalityConstant);
    CurrentGeneration[i].NRoulette = i;
    sizeRoulette += prob;

    for (int j = 0; j < prob; j++)
      roulette.Add(i);
}
return sizeRoulette;
```

ESTIMATE

Il metodo Estimate genera una costante di proporzionalità in modo tale che, eseguendo il prodotto fra tale costante ed il valore ricevuto come parametro di ingresso, si ottenga una quantità sempre maggiore di 100 PERCHE'??????? A COSA SERVE?????. La sua firma risulta essere:

```
static int Estimate(double sample)
{
    int k = 1;
    while (sample*k < 100)
    {
        k = k * 10;
    }
    return k;
}
```

Dove:

- **sample**: Valore della fitness presa come campione.

REPAIR

Il metodo Repair è stato progettato per trasformare un percorso che non risulta essere un circuito hamiltoniano in un circuito hamiltoniano. Sappiamo che visitare più volte lo stesso nodo rende tale proprietà non vera così come la presenza di almeno un nodo isolato. L'algoritmo si sviluppa in due fasi: in un primo momento si eliminano dal vettore che codifica il percordo tutti i nodi duplicati, successivamente si fa in modo che quelli isolati vengano connessi al nodo ad essi più vicini. Un esempio di funzionamento dell'algoritmo è riportato nei tre grafici sottostanti.

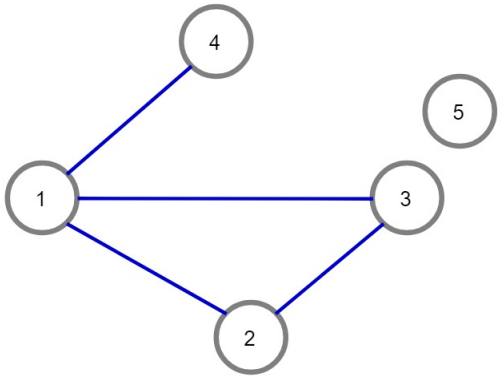

Figura 1: Circuito non hamiltoniano figlio

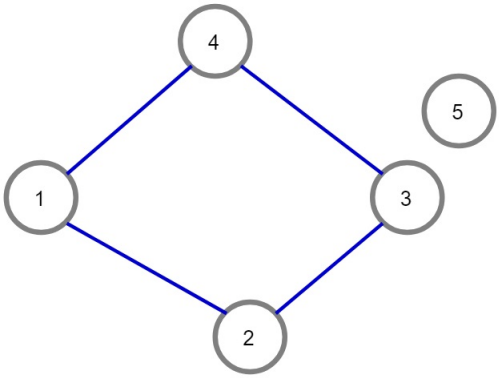

Figura 2: Circuito hamiltoniano incompleto

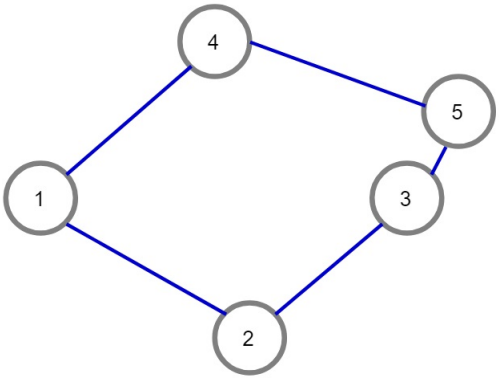

Figura 3: Circuito hamiltoniano figlio

Discutiamo ora il codice prodotto. Costruiamo due liste di interi chiamate **isolatedNodes** e **nearlestIsolatedNodes** dove rispettivamente contengono all'i-esimo elemento l'indice dell'i-esimo nodo isolato e l' indice del nodo ad esso più vicino[48]. Per popolare tali liste utilizziamo i metodi **FindIsolatedNodes** LINK!!!!!! e **FindNearestNode** LINK!!!!!!!!!. Dichiariamo le due liste **pathIncomplete** e **pathComplete**, nella prima andiamo ad inserire il percorso originale privo degli elementi che lo rendono non hamiltoniano mentre nella seconda costruiamo il percorso completo di tutti i nodi ed hamiltoniano. Per ottenere quest'ultimo risultato procediamo ciclicamente con la copia di ogni elemento appartente a *pathIncomplete* in **pathComplete** facendo però in modo che, se per un qualsisi i, j, m vale $pathIncomplete[i] = nearlestIsolatedNodes[j]$, allora nel caso andiamo a porre $pathComplete[m] = pathIncomplete[i]$ dovrà anche valere $pathComplete[m+1] = isolatedNodes[j]$ e $pathComplete[m+2] = pathIncomplete[i+1]$ simili a quanto avviene nella Figura 3!!!!!!!!!!!!!!!!!!!!!!!!.

```
int positionInsertNode = 0;

for (int i = 0; i < pathIncomplete.Count; i++)
{
    if (nearlestIsolatedNodes.Contains(pathIncomplete[i]))
    {
        pathComplete[positionInsertNode] = pathIncomplete[i];
```

[48]Ci sono in realtà delle eccezzioni a quest'ultima affermazione che vedremo in seguito

```
            pathComplete[positionInsertNode + 1] =
                isolatedNodes[nearlestIsolatedNodes.IndexOf(pathIncomplete[i])];
            positionInsertNode++;
        }
        else
            pathComplete[positionInsertNode] = pathIncomplete[i];

        positionInsertNode++;
    }
    return new PathGenetic(pathComplete, instance);
```

FINDISOLATEDNODES

Tale funzione viene utilizzata per identificare tutti i nodi isolati presenti in un generico percorso. La sua firma risulta essere:

```
static void FindIsolatedNodes(Instance instance, int[] path, List<int>
    isolatedNodes)
```

- **instance**: oggetto della classe Instance contenente tutti i dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' utente;
- **isolatedNodes**: Lista contenente gli indici di tutti i nodi isolati;

Data la semplicità del metodo non si ritiene utile far nessuna considerazione, riportiamo direttamente il codice realizzato.

```
bool nodeIsVisited = false;

for (int i = 0; i < instance.NNodes; i++)
{
    for (int j = 0; j < instance.NNodes; j++)
    {
         if (pathChild[j] == i)
         {
            nodeIsVisited = true;
            //If the node is visited can exit to for
            break;
         }
    }

    //If the node has nevere been visited is a isolated noode
    if (nodeIsVisited == false)
       isolatedNodes.Add(i);

    //Configure nodeIsVisited to its default value
    nodeIsVisited = false;
```

```
}
```

FINDNEARESTNODE

Dato un certo circuito ed una lista di nodi isolati in esso contenuti, il metodo FindNearestNode fornisce per ognuno di essi il nodo più vicino *valido* ossia che rispetti le seguenti condizioni:

- Non deve essere un nodo isolato;
- Non deve essere il nodo più vicino di un nodo isolato precedentemente analizzato. In altre parole se n_3 è un nodo non isolato e risulta il nodo più vicino dei nodi isolati n_1 e n_2 non è possibile avere: $nearestNeighIsolNode[indice_{n_1}] = n_3 \wedge nearestNeighIsolNode[indicen_2] = n_3$. Per convenzione, supponendo $indice_{n_1} < indice_{n_2}$ vale $nearestNeighIsolNode[indice_{n_1}] = n_3$ mentre a $nearestNeighIsolNode[indice_{n_2}]$ viene assegnato il successivo nodo valido più vicino disponibile.

Il codice commentato della funzione viene riportato di seguito.
COMMENTA CODICE!!!!!!!!!!!!!!!!!!!!!!!! RIPORTA TUTTO IL METODO!!!!!!!!!!!!!!!! PUBLIC ...

```
int nextNode = 0;
int nearestNode = 0;
bool find = true;

for (int i = 0; i < isolatedNodes.Count; i++)
{
    find = false;
    nextNode = 0;
    do
    {
       nearestNode = listArray[isolatedNodes[i]][nextNode];

       if (((isolatedNodes.Contains(nearestNode)) == false) &&
          (nearestNeighIsolNode.Contains(nearestNode) == false))
       {
          nearestNeighIsolNode.Add(nearestNode);
          find = false;
       }
       else
       {
           nextNode++;
           find = true;
       }
    } while (find);
}
```

BESTSOLUTION

Il metodo BestSolution riceve come input una serie di percorsi hamiltoniani memorizzati attraverso la classe **PathGenetic** LINK!!!!!!!!!!!!!!!!!!! ed in output fornisce quello a costo minore, la sua intestazione risulta essere:

```
public static PathGenetic BestSolution(List<PathGenetic> population)
```

Dove:

- **population**: Insieme di cammini hamiltoniani;

Di seguito il codice:

```
PathGenetic currentBestPath = population[0];

for (int i = 1; i < population.Count; i++)
{
   if (population[i].cost < currentBestPath.cost)
      currentBestPath = population[i];
}

return currentBestPath;
```