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 bandÃň 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, ChvÃątal, 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, Chvàtal, 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 in 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.
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 poiché è sempre possibile sommare a tutti i costi una costante sufficientemente elevata senza alterare 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, poiché 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 incidono 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 poiché è sprovvisto dei vincoli di subtour che impediscono alla soluzione ottima di avere dei subtour al suo interno.

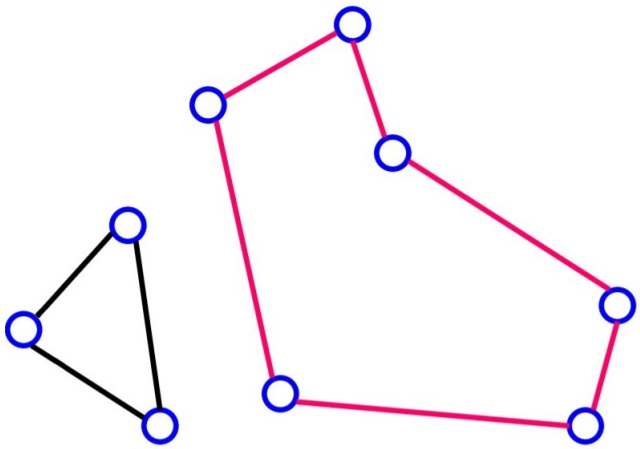

Figura 1: Soluzione con due subtour

Una possibile formulazione per i vincoli di subtour 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: poiché tutti i subtour violano tale vincolo la soluzione ottima non potrà contenere subtour al suo interno. Essendo il numero di vincoli di subtour pari al numero di sottoinsiemi S distinti, il numero di tali vincoli risulta esponenziale rispetto alla dimensione dell' istanza. In particolare il numero di sottoinsiemi S distinti che si possono formare con n nodi è 2^n: questo perchè associando un bit per ogni vertice (il cui valore definisce se appartiene o meno al sottoinsieme) un sottoinsieme S risulta identificato da una sequenza di n bit e quindi si hanno 2^n possibili sottoinsimi S distinti. Dato che si è imposto che il vertice 1 appartiene sempre ad ogni S e che $S \subsetneq V$, si ha che il numero di vincoli di subtour risultano pari a $2^{n-1} - 1$.
Una seconda formulazione equivalente per esprimere i vincoli di subtour è la seguente:

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

I vincoli di subtour non sono stati aggiunti al modello per via del loro numero: sarà nostra cura progettare un opportuno separatore, ossia una funzione che fornita in ingresso una soluzione x* ottima per il modello corrente generi tutti i vincoli di subtour che verranno aggiunti "al volo" al modello.

LETTURA FILE DI INPUT

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

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

Tali istanze, memorizzate ciascuna in un file di testo, hanno un formato ben preciso grazie al quale è stato possibile progettare un opportuno parser al fine di popolare le strutture dati interne del programma con i dati presenti in tale file. Conoscendo preventivamente la struttura del file si sono, in fase di progettazione, limitati i controlli inerenti a possibili modifiche riguardanti la struttura stessa del file, supponendo che l' utente fornisca in input solo file aventi la struttura nativa.

STRUTTURA DEL PROGETTO

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

- Src contiene i codici sorgenti scritti in C# e l'eseguibile;
- Data include le istanze del problema del commesso viaggiatore appartenenti alla TSPlib;
- Concorde la quale contiene i file sorgenti in linguaggio C del programma Concorde la cui trattazione è rimandata al Capitolo X.

Il software sviluppato è composto da dieci classi, riportiamo di seguito il nominativo di ogni classe:

- Instances
- ItemList
- PathGenetic
- PathStandard
- Point
- Program
- 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 poiché una loro trattazione risulta in questo momento prematura.

CLASSE POINT

La classe Point è stata realizzate al fine di memorizzare le coordinate di un nodo appartenente all' istanza del problema del Commesso Viaggiatore fornito in ingresso al programma. Tale classe presenta due variabili private di tipo double chiamate rispettivamente x e y necessarie rispettivamente per memorizzare la coordinata x e la coordinata y del nodo. Il costruttore della classe semplicemente setta queste due variabili ai valori del nodo letto dal file di input.
Il metodo Distance è l' unico metodo appartenente a questa classe ed è stato implementato per calcolare il costo del lato congiungente due vertici del grafo; la sua firma è:

```
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 realizzato nel caso in cui pointType risulti uguale a EUC_2D, in questo caso il costo del lato risulta 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 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	?
double	zBest	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.
double	dMin	?
double	dMax	?

Anzichè definire xMin/yMin e xMax/yMax definirei dMin e dMax. Se preferisci tenerle separate non ci sono problemi :)Il codice comunque non lo ho modificato.
In piu sarebbe piu giusto chiamare zBest xBest perche ascoltando le reg ho capito che la z ci andava nel suo esempio non nel nostro..
mi sono accorto anche che sizePopulation non centra davvero nulla con inst e quindi la toglierei da qui
Questa classe è risultata particolarmente utile poiché, passando come parametro una sua istanza ad un metodo, quest' ultimo ha accesso a tutte le variabili elencate nella tabella. Non avere questa classe significava dover passare molti più parametri alle funzioni rendendo il codice meno chiaro. L' unico metodo appartenente a questa classe risulta essere il metodo 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 probelma del Commesso Viaggiatore fornita in ingresso dall' utente;

Tale metodo semplicemte stampa a video le coordinate di tutti i nodi contenuti nell' array Point dell' oggetto inst. Per scrupolo 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 TSP

La classe TSP contiene il metodo TSPOpt invocato dal metodo Main e tutti gli algoritmi di risoluzione realizzati. In particolare il metodo TSPOpt realizza l'interfaccia grafica che consente all' utente di invocare l' algoritmo risolutore desiderato; nel caso in cui non si siano registrate anomalie, la funzione ritorna il valore true, viceversa false. *[In realtà da noi ritorna sempre true..sarebbe*

meglio tornasse false in qualche caso]
se dici questo ÃÍ anche un buon punto per dire due parole sull' interfaccia grafica creata..

CLASSE UTILITY

La classe Utility comprende al suo interno il codice di tutti i metodi invocati dagli algoritmi risolutori presenti nella classe TSP: si è deciso di dirottare in questa classe tali funzioni per avere nella classe TSP un codice più leggibile.

CLASSE PROGRAM

La classe Program contiene il metodo Main che, come noto, rappresenta il punto di inizio del programma: dal sistema operativo riceve in input l' array argv di stringhe contenente i parametri inseriti dall' utente da riga di comando. Appartengono a questa classe anche i metodi ParseInst e Populate; tali metodi consentono di popolare i campi associati all' istanza (che come vedremo è stata chiamata inst) della classe Istance. Mentre ParseInst svolge il suo compito utilizzando i valori passati dall' utente da riga di comando, Populate utilizza i dati presenti nel file di testo contenente l' istanza del problema del Commesso Viaggiatore. Firma e implementazione di tali metodi è rimandata al successivo capitolo.
Nel metodo Main principalmente si crea una istanza della classe Instance e si invocano nell' ordine i metodi ParseInst, Populate e TSPOpt. Nel caso non si siano rivelate anomalie e l' utente abbia scelto di utilizzare un algoritmo esatto, viene stampato a video il tempo impiegato da Cplex per il calcolo della soluzione ottima; nel caso in cui l' utente abbia scelto un algoritmo euristico viene stampato il tempo impiegato da quest' ultimo per produrre la soluzione(che non necessariamente è quella ottima) fornita in output.*[Questa separazione non la facciamo ma secondo me è da fare perchè se usiamo un euristico e poi a video stampiamo "la soluzione ottima è stata trovata in..." è sbagliato]*. Si osserva che la stampa a video della soluzione ottima o della miglior soluzione trovata fino a quel momento nel caso di algoritmi euristici, viene effettuata all' interno dell' algoritmo risolutore.
Nella seguente tabella sono mostrate le variabili definite in questa classe assieme ad una breve descrizione. Poiché tali variabili sono costanti, è riportato pure il loro valore.

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	?

ci sarebbe pure clock, non so se ti sei dimenticato di toglierla o è giusto lasciarla globale

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. Per ottenere una maggiore flessibilità , si è deciso che il valore di un parametro inserito da riga di comando dovrà essere preceduto da una stringa identificatrice di quest' ultimo: in particolare si è deciso che il nome del file dovrà essere preceduto dalla stringa -file mentre il time limit dovrà essere preceduto dalla stringa -timelimit; questo permette di realizzare un codice che consente all' utente di inserire in un ordina arbitrario tali parametri essendo sempre cosciente di quale parametro ha settato. Si osserva che allo stato attuale del programma il vantaggio è relativo passando solo due parametri ma, nel caso di futuri aggiornamenti che prevedono l' inserimento di un numero maggiore di parametri da riga di comando, il vantaggio diventerebbe notevole. 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 un riferimento l'istanza della classe Instance contente tutti dati che descrivono l' istanza del probelma del Commesso Viaggiatore fornita in ingresso dall' utente;

- input: rappresenta un vettore contenente i parametri di input forniti da riga di comando dall' utente.

Il metodo è composto da un ciclo for il quale scandisce ad ogni iterazione tutte le celle dell'array input: se è presenta una uguaglianza fra l' elemento i dell' array e una stringa identificatrice si assegna ad una opportuna variabile di inst il relativo valore che si trova in input[i+1]. Successivamente si esegue l'istruzione continue in modo da interrompere il corrente ciclo per iniziare il successivo.

```
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

Come già accennato, il metodo Populate è stato implementato per memorizzare tutte le informazioni relative all' istanza del problema del Commesso Viaggiatore contenute nel file di testo nei rispettivi campi presenti in inst. Essendoci dati di natura diversa all' interno del file, quest' ultimo è diviso in sezioni che cominciano con una parola chiave. Tranne la sezione che inizia con la parola chiave NODE COORD SECTION, tutte le altre si sviluppano in una sola riga la quale ha la seguente struttura:

<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 puó 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. In ogni caso sarà nostra cura quando ci interfacciamo con l' utente far partire la numerazione dei nodi da 1.
    * 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)
```

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. Poiché si vuole leggere tutto il contenuto del file, è necessario invocare ReadLine() ciclicamente sull' oggetto sr creato finchè line risulta diversa da null. Quando line risulta pari a null significa che il file è terminato e quindi tutto il suo contenuto è stato letto.
Per quanto detto si è realizzato il seguente ciclo while per scandire tutte le righe del file:

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

Poiché 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. In caso in cui il confronto dia esito positivo, per prelevare il contenuto della sezione è neccessario 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 in base zero da cui iniziare l'eliminazione dei caratteri;
- count: numero di caratteri da eliminare.

Per quanto visto, risulta immediata la comprensione del codice neccssario per prelevare il contenuto della sezione e memorizzarlo in un generica variabile, chiamata campo:

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

Si ritiene utile osservare che è stato necessario incrementare il valore ritornato da meodo IndexOf di 2 poiché è presenta uno spazio fra il carattere ":" e il contenuto della sezione.
Nel caso in cui campo non è di tipo string è necessario effettuare una conversione esplicita utilizzando i metodi ToInt o ToDouble della classe Convert: questo in ogni caso non è un problema poiché noto il tipo di dato contenuto di una sezione.
Poiché il contenuto della sezione NODE COORD SECTION si sviluppa in più righe, si è dichiarata una variabile di tipo bool chiamata readingCoordinates inizializzata al valore logico false; a questa variabile viene assegnato il valore logico true solo una volta che si è entrati in questa sezione. Per leggere il contenuto di questa sezione si è realizzato un if la cui condizione valutava se readingCoordinates assumeva il valore logico true; in tal caso sono eseguite le seguenti istruzioni:

```
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.
- 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.
  - StringSplitOptions.None: indica che gli elementi dell' array ritornato possono essere stringhe vuote.

Nel nostro caso dato che l' unico separatore è lo spazio vuoto, è stato sufficiente passare come primo parametro a Split un array con un solo elemento contenente il carattere ' '. Successivamente all' interno di una variabile di tipo intero chiamata i si è memorizzato l' indice del nodo che, per quanto visto, è memorizzato sempre in elements[0] ed è un numero intero: per tale ragione è stato necessario applicare il metodo ToInt32 della classe Convert. Si è infine memorizzato nel vettore Coor di inst, in un oggetto Point,le coordinate del nodo presenti in elements[1] e elements[2]. Come accennato in precedenza, mentre la numerazione dei nodi nell' istanza parte da 1, la memorizzazione delle coordinate in Coor inizia dall' indice 0.
Anche in questo caso è stato necessario effettuare una conversione da string a double ; inoltre è stato doveroso utilizzare il metodo Replace della classe string avente la seguente firma:

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

dove:

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

Questo perchè il generale le coordinate dei vertici sono numeri reali aventi come carattere di separazione fra la parte intera e quella decimale il carattere '.' anzichè il carattere ',' : questo fatto genera anomalie nella conversione fra i tipi di dato string e double e quindi è stato necessario effettuare tale sostituzione di caratteri.

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. Premettiamo che 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 enviroment di Cplex utilizzando la funzione CPXopenCPLEX la quale ritorna un puntatore all' envairoment 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.

L' enviroment è una struttura dati interna a Cplex i cui dettagli implementativi non sono di dominio pubblico, contenente variabili inerenti all' esecuzione di Cplex. Per esempio variabili interne all' enviroment sono il time limit e la variabile verbose. Ad un enviroment possiamo associare uno o più modelli: per associare un modello è necessario usare il metodo 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, di variabili e di vincoli.

Per poter definire la funzione obbiettivo è necessario aggiungere tutte le variabili definendo per ciascuna il relativo costo; un primo metodo per far ciò è inserire contemporaneamente tutte le variabili utilizzando 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 contenente ogniuno il nome della variabile aggiunta al modello.

A livello implementativo è risultato più semplice anzichè aggiungere al modello tutte le variabili contemporaneamente, utilizzare un secondo approccio che consiste nell' inserire una variabile per volta invocando la funzione CPXnewcols tante volte quante sono le variabili da aggiungere. In particolare si è inserita la funzione CPXnewcols in una coppia di cicli for in cui l' iniializzazione del ciclo for esterno risulta essere:

```
int i = 0;
```

mentre l' inizzializzazione del ciclo for interno è:

```
int j = i + 1;
```

Questo perchè essendo il grafo completo, per ogni coppia di nodi esiste un lato [i,j] a cui si associa una variabile in Cplex. Poiché il grafo non è orientato, associando una variabile ad ogni lato avremmo due variabili che corrispondono allo stesso lato: per questo motivo si è scelto di associare una variabile ad un lato [i,j] solo se $i < j$.
Aggiungere una variabile alla volta significa però che i parametri obj, lb, ub e ctype che CPXnewcols si aspetta essere array, risultano variabili.
Questa problematica è stata risolta utilizzando l' operatore di indirizzo & prima del nome della variabile nell' invocazione della funzione simulando così una variabile come un array contenente un

solo elemento. La funzione CPXnewcols ritorna un intero che è pari a 0 se non sono state riscontrate anomalie durante l' inserimento delle variabili, un codice di errore viceversa.
Dopo quanto detto si ritiene utile mostrare come avviene l' invocazione del metodo CPXnewcols nel nostro caso particolare e i parametri ad esso forniti:

```
CPXnewcols(env, lp, 1, &obj, &zero, &ub, &binary, cname)
```

dove:

- env: puntatore all' enviroment di Cplex nel quale vuole essere inserito il modello.
- lp: puntatore al problema di programmazione lineare.
- 1: aggiungo una sola variabile
- obj: variabile double che memorizza al suo interno il costo del lato [i,j], il costo viene calcolato utilizzando il metodo dist già descritto in precedenza.
- zero: variabile di tipo double contenente il valore 0.0;
- ub: variabile di tipo double contenente il valore 1.0;
- binary: variabile di tipo char che assume il valore costante 'B';
- cname :

Dal momento che ad ogni lato del grafo è associata una variabile, è necessario realizzare una funzione chiamata xPos, la quale 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 xPos coincide con quello aspettato,in caso contrario viene sollevata una eccezione. Firma e dettagli implementativi di xPos saranno forniti nel paragrafo successivo.

```
if(CPXgetnumcols(env, lp) - 1 != xPos(i, j, inst))
   printError(" ... errata posizione per x");
```

Passa al paragrafo successivo
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 involta invocando CPXnewrows tante volte quante sono i vincoli da aggiungere.

COSTRUZIONE MODELLO E RELATIVA RISOLUZIONE 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; tale istanza è stata nominata 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 di tipo INumVar. Per creare una nuova variabile e aggiungerla al modello si invoca il metodo NumVar sull' oggetto cplex, esso ha come firma:

```
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.

Tale metodo ritorna la variabile creata che risulta necessario memorizzare in una struttura dati: questo fondamentalmente per due motivi:

- Molti metodi presenti nella libreria di Cplex richiedono come parametro l' oggetto che rappresenta la variabile del modello.
- Sull' oggetto che rappresenta la variabile è possibile invocare i metodi LB e UB che consentono di modificare il lower bound e l' upper bound della variabile; questi metodi sono utilizzati nei capitoli successivi.

La struttura dati utilizzata per memorizzare tutte le variabili è un array che è stato chiamato z.

Per definire la funzione obbiettivo è necessario utilizzare il metodo LinearNumExpr sull' oggetto cplex; tale metodo ritorna un oggetto di tipo ILinearNumExpr.

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

La variabile expr rappresenta una espressione lineare nella forma:

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

dove x_i sono variabili di tipo INumVar mentre c e a_i sono coefficienti 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.

Poiché expr rappresenta la funzione obbiettivo, coef rappresenta il costo associato alla variabile. A livello implementativo per creare una nuova variabile ed aggiungerla alla funzione obbiettivo si è realizzato il seguente codice:

```
//Populating objective function

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

  for (int j = i + 1; j < instance.NNodes; j++)
  {
      //zPos return 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, "z(" + (i + 1) +
        "," + (j + 1) + ")");
```

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

Creata l' espressione matematica è stata associata all' oggetto cplex specificando che essa rappresenta la funzione obbiettivo del modello: ciò è stato fatto utilizzando il metodo AddMinimize:

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

Passiamo ora ad aggiungere al modello gli n vincoli di grado, uno per ogni nodo. Per creare un vincolo è sempre necessario creare un oggetto di tipo ILinearNumExpr e ad esso aggiungere le variabili che partecipano al vincolo, solo successivamente si andrà ad aggiungere il vincoli al modello definendone il tipo e il termine noto. Supponendo di voler aggiungere il vincolo di grado per il nodo i, è necessario aggiungere all' espressione tutte le $n-1$ variabili corrispondenti ai lati che incidono in i. Per far ciò si è realizzato il seguente codice:

```
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 links (i,j) or (j,i) have coefficent 1
   //zPos return the correct position where link is stored inside the
      vector z

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

        ...
```

Infine per aggiungere un vincolo al modello ÃÍ necessario invocare sull' istanza cplex uno dei seguenti metodi a seconda del vincolo:

- AddEq: nel caso in cui il vincolo sia una equazione;
- AddLe:nel caso in cui il vincolo sia una disequazione avente segno $\leq$
- AddGe:nel caso in cui il vincolo sia una disequazione avente segno $\geq$

Poiché il vincolo di grado è una equazione è stato necessario utilizzare il metodo AddEq, riportiamo di seguito la firma di tale 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.

I metodi AddLe ed AddGe hanno la medesima intestazione.
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. BuilModel viene invocata in tutti gli algoritmi di risoluzione realizzati che prevedono l'utilizzo di Cplex. Tale funzione ha la seguente firma:

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

dove:

- cplex: oggetto sul quale si definirÃă 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.
- n: 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 stata definita la funzione obbiettivo, le variabili e i vincoli il metodo Solve:

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

Risolto il modello il costo della soluzione è memorizzato nella variabile double ObjValue accessibile tramite l' istanza della classe Cplex creata:

```
cplex.ObjValue;
```

Infine per ottenere il valore della soluzione ottima è necessario invocare sull' oggetto cplex la funzione GetValues la cui firma è:

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

dove:

- var: rappresenta il vattore contenente tutte le variabile appartenenti al modello.

Nel caso in cui si voglia anzichè il valore di tutte le variabili solo il valore di una certa variabile, ÃÍ necessario utilizzare il metodo GetValue e passare come parametro solamente la variabile di cui si desidera conoscere il valore.

METODO LOOP

Una volta illustrato come è possibile istanziare il modello di programmazione lineare intera ed ottenere la soluzione ottima calcolata da Cplex, è stato necessario sviluppare un opportuno algoritmo in grado di calcolare la soluzione ottima tenendo conto anche dei vincoli di subtour. Come già accennato, il modello fornito a Cplex non è completo e quindi la soluzione ritornata potrebbe al suo interno contenere dei subtour; questo fatto comporta che il tour calcolato possa non essere un circuito hamiltoniano e che quindi non sia una soluzione ammissibile per il problema del Commesso Viaggiatore. Il metodo Loop alla generica iterazione i, nel caso in cui la soluzione ritornata da Cplex contenga subtour al suo interno, aggiunge al modello corrente un vincolo per ogni subtour presente in tale soluzione; tali vincoli hanno la seguente forma:

$$\sum_{e \in E(S_k)} x_e \leq |S_k| - 1 \ , \ k = 1, \ldots, m \tag{6}$$

dove $S_k \subset V$ sono i vertici che caratterizzano il k-esimo subtour.
In questo modo, la successiva soluzione prodotta da Cplex può ancora avere dei subtour al suo interno, ma certamente non può avere i medesimi presenti nella soluzione precedente. Dopo t iterazioni del metodo Loop la soluzione ottima ritornata da Cplex sarà priva di subtour: questo significa che essa è un circuito hamiltoniano e quindi è la soluzione ottima che sarà fornita all' utente.
Per poter implementare il metodo Loop risulta evidente la necessità di sviluppare un' opportuna funzione in grado di individuare i subtour e creare i relativi vincoli; tale funzione è stata chiamata UpdateCC, inserita nella classe Utility, e sarà descritta nel paragrafo successivo.
Per ottenere una maggior chiarezza da un punto di vista logico, il vincolo di subtour viene creato all' interno del metodo UpdateCC ma viene aggiunto al modello nel metodo Loop. In particolare si sono utilizzate le strutture dati:

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

le quali sono popolate all' interno del metodo UpdateCC; in particolare all' interno di rcExpr[i] è memorizzata l' espressione contenente le variabili dell' i-esimo vincolo di subtour mentre in bufferCoeffRC[i] viene memorizzato il relativo termine noto. Una volta scanditi tutti i lati della soluzione ottima, se è presente almento un subtour il vincolo viene aggiunto utilizzando il metodo AddLe. A titolo illustrativo riportiamo il ciclo for che esegue ciò:

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

METODO UpdateCC

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].

La prima operazione prevede di confrontare le componenti connesse dei due nodi ricevuti in ingresso: se diverse significa che il lato corrente non genera subtour e quindi si procede ad aggiornare le componenti connesse dei vertici nell' array relatedComponents tramite il codice seguente:

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

}
relatedComponents[j] = relatedComponents[i];
```

Si osserva che si è deciso di assegnare a tutti i nodi appartenenti alla componente connessa del nodo j, il valore della componente connessa del nodo i, naturalmente si poteva fare equivalentemente il viceversa.

Nel caso in cui il lato crei un subtour è necessario aggiungere il vincolo (6) al modello: per far ciò si è creata la consueta variabile di tipo ILinearNumExpr a cui si sono aggiunte tutte le variabili associate ai lati congiungenti nodi aventi come componente connessa relatedComponents[i]. Quando detto è stato tradotto nel seguente codice:

```
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++;
      }
}
```

Come ultima operazione da compiere è stato necessario inserire nella lista rcExpr l' oggetto expr e nella lista bufferCoeffRC la variabile cnt.

CALLBACK

Una modalità avanzata di utilizzare Cplex prevede di agire all'interno del proprio algoritmo di Branch-and-cut; questo è reso possibile attraverso un meccanismo informatico che prende il nome di CallBack. Per ovvi motivi i codici sorgenti di Cplex non sono di libero accesso: è solo grazie alle CallBack che un programmatore esterno può interagire con il suo flusso esecutivo.
Cplex è stato progettato in modo tale che in alcune sue sezioni invochi delle funzioni chiamate proprio CallBacks. Esse di dafault non sono installate risultando cosÃň trasparenti a Cplex, nel caso in cui si provvede ad installarle nei punti di codice in cui vengono invocate il flusso di programma passa da Cplex alle CallBack. Tipicamente le CallBack eseguono elaborazioni che verranno inoltrate a Cplex il quale le utilizza durante l' evoluzione dell' algoritmo di Branch-and-cut. Tra le varie CallBack che Cplex mette a disposizione c' è ne sono due particolarmente importanti che saranno oggetto dei successivi paragrafi: LazyCallback e UserConstraintCallback.

LAZYCALLBACK

Durante il processo di risoluzione del branch-and-cut, ogni volta che Cplex risolvendo il rilassamento continuo in un nodo calcola una soluzione ammissibile il cui costo è inferiore rispetto all' incumbet attuale viene invocata la lazycontraintCallback prima di aggiornare l'incumbent. PoichÃľ , come più volte ricordato, al modello di partenza non sono stati aggiunti i vincoli di subtour è possibile che la soluzione calcolata da Cplex sia ammissibile per il modello corrente ma contenga dei subtour al suo interno. Il nostro obbiettivo è usare la lazycontraint callback per verificare se la soluzione calcolata sia priva di cicli e solo in questo caso aggiornare l' incumbent. Si osserva che a differenza del metodo Loop una volta che Cplex produce la soluzione ottima questa certamente è ammissibile e corrisponde quindi al tour ottimo che si vuole cercare. A livello implementativo sarÃă mostrato sia come installare la questa callBack in C# sia in c poichÃľ questo ci tornerÃă utile nel prosegui della tesi.