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 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 poiché 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 poiché è sprovvisto dei vincoli di subtour che impediscono soluzioni il cui grafo risulta non connesso.

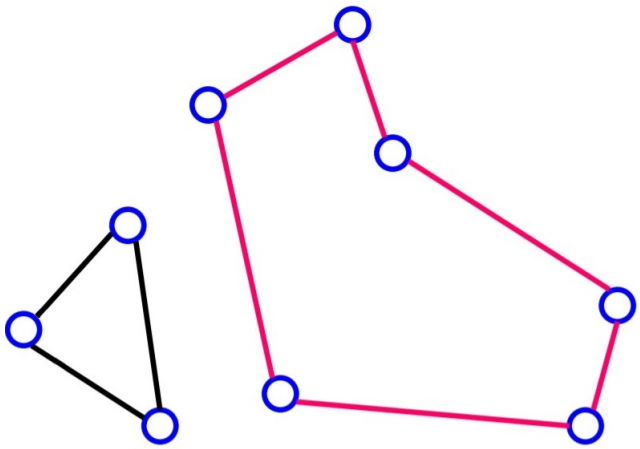

Figura 1: Soluzione con due subtour

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: poiché 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 ÃÍ quindi possibile definirne 2^n distinti. In realtÃă 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 perÃš non ÃÍ necessario utilizzare tutti i vincoli di subtour elimination, ÃÍ sufficiente considerarne un numero molto piÃź ridotto. Non potendo conoscere in anticipo quali siano quelli essenziali sarÃă 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 ÃÍ 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].

[2]Nessun altro tipo di input ÃÍ supportato

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
- 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 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:

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

```
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 piÃź. per xBest ok
L' unico metodo appartenente a questa classe, esclusi i vari getter e setter, ÃÍ **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 funzionalitÃă 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 attivitÃă:

- Creata una istanza della classe **Instance** ed invocati i due metodi precedentemente nominati.
- Creato un oggetto della classe **Stopwatch**. Questa classe ÃÍ fornita direttamente da Visual Studio appartenente al Namespace **System.Diagnostics** ÃÍ fornisce le funzionalitÃă 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 ÃÍ 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 ÃÍ 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 cosÃň detti ticks come unitÃă di misura per il tempo di calcolo, questa costante indica quanti ne trascorrono in un secondo.

CLASSE TSP

La classe **TSP** ÃÍ 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 giÃă specificato nella descrizione della classe Program, TSPOpt ÃÍ 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.

[4]Fatta eccezione per l'UserCutCallBack che ÃÍ gestita esternamente da una DLL

Entrando nello specifico per quanto riguarda gli elementi inizializzati troviamo un oggetto della classe **Cplex** che come giÃă accennato in precedenza ci permetterÃă 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 puÃš essere considerata come una libreria: contiene al suo interno solamente metodi **statici** che si ÃÍ deciso di raggruppare al suo interno per rendere il codice il piÃź 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 proprietÃă 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 ÃÍ gestita solamente la possibilitÃă 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 ÃÍ 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 rileverÃă utile 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**:

[5]Per maggiori dettagli si veda la sezione dedicata a GNUPlot

```
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 pressochÃĺ identica tra loro e cioÃĺ 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 é 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 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[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)
```

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** 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);
```

[6]Tale scelta é per mantenere una conformitá 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

```
        //Correct end of the file
        break;
        }

        ...
}
```

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. 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 giÃă 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 ÃÍ 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 ÃÍ 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 ÃÍ 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 ÃÍ 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; ÃÍ possibile procedere in piÃź modi, quello da noi scelto ÃÍ 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 semplicitÃă 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 sarÃă 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 ÃÍ privo di direzione. Si presenta quindi la necessitÃă di definire una convenzione per l'assegnazione del nome ai vari lati. La scelta adottata ÃÍ la seguente: considerando due generici nodi **i** e **j** allora il loro lato sarÃă 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 ÃÍ **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).
Una ulteriore considerazione necessaria é la seguente: come mostrato poco fa il metodo **CPXnewcols** si aspetta il passaggio di diversi array mentre noi vorremmo utilizzare semplici variabili. La soluzione é molto semplice e consiste nell'anteporre il carattere & prima di ogni variabile in questo modo stiamo in realtÃă 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';
```

[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 $i < j \& v < W$

```
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 ÃÍ 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 ÃÍ 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 ÃÍ 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 puÃš 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 funzionalitÃă offerta de quest'ultima.
Introduciamo ora una seconda interfaccia **ILinearNumExpr** che come si puÃš 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 é 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 é 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 é 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 poiché 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 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 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 é 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 é da noi altamente sconsigliato in quanto sperimentalmente é 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 pó di storia!!!!!! ————————————————————- L'idea alla sua base é molto semplice ed é 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 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

[10] Se non viene modificato di default risulta essere la classica console del progetto C#
[11] Da qui deriva il nome del metodo in quanto la soluzione consiste in un loop delle stesse operazioni

e di generarne gli opportuni vincoli per eliminarli.
In letteratura esistono molteplici modi per eseguire tali operazioni, quella da noi adottata si rifá 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 é 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** é presente una espressione sola significa che la soluzione attuale é 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 giÃă 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.
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 é 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 é 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].

[12] Nello specifico la parte di nostro interesse é quella che impedisce la formazione di piÃź 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**
[14] Per semplicitá si é deciso di identificare ogni **cc** con un valore intero univoco

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).

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 é 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 é deciso di inglobare la **cc** del nodo j in quella del nodo i.
Il secondo caso é 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.

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. CiÃš che ci apprestiamo ad illustrare ÃÍ sfruttale 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.