

## SEZIONE UNO

Questa tesi tratta lo studio di molteplici metodi risolutivi per il "Problema del commesso viaggiatore", indicato in lingua inglese con la sigla **TSP**<sup>1</sup>. Il testo segue di pari passo lo sviluppo di un software risolutivo da noi realizzato e pertanto, oltre a nozioni puramente teoriche, vengono anche presentati in maniera dettagliata sia l'ambiente di sviluppo utilizzato che il codice di programmazione.

A seguire è possibile trovare i risultati in entrambe le forme tabellari che grafiche di svariati test eseguiti con lo scopo di verificare la bontà di quanto prodotto.

Le caratteristiche del codice di programmazione, che risultano di minore interesse ai fini della tesi, sono riportate nella sezione finale **Appendice** del testo. D'altro canto ogni altra nozione algoritmica viene commentata nel dettaglio solamente durante il suo primo utilizzo così da non appesantire eccessivamente la lettura.

## INTRODUZIONE

Utilizziamo questa sezione introduttiva per parlare della storia, le applicazioni e correnti sfide riguardanti il *TSP*, da sempre uno dei problemi più discussi e studiati nella disciplina della **Ricerca Operativa** soprattutto grazie alla sua rilevanza in numerose applicazioni pratiche.

Il particolare nome *TSP* deriva dalla sua più tipica rappresentazione: data una rete di città e strade che le collegano, ci si chiede quale sia il percorso **migliore** che un commesso viaggiatore dovrebbe seguire nel caso in cui voglia visitare ogni città una ed una sola volta ritornando infine al punto di partenza. Per quanto detto, risulta naturale modellare questo problema attraverso un grafo **pesato** i cui nodi rappresentano le città mentre ed i lati corrispondono alle vie di accesso che le collegano. Per quanto riguarda il **peso** da attribuire a queste ultime è possibile utilizzare qualsivoglia tecnica: una semplice lunghezza fisica, tempi di percorrenza o la presenza di pedaggi sono solo alcuni esempi. Data questa situazione possiamo notare come sia molto facile mutarla in infinite altre, sia di uso pratico che teorico. Fondamentali sono le applicazioni nell'ambito della logistica distributiva più comunemente note come problemi di routing. Non ci riferisce solamente al classico smistamento di pacchetti in una rete internet ma anche l'organizzazione di sistemi di distribuzione per beni e servizi: movimentazione di pezzi o semilavorati tra reparti di produzione, raccolta e distribuzione di materiali, smistamento di merci da centri di produzione a quelli di distribuzione e molto altro ancora.

Sebbene le applicazioni nel contesto dei trasporti rimangano comunque le più naturali, la semplicità del *modello* ha portato negli anni allo sviluppo di innumerevoli applicazioni nelle più svariate aree di interesse. Un esempio può essere la programmazione di una macchina per eseguire fori in un circuito<sup>2</sup>

Per inquadrare l'evoluzione temporale del *TSP* è necessario risalire fino all'Ottocento e quindi ai matematici Sir William Rowan Hamilton e Thomas Penyngton. Il primo, ad esempio, durante l'anno 1857 nella città di Dublino e più specificatamente nel corso di una riunione della British Association for the Advancement of Science, descrisse il così detto "Icosian game": questo 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 era originariamente composto da 12 vertici, 30 spigoli e 20 facce identiche a forma di triangolo equilatero.

---

<sup>1</sup>Travelling Salesman Problem.

<sup>2</sup>In questo esempio i fori da forare sono le città, le strade sono definite dai possibili movimenti della macchina ed il loro peso è il tempo necessario per spostare la testa del trapano.

Questo problema è in realtà un particolarissimo 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 veder nascere la formulazione più generale del TSP, proposta all'inizio di questa sezione, è necessario attendere il secolo successivo, negli anni Venti e Trenta per la precisione, con il matematico ed economista Karl Menger. Ciò non significa che furono contemporaneamente proposti anche metodi risolutivi *intelligenti*: per molto tempo non si ebbe altra idea che quella di generare e valutare tutte le soluzioni possibili, mantenendo il problema praticamente insolubile. Il numero totale dei differenti percorsi possibili attraverso  $n$  città è infatti esorbitante: dato un qualsiasi punto di partenza, ci sono a disposizione  $(n - 1)$  scelte per il successivo,  $(n - 2)$  per quello dopo ancora e così via, per un totale di  $(n - 1)!$  percorsi anche se per simmetria quelli tra loro distinti risultano *solamente* la metà  $\frac{(n-1)!}{2}$ .

Solo nel 1954, George Dantzig, Ray Fulkerson e Selmer Johnson proposero un metodo più raffinato per risolvere il TSP anche se allora limitato ad un campione di  $n = 49$  città: queste rappresentavano le capitali degli Stati Uniti e il costo del percorso era calcolato in base alle distanze stradali.

Nel 1962, *Procter and Gamble* bandì un concorso per 33 città mentre nel 1977 si mirava a collegare nel modo più efficiente possibile le 120 principali città della Germania Federale. La vittoria andò a Martin Grötschel oggi Presidente del Konrad-Zuse-Zentrum für Informationstechnik 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 ed in contemporanea Groetschel e Holland trovarono il TSP ottimale riguardante un giro del mondo passante per 666 mete di interesse.

Nel 2001, Applegate, Bixby, Chvátal, and Cook trovarono la soluzione esatta ad 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, Chvátal, 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 esserne uno migliore. L'esecuzione richiese approssimativamente 15,7 anni CPU.

Questa carrellata storica mostra come il semplice passaggio da un decennio al successivo portava il superamento di ostacoli via via sempre maggiori a testimonianza dell'enorme studio svolto a livello globale, agevolato dallo sviluppo di hardware sempre migliore, portando infine alla comparsa dei primi **software solver**: tra i più noti troviamo **IBM ILOG CPLEX Optimization Studio** o più semplicemente **CPLEX** ed il già citato **Concorde**, entrambi utilizzati all'interno di questa tesi.

**CPLEX** è stato sviluppato da Robert E. Bixby /footnoteDocente presso la Rice University, Texas e commercializzato a partire dal 1988 da **CPLEX Optimization Inc.**. Attualmente la licenza è proprietà di **IBM** la quale concede, per fini accademici e riveste un ruolo centrale in gran parte degli argomenti trattati in questa tesi.

D'altro canto Concorde è stato utilizzato solo marginalmente e vi è dedicata una sezione apposita **Concorde**, per il momento basti pensare che ne è stata realizzata anche una versione per smartphone /footnoteAl giorno d'oggi solo per ambiente iOS, scaricabile gratuitamente dell' App Store capace anch'essa di risolvere i più comuni problemi *TSP* in frazioni di secondo.

Tali risultati sembrerebbero andare teoricamente contro alla natura stessa dei *TSP* in quanto è dimostrabile la loro appartenenza alla categoria di problemi **NP-hard**: ciò significa che, al momento, non è noto in letteratura un algoritmo che risolva una sua qualunque istanza in tempo **polinomiale**. Con la nascita di sempre nuove applicazioni aventi alla loro base problemi *NP-hard*, è sorta la necessità di poter usufruire di algoritmi che valorizzassero maggiormente la velocità di risoluzione anche a discapito del ritrovamento della soluzione ottima, accontentandosi di assicurarne una solamente il più *vicino* ad essa possibile. Tali algoritmi sono detti **euristici** e naturalmente ne sono presenti diversi anche per il *TSP*: è proprio grazie ad un loro utilizzo combinato a tecniche esatte<sup>3</sup> che i *solver* moderni possono raggiungere risultati apparentemente impensabili. Nel corso di questa tesi viene dato largo spazio ad entrambe le categorie algoritmiche *esatte* ed *euristiche*.

## AMBIENTE DI SVILUPPO: CPLEX, Visual Studio e C#

Il progetto è stato sviluppato in ambiente Windows, in particolare il sistema operativo scelto è Windows 10.

**CPLEX** è una componente centrale per progetto e quindi, prima di procedere, mostriamo i passi principali da eseguire per assicurarsi di potervi interagire dall'IDE<sup>4</sup> scelto. Tra i diversi linguaggi di programmazione compatibili con *CPLEX* si è scelto di utilizzare il **C#**, successore del **C++** ed anch'esso orientato agli oggetti.

L'*IDE* più comune per chi desidera utilizzare tale linguaggio è senza dubbio **Visual Studio**, sviluppato dalla stessa **Microsoft** e distribuito gratuitamente. La versione utilizzata in questo progetto, e dunque quella a cui fa riferimento questa guida, è **Visual Studio Community 2017**. Una volta aperto l'installer reperibile al seguente [indirizzo](#) è sufficiente selezionare i pacchetti **Sviluppo per desktop .NET** e **Sviluppo di applicazioni desktop con C++** (vedremo in seguito perché sono necessari pacchetti C++).

Terminato questo processo, assicuriamoci che la versione di *CPLEX* presente nella macchina sia **almeno** la **12.7.0**<sup>5</sup> utilizzata anche per questo progetto.

Ogni progetto *Visual Studio* che desidera utilizzare librerie esterne deve necessariamente settare le sue proprietà indicando le directory in cui è possibile reperirle. Creiamo un progetto **C#** selezionando dal menu a tendina **Visual C#** e quindi **App console (.NET Framework)**, forniamo il nome e percorso che desideriamo come mostrato in figura (1).

La connessione alle librerie di *CPLEX* viene stabilita seguendo i passaggi seguenti:

- Selezionare la voce **Progetto** dai menù e quindi **Aggiungi riferimento...**;
- Premere il pulsante **Sfoglia** e dopo essersi recati nella propria cartella di installazione di *CPLEX*, in genere "C:\Program Files\IBM\ILOG\CPLEX\_StudioXXXX\cplex" accedere alla sotto directory "\bin\x64\_win64" e selezionare i file **ILOG.CPLEX.dll** e **ILOG.Concert.dll**;
- Selezionare la voce **Compilazione** dai menù e quindi **Gestione configurazione...**, nella nuova finestra inserire all'interno di **Piattaforma** un nuovo campo e selezionare **x64**<sup>6</sup>. Per quanto riguarda il campo **Configurazione** è indifferente selezionare **Debug** oppure **Release**: come è facilmente intuibile nella prima modalità offre tool di *debug* aggiuntivi a discapito di

<sup>3</sup>Oltre ad hardware molto performante rispetto al passato.

<sup>4</sup>**Integrated Development Environment**: in altre parole è l'acronimo per *ambiente di sviluppo (integrato)*.

<sup>5</sup>Versioni precedenti non sono compatibili con *Visual Studio Community 2017*.

<sup>6</sup>Il progetto utilizza 64 bit.

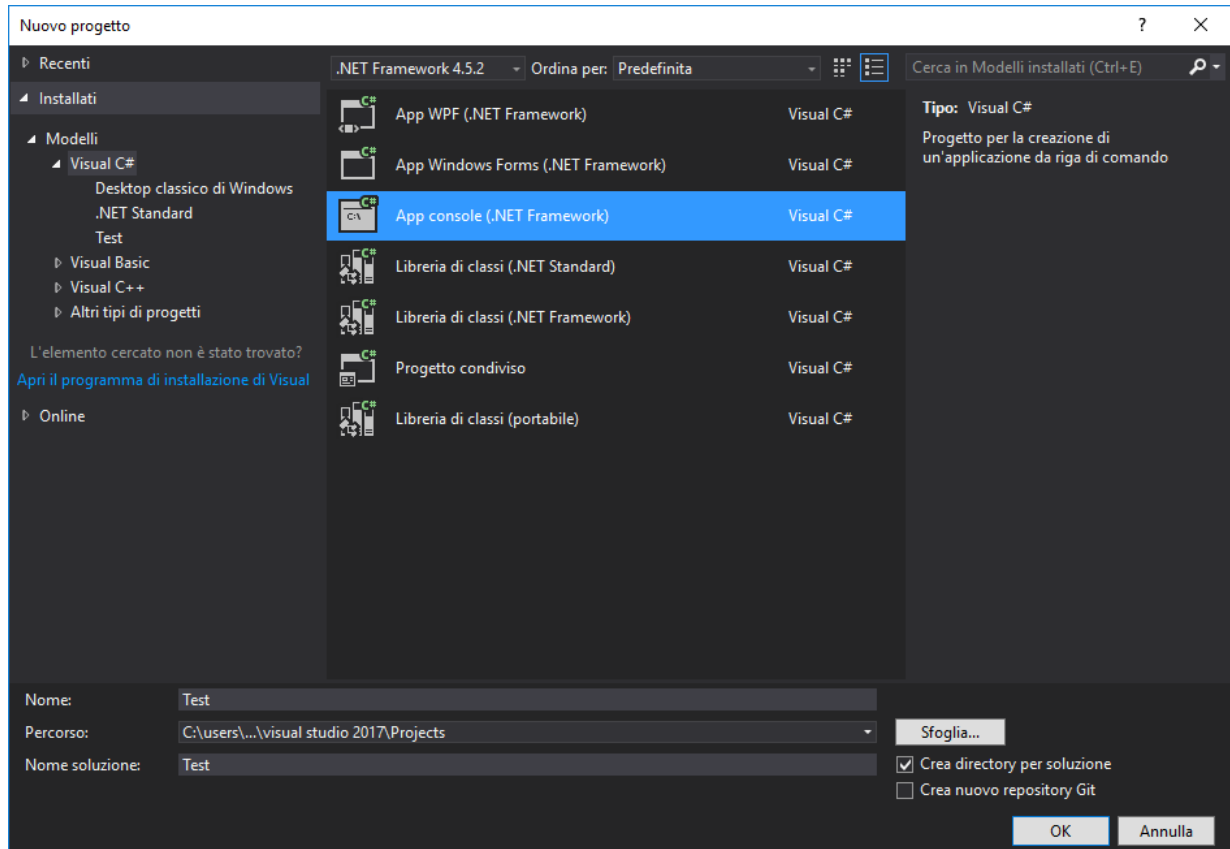


Fig. 1: Creazione progetto C#

una minima perdita prestazionale. Nel nostro caso la modalità **Release** è quella utilizzata per ottenere i risultati mostrati nella sezione **Test e Risultati**;

La completa interazione con *CPLEX* deve essere eseguita attraverso gli oggetti e le funzionalità offerta dai due pacchetti **ILOG.CPLEX** che **ILOG.Concert**, ora accessibili attraverso la direttiva **using**.

## Creazione ed utilizzo DLL C/C++

In questa sezione viene spiegato come sia possibile utilizzare codice esterno, compilato anche in linguaggi differenti dal **C#**, sotto forma di **DLL**. Nel nostro caso si giungerà ad un punto in cui è necessario servirsi del linguaggio di programmazione **C** per l'interazione con librerie di **Concorde** software solver dedicato ai *TSPs*.

La problematica principale con cui ci si scontra in questi casi è l'incompatibilità dei tipi, qui ancora più accentuata in quanto *C*, al contrario di *C#*, non è un linguaggio orientato agli oggetti e l'interfaccia con *CPLEX* segue un differente approccio.

La soluzione migliore è quella di comunicare alla *DLL* solamente le informazioni fornite in input dall'utente e cioè il nome del file contenente i dati ed il time limit, così che possa gestire autonomamente l'intera procedura. In altre parole il codice *C#* diventa in questo caso solamente<sup>7</sup> una interfaccia per richiamare la *DLL*<sup>8</sup>.

Entriamo ora nel dettaglio della procedura da seguire:

- Dall'interno di *Visual Studio* creare un nuovo progetto selezionando la voce **Visual C++** e quindi **Progetto Win32**. Nel caso in cui queste opzioni siano assenti significa che durante l'installazione dell'*IDE* non sono stati selezionati i pacchetti *textitC++*, per maggiori dettagli andare alla sezione **Ambiente di sviluppo**;
- Nelle schermate successive è necessario selezionare l'opzione **DLL** come *tipo di applicazione* e **Progetto vuoto** come *opzione aggiuntiva*;
- Creatosi il nuovo e progetto vuoto, dal menu **Esplora soluzioni** nella cartella **File di origine** premiamo il tasto destro ed aggiungiamo un nuovo elemento. Selezioniamo dal menu **File di C++ (.cpp)**, assegniamo il nome che preferiamo ed infine il tasto **Aggiungi**.
- Il file appena creato è compilato come codice **C++**, racchiudendolo però dentro

---

```
extern "C"
{
    \\Codice..
}
```

---

viene automaticamente interpretato come linguaggio **C**.

- Definiamo il metodo *entry point* per la *DLL* inserendo il prefisso **\_\_declspec(dllexport)**. Per semplificare la programmazione, è consigliabile sfruttare il punto di ingresso come una semplice interfaccia per la chiamata ad una seconda funzione che avvia effettivamente la risoluzione della istanza *TSP*:

---

<sup>7</sup>In realtà mantiene anche un cronometro per il tempo di risoluzione.

<sup>8</sup>L'output standard della *DLL* richiamata viene automaticamente settato a quello del progetto chiamante senza la necessità di eseguire alcun settaggio.

---

```
__declspec(dllexport) int Concorde(char *fileName, int timeLimit)
{
    return ExeMain(strtok(fileName, "\\0"), timeLimit);
}
```

---

- In modo analogo alla creazione del file corrente, selezionando invece l'opzione per aggiungere file esistenti, importiamo tutte le componenti con estensione **.C** necessarie da Concorde. Una lista dettagliata è fornita nella sezione **Concorde**.
- Settiamo ora le proprietà del progetto in modo tale che sia possibile utilizzare CPLEX, diverse guide sono già disponibili e la procedura non viene qui riportata. In aggiunta è necessario selezionare nel sottomenù **C/C++ → Generale → Directory di inclusione aggiuntive** la cartella dove sono presenti i file **.h** di Concorde.
- Completata la stesura del codice, la effettiva creazione della *DLL* è avviabile dal menu **Compilazione** selezionando la voce **Compila soluzione**<sup>9</sup>. A compilazione terminata, la *DLL* si trova all'interno della sottocartella **/x64/Release**.
- Copiare la *DLL* ottenuta nella directory del **progetto C#**, in particolare su **/bin/x64/Release** e **/bin/x64/Debug**, per utilizzarla nelle due modalità.
- L'importazione della *DLL* all'interno di una classe *C#* del progetto avviene attraverso:

---

```
[DllImport("NOMEDLL.dll")]
public static extern int ENTRYPOINTDLL(StringBuilder fileName,
    int timeLimit);
```

---

Il metodo *entry point* è quindi richiamabile come qualsiasi altra funzione.

Prima di concludere la sezione, si specifica che la classe **StringBuilder** è una interfaccia naturale offerta da *C#* per la conversione automatica del proprio tipo **String** a **char\*** del linguaggio *C*.

---

<sup>9</sup> Assicurarsi che nella barra degli strumenti sia selezionata la modalità **release** a **64 bit**, in realtà è solamente sufficiente che i bit coincidano con quelli adottati nel progetto originale *C#*.

## SEZIONE DUE

Questa breve sezione viene utilizzata per introdurre formalmente i *TSP* ed in particolare il modello matematico generale che li descrive, evidenziando per quale motivo tali problemi appartengono alla categoria **NP-Hard**.

### 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  $V = \{v_1, \dots, v_n\}$  è un insieme di  $n$  nodi e  $A = \{(i, j) : i, j \in V\}$  è un insieme di  $m$  archi<sup>10</sup>.

Senza perdita di generalità, si suppone che il grafo  $G$  sia completo e che il costo associato all'arco  $[i, j]$ , indicato con  $c_{ij}$ , sia non negativo. Si osservi che aver imposto  $c_{ij} \geq 0$  non è limitativo poiché è sempre possibile sommare a tutti i costi una valore costante, sufficientemente elevato, in modo tale da renderli positivi senza alterarne l'ordinamento delle soluzioni.

A differenza di quanto detto in precedenza, per tutto il proseguimento della tesi supporremo il grafo  $G$  non orientato. Tale scelta deriva dal fatto che  $c_{ij}$ , nel nostro contesto, rappresenta sempre la distanza, tipicamente euclidea, fra i vertici  $i$  e  $j$  pertanto:

$$c_{ij} = c_{ji} \quad (1)$$

Quando il grafo è non orientato, la famiglia di coppie non ordinate di elementi appartenenti a  $V$  viene indicata per convenzione con la lettera  $E$ .

Definita la nomenclatura del problema, si considera come variabili decisionali proprio i lati del grafo nel seguente modo:

$$x_e = \begin{cases} 1 & \text{se il lato } e \in E \text{ viene scelto nella soluzione trovata} \\ 0 & \text{altrimenti} \end{cases} \quad (2)$$

Di conseguenza la formulazione del modello matematico diviene:

$$\min \underbrace{\sum_{e \in E} c_e x_e}_{\text{costo circuito}} \quad (3)$$

$$\underbrace{\sum_{e \in \delta(V)} x_e}_{\text{due lati incidenti in } v} = 2, \quad \forall v \in V \quad (4)$$

$$0 \leq x_e \leq 1 \text{ intera}, \quad \forall e \in E \quad (5)$$

L'insieme definito da (4) vengono chiamati **vincoli di grado** e impongono che in ogni vertice incidano esattamente due lati. Questa formulazione è si **compatta** in quanto il numero di vincoli è

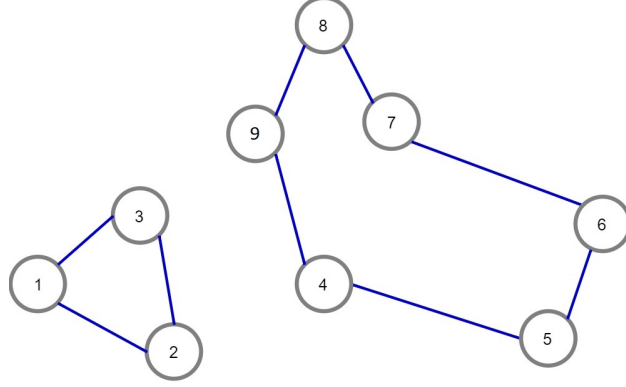


Fig. 2: Soluzione con due subtour

**polinomiale** rispetto ad  $n$  ma contemporaneamente **non completa**: nessuna equazione impedisce la formazione di *subtour* e quindi di ottenere una soluzione dove il grafo non risulti connesso.

Una possibile formulazione di vincoli atti ad impedire la formazione di subtour, detta appunto **subtour elimination**, risulta essere:

$$\sum_{e \in \delta(S)} x_e \geq 1, \forall S \subsetneq V : 1 \in S, |S| \geq 2 \quad (6)$$

Il vincolo (6) indica che se si considera un **qualunque** sottoinsieme  $S \subsetneq V$ , che includa il vertice numerato con il simbolo 1, allora il taglio di  $G$  indotto da  $S$ :

$$\delta(S) = \{[i, j] \in E : i \in S, j \notin S\} \quad (7)$$

deve contenere almeno un lato appartenente ad  $E$ : poichè tutti i subtour violano tale vincolo la soluzione ottima non potrà contenerne al suo interno.

Analizzando il numero dei possibili sottoinsiemi  $S$  distinti si nota che tale quantità risulta **esponenziale** rispetto ad  $n$ , pertanto è proprio la (6) a rendere i *TSPs* **NP-hard**.

In particolare la dimensione di  $S$ , dato un numero  $n$  di nodi, è  $2^n$ : immaginando di associare un bit ad ogni vertice (il cui valore definisce se appartiene o meno al sottoinsieme), e quindi di rappresentare un qualsiasi sottoinsieme attraverso una sequenza di  $n$  bit se ne possono definire  $2^n$  distinti. In realtà avendo imposto che il vertice 1 appartenga sempre ad ogni  $S$  e che  $S \subsetneq V$ , la quantità trovata va ridimensionata a  $2^{n-1} - 1$ , di grado comunque esponenziale rispetto ad  $n$ .

Per completezza la (8) mostra una formulazione alternativa della (6) caratterizzata dal medesimo numero di disequazione:

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \forall S \subsetneq V, |S| \geq 2 \quad (8)$$

La gestione **contemporanea** di un numero esponenziale di vincoli implica in genere tempi di risoluzione troppo elevati. Nella pratica però non è necessario inserirli tutti nel modello matematico,

<sup>10</sup>Chiaramente sia  $n$  che  $m$  sono interi positivi.



è sufficiente considerarne un numero molto più ridotto e cioè solamente quelli **utili** all'ottenimento della soluzione ottima. Non esiste alcun metodo che permetta di conoscere tale informazione in anticipo ed è compito degli algoritmi risolutori definire un opportuno **separatore**: una funzione che fornita in ingresso una soluzione  $x^*$  ottima per il modello corrente generi tutti i vincoli violati da essa.

L'aggiunta di tali disequazioni al modello matematico impone alla successiva risoluzione di trovare una soluzione ottima diversa dalla precedente  $x^*$ . Nel momento in cui il separatore non fornisce più alcun vincolo aggiuntivo significa che siamo in presenza dell'ottimo globale.

## SEZIONE TRE

Questa sezione introduce la struttura del progetto per quanto riguarda la sua parte puramente di programmazione così da rendere la prosecuzione del testo più comprensibile e semplice.

Sono presentate le principali classi e strutture dati realizzate utili a livello globale per tutti gli algoritmi implementati mentre altre di interesse più *locale* vengono introdotte solamente più avanti nel momento più opportuno.

### STRUTTURA DEL PROGETTO VISUAL STUDIO

Per prima cosa è importante chiarire l'organizzazione delle cartelle e files, una struttura ordinata agevola enormemente ogni fase del progetto: progettazione, stesura del codice, debugging, simulazione della release e versione finale.

All'interno della cartella radice, da chiamata TSPCsharp, si sono create le seguenti sottocartelle:

- **Src:** contiene il progetto della applicazione e quindi tutti i file sorgente;
- **Data:** include le istanze *TSP* utilizzabili dal software;
- **Concorde:** contiene la porzione dei file sorgente, in linguaggio di programmazione *C*, del software solver **Concorde** necessarie per il progetto;
- **Output:** la directory dove l'applicazione pone i file di output prodotti;

Il software sviluppato è composto dalle seguenti dieci classi:

- **Instance**
- **ItemList**
- **PathGenetic**
- **PathStandard**
- **Point**
- **Program**
- **Tabu**
- **TSP**
- **TSPLazyConsCallback**
- **Utility**

Per le classi **Point**, **Instance**, **Program**, **TSP** e **Utility** viene fornita una descrizione in questa sezione essendo di interesse globale per tutto il progetto.

Si specifica che le variabili globali di qualsiasi classe, presenti solamente in quelle che offrono strutture dati utili al programma, sono dichiarate con metodo di accesso **internal**<sup>11</sup> e possiedono i propri metodi **Getter**<sup>12</sup> e **Setter**<sup>13</sup> che ne ereditano il livello di accessibilità. Il linguaggio di programmazione C# offre la seguente sintassi semplificata:

---

```
protected var variabile { get; set; }
```

---

## CLASSE POINT

La classe **Point** offre una struttura dati in grado di memorizzare le coordinate bidimensionali di un singolo nodo  $n$ . Presenta a tal fine le variabili globali **x** e **y** di tipo **double**. Il costruttore della classe non fa altro che ricevere in input i valori da assegnare a queste ultime. La classe presenta inoltre un ulteriore metodo pubblico e statico chiamato **Distance** che permette il calcolo della distanza tra due nodi:

---

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

---

Dove:

- **p1**: riferimento ad un oggetto della classe **Point** che rappresenta il primo nodo;
- **p2**: riferimento ad un oggetto della classe **Point** che rappresenta il primo nodo;
- **pointType**: stringa che descrive quale tipo di formula matematica si debba utilizzare per calcolare il costo del lato delimitato da **p1** e **p2**. Può assumere i seguenti valori:
  - EUC\_2D
  - ATT
  - MAN\_2D
  - GEO
  - MAX\_2D
  - CEIL\_2D

Di seguito è riportato il codice del metodo **Distance** dove le formule matematiche utilizzate seguono le direttive indicate dai creatori delle istanze *TSP* compatibili con l'applicazione<sup>14</sup>.

---

```
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 (pointType == "MAN_2D")
```

---

<sup>11</sup>Membri interni sono accessibili solo all'interno di file nello stesso assembly.

<sup>12</sup>Utilizzato per leggere la variabile.

<sup>13</sup>Utilizzato per modificare la variabili.

<sup>14</sup>Il documento è reperibile al seguente [indirizzo](#).

```

{
    xD = Math.Abs(xD);
    yD = Math.Abs(yD);
    return (int)(xD + yD + 0.5);
}

else if (pointType == "MAX_2D")
{
    xD = Math.Abs(xD);
    yD = Math.Abs(yD);

    int fI = Convert.ToInt32(xD + 0.5);
    int sI = Convert.ToInt32(yD + 0.5);
    if (fI >= sI)
        return fI;
    else
        return sI;
}
else if (pointType == "GEO")
{
    double PI = Math.PI;

    int deg = (int)(p1.x + 0.5);
    double min = p1.x - deg;
    double latitude1 = PI * (deg + 5 * min / 3) / 180;

    deg = (int)(p1.y + 0.5);
    min = p1.y - deg;
    double longitude1 = PI * (deg + 5 * min / 3.0) / 180;

    deg = (int)(p2.x + 0.5);
    min = p2.x - deg;
    double latitude2 = PI * (deg + 5 * min / 3.0) / 180;

    deg = (int)(p2.y + 0.5);
    min = p2.y - deg;
    double longitude2 = PI * (deg + 5 * min / 3.0) / 180;

    double RRR = 6378.388;
    double q1 = Math.Cos(longitude1 - longitude2);
    double q2 = Math.Cos(latitude1 - latitude2);
    double q3 = Math.Cos(latitude1 + latitude2);

    return (int)((RRR * Math.Acos(0.5 * ((1 + q1) * q2 - (1 - q1) * q3)))
        + 1);
}
else if (pointType == "ATT")
{
    double rij = Math.Sqrt((xD * xD + yD * yD) / 10.0);
    int tij = Convert.ToInt32(rij);

    if (tij < rij)

```

```

        return tij + 1;
    else
        return tij;
}
else if (pointType == "CEIL_2D")
{
    return Math.Ceiling(Math.Sqrt(xD * xD + yD * yD) + 0.5);
}

//Nel caso in cui
throw new Exception("Bad input format");

```

---

## CLASSE INSTANCE

La classe **Instance** memorizza tutti i dati caratterizzanti l'istanza del Problema del Commesso Viaggiatore corrente. Essendo le variabili globali di questa classe numerose viene riportato il loro codice dichiarativo commentato:

---

```

//Dati input utente

//Nome file
internal string inputFile { get; set; }
//Tempo limite di esecuzione
internal double timeLimit { get; set; }

//Dati ricavati dal file di input

//Numero di nodi
internal int nNodes { get; set; }
//Coordinate (x,y) di ogni nodo
internal Point[] coord { get; set; }

//Determina come calcolare la distanza tra due punti
internal string edgeType { get; set; }

//Parametri da definire durante la risoluzione

//Costo della migliore soluzione intera trovata
internal double xBest { get; set; }
//Valore di ogni lato nella migliore soluzione intera trovata
internal double[] bestSol { get; set; }
//Migliore lower bound trovato, utilizzato in alcuni algoritmi
internal double bestLb { get; set; }

```

---

All'interno di questa classe è presente l'unico metodo pubblico e statico, esclusi i vari *get* e *set*, **Print** utilizzato per stampare all'interno della *console* tutte le coordinate memorizzate in un oggetto di tipo **Instance** -> **coord**:

---

```

static public void Print(Instance inst)

```

---

Dove:

- **inst**: riferimento ad un oggetto della classe **Instance** contenente tutti dati che descrivono l'istanza del Problema del Commesso Viaggiatore fornita in ingresso dall'utente;

Il contenuto di **Print** è per definizione molto banale. L'unico aspetto degno di nota è che i reali indici identificativi dei nodi sono compresi tra i valori 1 ed  $n$  mentre, dato che nei linguaggi di programmazione le strutture dati *contenitori* come ad esempio i **vettori** o le **liste** partono da 0, il loro indice di memorizzazione è sempre scalato di uno. *Print* è realizzato in modo tale da mostrare il reale indice di ogni nodo:

---

```
for (int i = 0; i < inst.NNodes; i++)
    Console.WriteLine("Point #" + (i + 1) + " = (" + inst.coord[i].x + " ; "
        + inst.coord[i].y + ")");

//Esempio di stampa:
//Point #1= (x_1;y_1)
//Point #2= (x_2;y_2)
//..
//Point #n= (x_n;y_n)
```

---

## CLASSE PROGRAM

**Program** è la classe *entry point* del progetto creata automaticamente dall'*IDE*. Contiene pertanto il metodo di *ingresso* **Main** che per convenzione, all'interno del linguaggio *C#*, è sempre il primo ad essere eseguito: riceve quindi come parametro di ingresso un vettore di stringhe, chiamato **argv**, al cui interno si trova la riga di comando fornita dall'utente<sup>15</sup>.

L'applicazione si aspetta di trovarvi memorizzate le informazioni riguardanti il **tempo limite** di esecuzione e soprattutto il **nome** dell'istanza *TSP* da risolvere<sup>16</sup>.

La prima azione eseguita quindi dalla classe *Program* è l'analisi del vettore *argv*, operazione delegata al metodo **ParseInput** descritto alla seguente sezione **Lettura Input File**.

Successivamente è necessario effettuare l'accesso in lettura al file, indicato dall'utente, contenente le informazioni dell'istanza *TSP* da risolvere. Anche questa operazione viene delegata ad un secondo metodo **Populate** analizzato nella sezione **Metodo Populate**.

Dichiarato e parzialmente popolato un oggetto di tipo **Instance** grazie ai metodi sopra citati, è invocato il metodo **Solve** contenuto nella **TSP**, la prossima ad essere descritta. Per ora basti sapere che *Solve* chiede all'utente di indicare l'algoritmo risolutore che desidera applicare e provvede alla sua esecuzione.

L'ultima operazione effettuata dal metodo *Main* consiste nel richiedere all'utente se desidera eliminare i file con estensione **.dat** e **.lp** creatisi durante l'esecuzione del programma. I dettagli riguardanti i file generati con queste estensioni sono riportati nel seguito della tesi.

---

```
foreach (string file in Directory.GetFiles("../...\\...\\...\\Output\\",
    "*.dat")).Where(item => item.EndsWith(".dat"))
```

---

<sup>15</sup> *Visual Studio* offre la possibilità di simulare l'utilizzo della riga di comando anche quando l'applicazione è avviata dall'interno dell'*IDE* stesso.

<sup>16</sup> Reperibile all'interno dell'apposita cartella *Data*.

```

{
    File.Delete(file);
}

foreach (string file in Directory.GetFiles("..\\..\\..\\..\\Output\\",
    "*.lp").Where(item => item.EndsWith(".lp")))
{
    File.Delete(file);
}

```

---

La classe *Program* definisce l'unica costante globale pubblica di tutto il progetto:

---

```
public const int VERBOSE = 5;
```

---

Le varie stampe a video eseguite tramite *console* nella applicazione, sono classificate secondo una scala numerica di importanza in modo tale che: solo nel caso in cui il corrispettivo valore risulti infiore o pari al **VERBOSE** attuale l'istruzione sia realmente eseguita.

---

```

if (VERBOSE >= 5)
    Console.WriteLine("Esempio VERBOSE");

```

---

## CLASSE TSP

La classe **TSP** è pensata come il cuore del programma in quanto la struttura principale di tutti i metodi risolutivi viene riportata al suo interno. Vi è presente un unico metodo pubblico **Solve**:

---

```
static public void Solve(Instance instance)
```

---

Dove:

- **instance**: riferimento ad un oggetto della classe **Instance** contenente tutti dati che descrivono l'istanza del Problema del Commesso Viaggiatore fornita in ingresso dall'utente;

Le operazioni eseguite da questo metodo sono raggruppabili in:

- Inizializzazione degli oggetti comunemente utilizzati dalla maggior parte degli algoritmi risolutori;
- Viene richiesto all'utente di indicare quale metodo di risoluzione desidera che venga applicato tra quelli disponibili. Se necessario richiede anche parametri aggiuntivi caratterizzanti la scelta effettuata;
- Viene invocato il metodo corretto per gestire la richiesta espressa dall'utente;
- Nel caso in cui l'algoritmo abbia prodotto una soluzione valida, il suo costo ed il tempo di esecuzione trascorso sono stampati nella *console*;

Per quanto riguarda gli oggetti da inizializzare troviamo: un oggetto della classe **Stopwatch** utilizzato come cronometro per gestire il tempo di esecuzione trascorso (viene anche avviato), un oggetto della classe **Process** necessario per disegnare una qualsiasi soluzione attraverso il programma **GNUPlot** ed infine un oggetto di tipo **Cplex** utilizzato per interfacciarsi con l'omonimo software

solver. Maggiori dettagli riguardanti la classe *Stopwatch* sono forniti nella sezione [Classe Stopwatch](#), per quanto riguarda l'inizializzazione dell'oggetto di tipo *Process* ed il software *GNUPlot* maggiori dettagli sono disponibili rispettivamente in [Metodo InitProcess](#) e [GNUPlot](#).

---

```
//Inizializzazione oggetto di tipo Cplex
Cplex cplex = new Cplex();
//Inizializzazione oggetto di tipo Stopwatch
Stopwatch clock = new Stopwatch();
//Avvio del cronometro
clock.Start();
//Il metodo InitProcess della classe Utility fornisce un oggetto process
    inizializzato pronto per essere utilizzato
//per la comunicazione con GNUPlot per disegnare il grafico di una
    qualsiasi soluzione trovata
Process process = Utility.InitProcess(instance);
```

---

La comunicazione con l'utente per conoscere quale metodo di risoluzione desidera utilizzare è molto semplice ed avviene attraverso la scrittura e lettura di stringhe nella *console C#*, pertanto non viene qui riportata.

L'avvio dello specifico algoritmo risolutivo viene riportato nella sezione ed esso dedicata in quanto al momento risulterebbe prematuro discuterne.

Infine la stampa riguardante l'eventuale costo della migliore soluzione trovata ed il relativo tempo di esecuzione richiesto dall'algoritmo selezionato, più lo stop del cronometro, avviene nel seguente modo:

---

```
//Program.VERBOSE >= 0 implica che la stampa venga sempre eseguita
//Se l'algoritmo avviato non ha modificato il valore di instance.xBest
    significa che nessuna soluzione è stata trovata
if(Program.VERBOSE >= 0 && instance.xBest != 0)
    Console.WriteLine("The best solution found in " +
        clock.ElapsedMilliseconds / 1000 + " seconds has cost: " +
        instance.xBest);
else
    Console.WriteLine("No solution found in " + clock.ElapsedMilliseconds
        / 1000 + " seconds");

//Il cronometro viene fermato
clock.Stop();
```

---

## CLASSE UTILITY

La classe *Utility* può essere considerata come una libreria: contiene al suo interno solamente metodi **statici**, utilizzati dagli algoritmi risolutivi implementati, che si è deciso di raggruppare al suo interno così da rendere il codice il più compatto e leggibile possibile. Questi saranno presenti durante il corso della tesi oppure inseriti nella sezione [Appendice](#).



## CLASSE PATHGENETIC

La classe **PathGenetic** utilizzata per memorizzare i dati di una soluzione generica, estende **PathStandard** già discussa nel paragrafo X.Y. aggiungendo due campi utili solamente per gli algoritmi genetici: il primo di tipo *double* memorizza la fitness associata alla soluzione, il secondo di tipo intero identifica il circuito all'atto dell'estrazione dei percorsi che formeranno la generazione successiva<sup>17</sup>. La classe è dotata del metodo privato **CalculateFitness** il quale semplicemente setta la variabile fitness come descritto in precedenza:

---

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

---

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

---

```
public PathGenetic(int[] path, double cost) : base()
{
    this.path = path;
    this.cost = cost;
    CalculateFitness();
    nRoulette = -1;
}

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

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

ANDREBBE UN COMMENTINO SU OGNUNO

---

## SEZIONE Quattro

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

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

---

<sup>17</sup>I suddetti parametri prendono nome **fitness** e **nRoulette**

Ogni istanza è memorizzata in un file di testo in un formato ben preciso, è stato quindi possibile progettare un opportuno parser che automaticamente riesca a estrapolare le informazioni contenute e popolare le strutture dati da noi create<sup>18</sup>.

## LETTURA INPUT FILE

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

---

```
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
```

---

<sup>18</sup>Nessun altro tipo di input è supportato

```

        inst.TimeLimit = Convert.ToDouble(input[++i]);
        continue;
    }
}

```

---

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

---

```

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

```

---

## METODO POPULATE

Il metodo Populate è utilizzato per la lettura dei dati contenuti all'interno del file di input e soprattutto alla loro memorizzazione all'interno di un oggetto di tipo **Instance** in modo tale che una volta conclusosi il metodo questo contenga tutte le informazioni necessarie per la creazione del modello matematico.

I file di input presenta una struttura pressoché identica tra loro e cioè una divisione in sezioni identificate da parole chiave. Fatta eccezione per la sezione che descrive le coordinate dei nodi, tutte le altre si sviluppano in una sola riga la cui struttura è del tipo:

<parolaChiave> : <valore>

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

- **NAME:**<string>
  - nome con cui l'istanza è nota in letteratura.
- **TYPE:**<string>
  - indica il tipo dell'istanza. Nel nostro ambito sarà sempre TSP.
- **COMMENT:**<string>
  - include informazioni aggiuntive, solitamente contiene il nome dei gli autori che hanno proposto l'istanza.
- **DIMENSION:**<integer>
  - indica il numero di nodi.
- **EDGE WEIGHT TYPE:**<string>
  - Definisce il modo con cui il costo del lato deve essere calcolato, i possibili valori che può assumere il contenuto di questa sezione sono stati già presentati a pagina 7 durante la descrizione del metodo Distance.
- **NODE COORD SECTION:**

- Il contenuto di questa sezione si sviluppa in più righe; in ogni riga troviamo nell'ordine:
  - \* Un numero progressivo intero positivo che comincia da 1 e che identifica il nodo. Osserviamo che anche se in input il primo nodo è numerato a partire da 1, nel vettore `Point` di `inst` le coordinate saranno memorizzate a partire dall'indice 0<sup>19</sup>.
  - \* 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** finché `line` risulta diversa da `null` oppure viene incontrata la parola chiave **EOF**.

---

```
while ((line = sr.ReadLine()) != null)  
{  
    ...  
  
    //This line signals the end of the file  
    if (line.StartsWith("EOF"))  
    {  
        Instance.Print(inst);  
        Console.WriteLine(line);  
        //Correct end of the file  
        break;  
    }  
  
    ...  
}
```

---

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 note 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

---

<sup>19</sup>Tale scelta è per mantenere una conformità con la metrica adottata dal linguaggio C# per l'enumerazione degli elementi dei vettori, nel caso in cui le coordinate vengano visualizzate a video il loro indice viene comunque incrementato di uno

positivo, per prelevare il contenuto della sezione è necessario applicare i metodi `IndexOf` e `Remove` sempre alla variabile `line`; l'intestazione di tali metodi è riportata di seguito:

---

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

---

dove:

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

---

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

---

dove:

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

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

---

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

---

Il codice riportato deve chiaramente effettuare un cast per i tipi diversi da string, i metodi necessari sono già disponibili all'interno della classe **Convert** di **C#**.

Una volta che ci troviamo all'interno della sezione **NODE COORD SECTION** la lettura delle coordinate viene eseguita eseguendo ciclicamente il seguente codice:

---

```
string[] elements = line.Split(new[] { ' ' },  
    StringSplitOptions.RemoveEmptyEntries);  
  
int i = Convert.ToInt32(elements[0]);  
  
inst.Coord[i - 1] = new Point(Convert.ToDouble(elements[1].Replace(".",  
    ",")), Convert.ToDouble(elements[2].Replace(".", ",")));
```

---

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

---

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

---

dove:

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

Ogni coordinata letta viene tradotta in un oggetto di tipo **Point** il quale è a sua volta memorizzato all'interno del vettore **Coord** dell'oggetto di tipo **Instance** nella posizione indice letta.

Come nota conclusiva specifichiamo che C# utilizza come separatore tra parte intera 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 da sostituire;
- **newValue**: stringa con cui sostituire tutte le occorrenze di **oldValue**.

## SEZIONE CINQUE

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 LINGUAGGIO 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 environment è possibile associare uno o più modelli attraverso il comando **CPXcreateprob**, la cui intestazione è:

---

```
CPXLPptr CPXcreateprob(CPXENVptr 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 obbiettivo; è possibile procedere in più modi, quello da noi scelto è di utilizzare la funzione **CPXnewcols** la cui firma è:

---

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

---

dove:

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

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

E' giunto quindi il momento di parlare di quali variabili vogliamo aggiungere al nostro modello tenendo presente che il medesimo discorso sarà applicato anche per la parte in C#. Sappiamo che per ogni coppia di nodi  $(i,j)$ <sup>20</sup> esiste un unico lato che li collega e che quest'ultimo è privo di direzione. Si presenta quindi la necessità di definire una convenzione per l'assegnazione del nome ai vari lati. La scelta adottata è la seguente: considerando due generici nodi  $i$  e  $j$  allora il loro lato sarà chiamato  $x(i,j)$  se  $i < j$  oppure  $x(j,i)$  se  $j < i$ <sup>21</sup>.

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  $(v,w)$ <sup>22</sup> la posizione di memoria in cui viene memorizzata l'informazione riguardante la prima coppia è **inferiore** rispetto alla seconda se e solo se  $(i < v) \vee (i == v \wedge j < w)$ . In altre parole saranno memorizzate in ordine le informazioni per i nodi  $(1,2), (1,3), \dots, (2,3), (2,4), \dots, (n-1,n)$ .

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

Le operazioni descritte sono state realizzate tramite il seguente codice:

---

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

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

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

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

---

<sup>20</sup>Ricordiamo che  $i$  deve essere diverso da  $j$  in quanto per i problemi da noi considerati i cappi non sono ammessi

<sup>21</sup>Notiamo che per quanto espresso nella nota precedente non ha senso considerare il caso  $i = j$

<sup>22</sup>dove assumiamo  $i < j \wedge v < w$



}

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

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

---

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

---

dove:

- **env**: puntatore all'environment 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 obiettivo, le variabili e i vincoli del modello.

In `C#` le variabili del modello sono oggetti il cui tipo deve implementare l'interfaccia **INumVar**. Non è necessario creare da noi una nuova classe infatti ci viene fornito il metodo **NumVar** della classe **CPLEX**:

---

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

---

dove:

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

Che come si può notare nella firma ha come tipo di ritorno un tipo di oggetto che implementa l'interfaccia da noi desiderata. Vedremo nel seguito della trattazione quanto utili risultano essere le funzionalità offerte da quest'ultima.

Introduciamo ora una seconda interfaccia **ILinearNumExpr** che come si può intuire viene implementata da oggetti che vogliono definire una espressione lineare. Anche in questo caso ci viene incontro la classe **Cplex** attraverso il metodo **LinearNumExpr**:

---

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

---

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

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

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

---

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

---

dove:

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

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

---

```
//Populating objective function

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

```

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

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

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

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

```

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

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

---

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

---

Per quanto riguarda i vincoli è necessario utilizzare i metodi **AddEq**, **AddLe**, **AddGe** che rispettivamente aggiungono al modello una equazione, una disequazione avente segno  $\leq$ , una disequazione avente segno  $\geq$ .

Nel nostro caso poichè ogni vincolo è una equazione riportiamo di seguito la firma della relativa funzione:

---

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

---

dove:

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

Il codice completo diventa quindi:

---

```

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

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

```

```

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

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

```

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

```

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

```

dove:

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

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

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

```

cplex.Solve();

```

Una volta avviata la risoluzione, CPLEX fornisce in automatico informazioni sul processo stampate nello standard output da noi definito<sup>23</sup>: inizialmente troviamo le impostazioni di risoluzione selezionate come ad esempio il numero di Thread .., successivamente .. .

Terminata l'operazione il costo della soluzione è memorizzato all'interno della variabile **ObjValue** di tipo **double** del solito oggetto **cplex**:

```

cplex.ObjValue;

```

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

```

public virtual double GetValues(INumVar[] var)

```

dove:

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

<sup>23</sup>Se non viene modificato di default risulta essere la classica *console* del progetto C#

È presente anche l'analogo metodo per accedere al valore di una sola variabile **GetValue**. Il suo utilizzo è da noi altamente sconsigliato in quanto sperimentalmente è stato 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

---

## SEZIONE SEI

### METODO LOOP

Il primo metodo sperimentato prende il nome di **LOOP**.

---

— Va messa un pò di storia!!!! — L'idea alla sua base è molto semplice ed è la seguente: inizialmente il modello fornito non deve contenere alcun vincolo di subtour elimination ed una volta risolto si procede 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 sé che quest'ultimo passo va ripetuto fino a quando la soluzione proposta non risulta accettabile e quindi priva di subtour<sup>24</sup>. È importante far notare che ogni iterazione del loop i vincoli aggiunti nella precedente sono ovviamente mantenuti.

In questo modo siamo sicuri di aver aggiunto al nostro modello solo i vincoli strettamente necessari il che non assicura che essi non siano un numero esponenziale.

Per poter implementare il metodo Loop risulta quindi evidente la necessità di sviluppare un'opportuna funzione in grado di individuare la presenza di subtour all'interno di una generica soluzione proposta e di generarne gli opportuni vincoli per eliminarli.

In letteratura esistono molteplici modi per eseguire tali operazioni, quella da noi adottata si rifà all'algoritmo di Kruskal per trovare un albero a costo minimo in un grafo connesso con lati non orientati<sup>25</sup>.

La tecnica da noi adottata è stata quella di creare due metodi chiamati **InitCC** e **UpdateCC**: il primo serve solamente come inizializzazione per le strutture dati utilizzate dal secondo il quale, se invocato una volta per ogni lato appartenente alla soluzione attuale ne trova tutte le componenti connesse indicando anche quali lati sono a loro appartenenti. I dettagli riguardo le loro implementazioni sono visibili nella appendice di questo testo, per ora specifichiamo solamente che al termine dell'utilizzo del metodo **UpdateCC** i seguenti oggetti:

---

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

---

risultano essere costruiti, in particolare **rcExpr** contiene le espressioni dei subtour elimination mentre invece **bufferCoeffRC** contiene il numero di lati appartenenti ad ogni subtour e quindi il termine noto delle precedenti espressioni<sup>26</sup>.

---

<sup>24</sup>Da qui deriva il nome del metodo in quanto la soluzione consiste in un loop delle stesse operazioni

<sup>25</sup>Nello specifico la parte di nostro interesse è quella che impedisce la formazione di più componenti connesse

<sup>26</sup>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**

Se all'interno di **rcExpr** è presente una espressione sola significa che la soluzione attuale è valida e quindi ottima per il problema, al contrario si deve procedere all'inserimento dei vincoli con un semplice ciclo for:

---

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

---

## METODI UpdateCC e InitCC

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

Prima di passare alla implementazione vera e propria introduciamoli ad alto livello: inizialmente si vuole assumere l'esistenza di  $n$  **cc** distinte, ognuna di esse contenente un nodo della soluzione. Questo è il compito della dalla funzione **InitCC**.

Successivamente per ogni lato della soluzione si vuole analizzare a quali **cc** sono assegnati i due nodi che lo caratterizzano. Se queste sono differenti vanno unificate in modo tale che tutti i nodi appartenenti, ad esempio, alla seconda ora appartengano tutti alla prima. Nel caso in cui invece le due **cc** coincidano significa che abbiamo trovato un subtour e il relativo vincolo di eliminazione deve essere definito. Tutte questo è invece compito del metodo **UpdateCC**.

Iniziamo quindi l'analisi del codice necessario. Per prima cosa si necessita di un vettore di interi che contenga all'indice  $i$  –esimo l'identificativo della **cc** alla quale appartiene il nodo  $i$ <sup>27</sup>. 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;

---

<sup>27</sup>Per semplicità si è deciso di identificare ogni **cc** con un valore intero univoco

- **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 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  $[i,j]$ .

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

---

```
if (relatedComponents[i] != relatedComponents[j])
{
    for (int k = 0; k < relatedComponents.Length; k++)
    {
        if ((k != j) && (relatedComponents[k] == relatedComponents[j]))
        {
            //Same as Kruskal
            relatedComponents[k] = relatedComponents[i];
        }
    }
    //Finally also the vallue relative to the Point i are updated
    relatedComponents[j] = relatedComponents[i];
}
```

---

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

---

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

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

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

---



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

## METODO LOOP CON PRIMA FASE EURISTICA

Il metodo Loop, indipendente dalla implementazione che si decide di utilizzare, vuole essere un algoritmo esatto. In altre parole è necessario assicurarsi che il risultato finale da esso prodotto sia **sempre** il migliore possibile. Come vedremo nei paragrafi successivi, sono stati pensati molteplici algoritmi, detti euristici, che al contrario cercano solamente di avvicinarsi al risultato ottimo limitando al contempo i loro tempi di esecuzione. È infatti quest'ultimo fattore a risultare cruciale per molti problemi di programmazione lineare, a maggior ragione per quelli che, come il *commesso viaggiatore*, vogliono studiare situazioni **np-difficili**<sup>28</sup>. Per quanto appena esposto si è pensato di progettare una variante del metodo **Loop** caratterizzata da una fase iniziale **euristica** i cui risultati vengono poi sfruttati da una seconda fase finale **esatta**. Durante la sua progettazione ci si accorge fin da subito che, come per ogni algoritmo euristico, non esistono specifici paletti che se fissati assicurano al **100%** il raggiungimento dei propri obiettivi. Nel nostro caso ciò che desideriamo è chiaramente una fase *euristica* più veloce di quella *esatta* ma che produca anche risultati utili a quest'ultima. Abbiamo quindi deciso che, all'interno della nostra applicazione, sia l'utente stesso a poter settare alcuni parametri che rendono le due fasi più o meno differenti tra loro. In questo modo, basandosi sulle proprie esperienze e test, è possibile ottenere i risultati migliori per qualsiasi istanza del problema che si desidera risolvere. Entriamo ora nel dettaglio delle due fasi chiarendo fin da subito che tutte e due devono sempre fornire soluzioni **valide** per il problema che andiamo a risolvere. In realtà ciò che è interessante discutere riguarda quasi solamente la fase *euristica* in quanto quella *esatta* è in tutto e per tutto il classico metodo **Loop** già esposto in precedenza<sup>29</sup>. Esistono innumerevoli modi per rendere euristico il metodo **Loop**, possiamo suddividerli in tre grandi categorie: della prima fanno parte le tecniche che rendono la risoluzione stessa da parte di **CPLEX** euristica, nella seconda ricadono i metodi che introducono nuovi vincoli al modello matematico ed infine la terza categoria è una semplice combinazione delle due precedenti. All'interno del nostro progetto sono state sviluppate tre varianti della fase *euristica*, una per ogni categoria appena esposta:

- **prima categoria:** durante la risoluzione del problema attraverso la tecnica del **Branch&Cut**, **Cplex**, oltre al banale calcolo della soluzione ottima per ogni nodo dell'albero decisionale, sfrutta internamente algoritmi euristici per agevolare il processo. Durante quest'ultimo si hanno quindi a disposizione due parametri, il primo è il costo della soluzione euristica migliore ( $C_{eu}$ ), mentre il secondo è il classico costo **lower bound**<sup>30</sup> ( $L_b$ ). È importate far notare che questi valori mutano mano a mano che si procede alla costruzione dell'albero decisione, in particolare  $C_{eu}$  cresce mentre  $L_b$  scende fino a che, teoricamente, non coincidano. La distanza **relativa** tra i due valori viene costantemente monitorata da **Cplex** e, se questa scende sotto

<sup>28</sup>In letteratura è noto infatti che i problemi appartenenti a quest'ultima categoria sono caratterizzati da tempi di risoluzione, al caso peggiore, esponenziali rispetto al numero di variabili che li caratterizzano. Nel caso in cui siano definiti nel tipico linguaggio della programmazione lineare, la caratteristica appena esposta è riscontrabile da un numero di vincoli anch'esso esponenziale.

<sup>29</sup>Ciò che varia è solamente che il modello matematico iniziale dato in pasto alla fase *esatta* presenta già dei vincoli di **subtour elimination** individuati dalla fase *euristica* dove però il modello matematico di partenza presenta già alcuni vincoli di *subtour elimination*.

<sup>30</sup>Nel nostro caso sarà il costo della migliore soluzione intera trovato.

la soglia minima del suo parametro interno **EpGap**, il processo di risoluzione viene considerato terminato e la miglior soluzione valida trovata viene restituita. Maggiori dettagli riguarda *EpGap* sono forniti nel paragrafo ad esso dedicato LINK!!!!!!!, per ora ci basta dire che se di default è settato ad un valore vicino allo 0, la sua variazione è proprio ciò che viene utilizzato nel nostro programma all'interno della fase *euristica* del metodo *Loop*. Risulta estremamente intuitivo e facilmente verificabile che per una specifica istanza non è possibile a priori determinare quanto velocemente verrà raggiunto un certo gap durante la fase di *Branch&Cut*. E' quindi consigliabile eseguire inizialmente il normale metodo **Loop**, analizzare l'output fornito da CPLEX durante la risoluzione ed osservare per quale gap soluzioni successive dell'albero decisionale non producono più miglioramenti sostanziali e o richiedono tempi di esecuzione eccessivi. Per quanto detto il valore di **EpGap** viene lasciato a discrezione dell'utente;

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

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

## EpGap

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

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

risulti inferiore ad *EpGap* il solver si arresta.

Il settaggio del parametro in questione è molto semplice, è sufficiente invocare il metodo **SetParam** sull'oggetto della classe **CPLEX** che si sta utilizzando come riportato di seguito:

---

```
cpLex.SetParam(CPLEX.DoubleParam.EpGap, newValue);
```

---

Essendo *EpGap* un valore di distanza **relativo** e non **assoluto**, la variabile **newValue** deve essere compresa tra i valori 0<sup>31</sup> e 1<sup>32</sup>.

---

<sup>31</sup>La risoluzione termina fornendo sempre la soluzione ottima.

<sup>32</sup>La risoluzione termina alla prima soluzione ammissibile individuata.

Completato il settaggio di *EpGap* è sufficiente procedere con il normale metodo risolutivo **Loop**. Al termine di quest'ultimo otteniamo una soluzione euristica ma soprattutto un insieme di vincoli di *subtour elimination*. Entriamo quindi nella fase *esatta* dell'algoritmo riportando al valore di default *EpGap* e ripetiamo il metodo risolutivo **Loop** sul modello matematico originale ampliato dai nuovi vincoli appena citati.

```
Insert 1 to use the classic loop method, 2 to use the optimal branch & cut , 3 to use heuristics methods , 4 to use matheuristics: 1
Insert 1 for normal resolution, 2 to specifie the % precion, 3 to use only a # of the nearest edges, 4 to use both previous options:
```

Fig. 3: Output

## TITOLO DA CAMBIARE

Nel paragrafo LINK!!!!!!!!!!!!!!!, è stato presentato il metodo **BuildModel** della classe **Utility** il cui compito è di creare il modello matematico, privo dei vincoli di *subtour elimination*, risolubile da CPLEX. Durante la presentazione della sua firma, era stato lasciato in sospeso l'utilizzo del parametro *nEdges* in quanto allora del tutto prematura.

Come specificato nel paragrafo LINK!!!!!!!!!!!!!!! una variante euristica del metodo **Loop**, prevede di utilizzare solamente un numero massimo  $m$  di lati incidenti in ogni nodo del problema in questione del commesso viaggiatore. Tale parametro è inoltre richiesto all'utente per i motivi già specificati e viene comunicato al metodo **BuildModel** proprio grazie alla variabile di ingresso **nEdges**<sup>33</sup>.

A livello implementativo, per rendere inutilizzabili certi collegamenti, è sufficiente non inserirli nel modello matematico oppure farlo ma fissandone sia il **lower** che l'**upper bound** a 0. Quest'ultima opzione è quella più comoda da utilizzare in quanto nella successiva fase *esatta* dell'algoritmo il modello matematico deve comunque aver disponibili al suo interno tutte le variabili.

L'individuazione algoritmica di quali collegamenti debbano essere abilitati risulta l'operazione più complessa da un punto di vista computazionale. Come vedremo nel corso di questa tesi, diversi algoritmi necessitano di conoscere per ogni nodo l'ordinamento completo dei lati ad esso incidenti basato sul loro costo e per tanto si è definita una unica funzione **BuildSL** adibita a tale scopo<sup>34</sup>.

I dettagli riguardanti al metodo **BuildSL** sono riportati al seguente paragrafo della appendice LINK!!!!!!!!!!!!!!!, in questo contesto ci basta indicare che essa restituisce una lista di vettori di interi, chiamata **listArray**, in cui  $i$  -esimo elemento contiene, in ordine crescente, la sequenza dei restanti  $n - 1$  vertici basandosi sulla loro distanza rispetto al nodo  $i$ . In altre parole, una volta invocato *BuildSL*, i lati il cui **upper bound** deve essere posto pari ad 1 sono individuabili dagli estremi  $[0, (listArray[0])[0]]$ ,  $[0, (listArray[0])[1]]$ , ...,  $[0, (listArray[0])[m]]$ , ...,  $[i, (listArray[i])[0]]$ , ...,  $[i, (listArray[i])[m]]$ ,  $1, (listArray[n - 1])[0]]$ , ...,  $[n - 1, (listArray[n - 1])[m]]$ .

<sup>33</sup>Come dettaglio implementativo specifichiamo inoltre che per convenzione si è deciso di settare  $nEdges = -1$  nel caso in cui si voglia costruire un modello con tutti i lati possibili abilitati

<sup>34</sup>In realtà il costo computazionale per trovare gli  $m$  lati meno costosi incidenti in un nodo, ripetuto per tutti gli  $n$  nodi disponibili, è il medesimo del metodo *BuildSL* e quindi l'utilizzo di quest'ultimo non porta alcuno svantaggio.

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

---

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

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

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

---

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

---

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

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

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

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

---

Terminata la costruzione del modello matematico si procede alla sua risoluzione attraverso il classico metodo **Loop**. Al suo termine, otteniamo una soluzione euristica e soprattutto una lista di vincoli di *subtour elimination*. Per passare alla risoluzione esatta del problema in analisi, manteniamo questi

ultimi e settiamo l'**upper bound** di ogni variabile ad 1. Questa ultima operazione è ottenibile invocando il metodo **ResetVariables** appartenente alla classe **Utility**:

---

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

---

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

## SEZIONE SETTE

In questa sezione esponiamo una tecnica alternativa per l'inserimento delle espressioni di **subtour elimination** all'interno di un sistema. Ciò che varia rispetto al metodo **Loop** presentato in precedenza è il **momento** in cui tali espressioni vengono definite.

L'idea è di sfruttare il fatto che CPLEX come metodo di risoluzione per i problemi di **PLI** utilizza la tecnica del **Branch&Cut**<sup>35</sup>. Viene inoltre offerta la possibilità di conoscere la soluzione trovata, sia essa frazionaria o intera, per ogni nodo dell'albero decisione ma soprattutto la possibilità di ampliare il modello matematico come meglio crediamo.

Quello che vogliamo fare risulta a questo punto molto chiaro: se alla analisi della soluzione di un nodo sono presenti **subtour** il modello matematico deve essere modificato per eliminarli.

A livello pratico CPLEX permette l'implementazione distinta di callback che vengono eseguite nel momento in cui viene trovata una soluzione intera oppure frazionaria<sup>36</sup>: nel primo caso è necessario implementare una **"lazy constraint callback"** mentre nel secondo caso una **"user cut callback"**. Da notare che in realtà solo soluzioni valide per i criteri di **fathoming** possono far scattare una callback, ciò non avviene ad esempio se il valore della soluzione di un nodo risulta maggiore rispetto a quello dell'**incumbent**<sup>37</sup>.

In generale i tagli possono essere definiti **locali** o **globali**: mentre i primi hanno validità esclusiva all'interno del sottoalbero avente come radice il nodo per la quale sono stati generati, i secondi hanno validità per tutti i nodi dell'albero decisionale e vengono memorizzati in una struttura globale detta **pool di tagli**. CPLEX inoltre fornisce la possibilità di definire un taglio **purgeable** o meno: nel primo caso significa che può essere rimosso in un secondo momento poiché ritenuto inefficace. Durante il proseguo della tesi i tagli saranno da considerarsi sempre globali e non **purgeable**.

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

- La soluzione ottima fornita da CPLEX risulta per costruzione priva di **subtour**: non è quindi più necessario, al contrario del metodo **Loop**, lanciare molteplici risoluzioni. A livello pratico è sufficiente invocare solo una volta il metodo **CPLEX.Solve()**.

---

<sup>35</sup>Da ora in avanti sarà abbreviato con la sigla **B&C**

<sup>36</sup>In informatica una callback è una funzione definita dall'utente che viene eseguita in automatico dal sistema ogniqualvolta scatta un particolare evento.

<sup>37</sup>Per **incumbent** si intende il valore migliore trovato fino a questo momento relativo ad una soluzione accettabile.

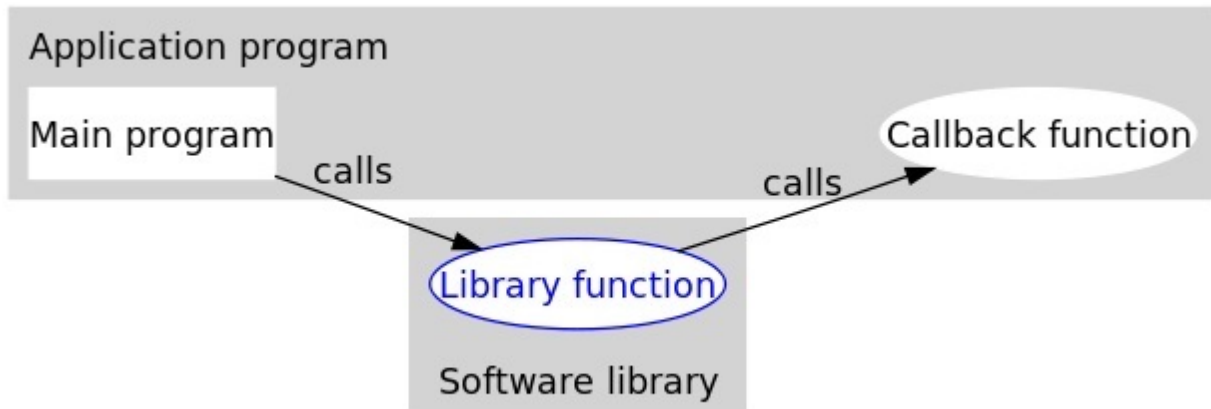


Fig. 4: Soluzione frazionaria

- Maggiore è il numero di vincoli che andiamo ad inserire, maggiore diventa il tempo di risoluzione per i vari nodi successivi dell'albero decisionale.
- Il numero di nodi che forniscono soluzioni frazionarie risulta di molto superiore rispetto a quelli con soluzione intera. Tenendo conto di quanto detto al punto precedente, non è quindi saggio andare ad analizzare tutte le soluzioni frazionarie ma solo una loro minima percentuale. Senza questo accorgimento si andrebbe inevitabilmente ad inserire innumerevoli vincoli superflui per l'ottenimento della soluzione ottima con conseguenti tempi di risoluzione eccessivamente elevati.
- I moderni processori hanno a disposizione molteplici core sia reali che virtuali e quindi sfruttare tecniche di multi-threading. In particolare CPLEX permette di settare il numero di thread utilizzabili, in modo tale che ognuno di essi si occupi dalla risoluzione di un nodo dell'albero decisionale: in questo modo, in linea teorica, si dovrebbe ottenere un boost delle prestazioni con conseguente riduzione dei tempi di calcolo. D'altro canto, come per qualsiasi applicazione informatica, l'utilizzo del multi-threading risulta rischioso in quanto l'accesso contemporaneo ai medesimi dati può portare ad una loro inconsistenza. Nel nostro caso può capitare che più callback eseguite contemporaneamente vadano a modificare variabili condivise andando così incontro ad eccezioni o anomalie tali da non garantire più la correttezza della soluzione prodotta da CPLEX.

Per evitare queste problematiche, i progettisti di CPLEX hanno preferito settare il numero di thread al valore **1** dopo l'installazione di una callback. È quindi nostro compito modificare tale parametro così da renderlo pari al numero di processori virtuali a nostra disposizione e di conseguenza assicurarci che le callback risultino **thread-safe**. Maggiori dettagli sono riportati nei successivi paragrafi.

## LAZYCONSTRAINT CALLBACK C#

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

quindi necessario creare una propria classe che estenda quest'ultima, nel nostro caso è stato deciso di chiamarla **TSPLazyConsCallback**, in questo modo è necessario definire al suo interno il metodo **Main** che verrà invocato automaticamente dal sistema ogniquale volta scatta la callback<sup>38</sup>.

Una volta terminato questo processo l'installazione della callback viene eseguita nel seguente modo:

---

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

---

Dove, come al solito, `cplex` è l'istanza della classe **CPLEX** sulla quale definiamo il modello matematico privo dei vincoli di subtour elimination.

Mostriamo ora la firma del costruttore della classe **TSPLazyConsCallback** riportando una breve descrizione dei parametri di ingresso:

---

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

---

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

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

---

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

---

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

---

<sup>38</sup>Rocordiamo che tutti i metodi astratti presenti all'interno di una classe astratta devono essere obbligatoriamente definiti da tutte le classi che estendono quest'ultima



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

---

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

...

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

---

La funzione **GetNodeId** risulta disponibile in quanto ereditata dalla classe **ControlCallback**. Dopo i dovuti chiarimenti riguardanti il multi-threading passiamo ora a descrivere nei dettagli come è stata realizzato il metodo **Main** della classe **TSPLazyConsCallback**. Prima di tutto per verificare l'eventuale presenza di subtour bisogna naturalmente accedere alla soluzione fornita per il nodo dell'albero decisionale in questione: a tal fine si possono utilizzare i metodi **GetValues** e **Get-Value**, ereditati entrambi dalla classe **ControlCallback**, che ricevono in input rispettivamente un vettore di riferimenti per variabili del modello matematico e un singolo riferimento ad una variabile di quest'ultimo. Dopo pochi test ci si accorge immediatamente che invocare più volte il metodo **GetValue**, ad esempio dentro un ciclo **for**, risulta molto più oneroso in termini temporali rispetto una singola evocazione del metodo **GetValues**: si può quindi dedurre che è molto più dispendioso effettuare molteplici accessi all'interfaccia fornita da CPLEX rispetto alla quantità di dati che ad essa richiediamo.

Per quanto appena detto la nostra scelta è ricaduta nel metodo **GetValues** che restituisce un vettore di **double** contenente il valore delle variabili (il cui riferimento è ricevuto come ingresso) nella soluzione corrente del modello matematico. Da notare che anche in questo caso anche se ci aspettiamo tutti valori interi, in particolare pari a 0 oppure 1, è possibile che ci siano in realtà discostamenti infinitesimi pertanto quando controlliamo il valore di una variabile verifichiamo semplicemente se è maggiore o minore del valore 0,5.

I metodi utilizzati per l'individuazione di eventuali subtour e la eventuale stampa del grafo attraverso GNUPlot sono identici a quelli utilizzati per il metodo **Loop** pertanto non sono qui riportati. Una volta ottenute tutte le informazioni riguardanti i subtour, al contrario di quanto viene fatto nel metodo **Loop** non è richiesto di ampliare direttamente il modello con nuovi vincoli ma, come era già stato accennato in precedenza, deve essere popolato il pool di tagli associato al modello matematico:

---

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

//if cuts.Length is 1 the graph has only one tour then cuts aren't needed
if (cuts.Length > 1)
{
    for (int i = 0; i < cuts.Length; i++)
    {
        cuts[i] = cplex.Le(ccExprLC[i], bufferCoeffCCLC[i] - 1);
    }
}
```



```

        Add(cuts[i], 1);
    }
}

```

---

Dove:

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

## CONCORDE

L'utilizzo di una **user cut callback** risulta molto più complicato rispetto a quanto appena visto per la **lazy constraint callback**: dato che tali callback scattano nel momento in cui viene trovata una soluzione frazionaria, l'individuazione di eventuali subtour non può essere eseguita con le tecniche esposte fino ad ora.

Per effettuare tale operazione si necessita di un separatore che, ricevendo in ingresso la soluzione  $x^*$  con almeno una componente frazionaria, fornisca in uscita un insieme  $S \subsetneq V$ ,  $|S| \geq 2$  tale per cui:

$$\sum_{(i,j) \in E(S)} x_{i,j}^* \not\leq |\hat{S}| - 1 \quad (9)$$

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

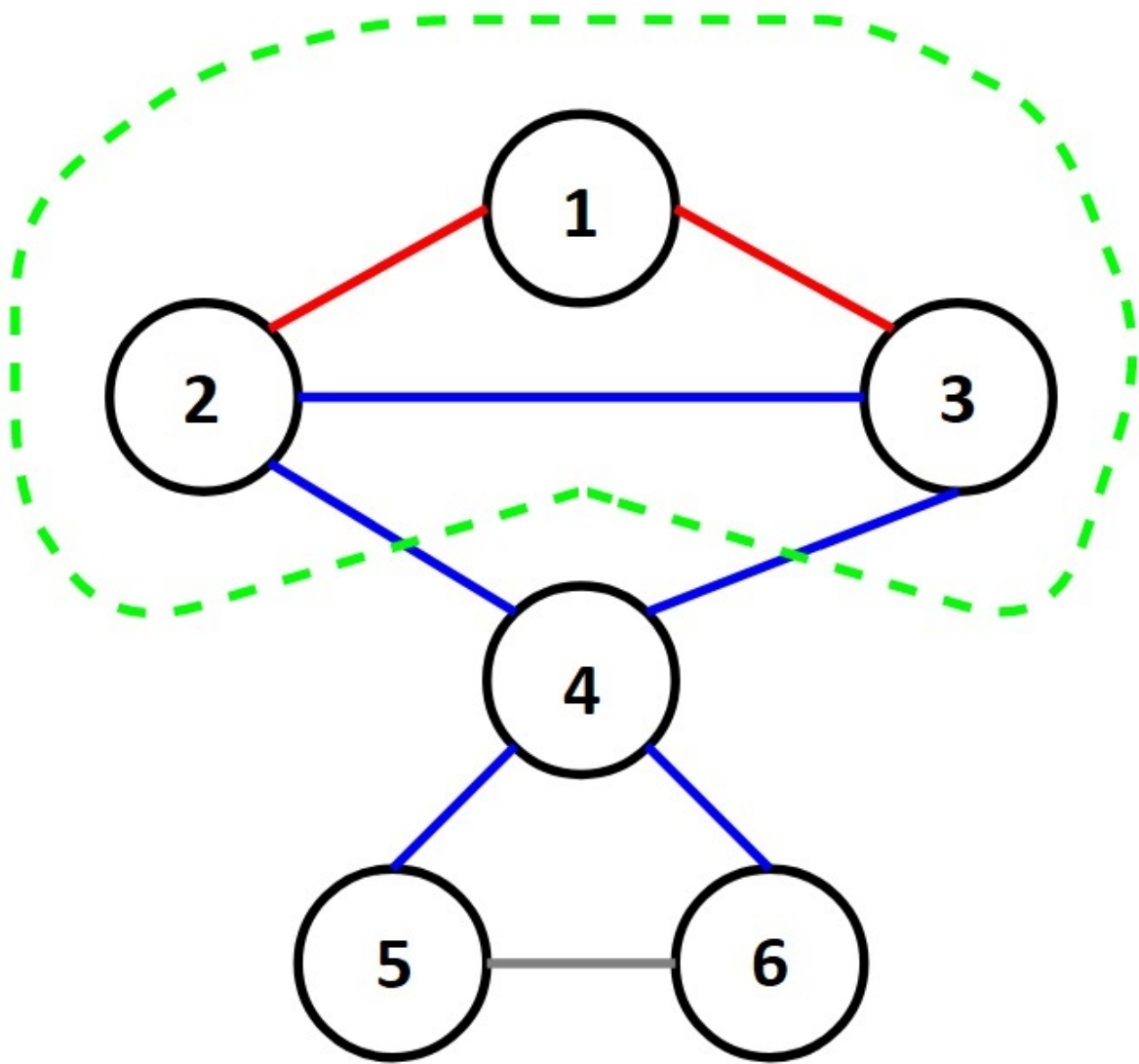


Fig. 5: Soluzione frazionaria

Tale grafo risulta connesso; tuttavia è presente un sottoinsieme  $S = 1,2,3$  per cui vale (9). Una seconda formulazione equivalente alla (9) risulta essere la seguente:

$$\sum_{(i,j) \in \delta(S)} x_{i,j}^* \not\geq 2 \quad (10)$$

Si osserva che il primo membro di (10) può essere visto come la capacità di una sezione di una rete di flusso se si interpretano le  $x^*$  come le capacità della rete. E' possibile calcolare una sezione di capacità minima risolvendo un problema di max flow che sappiamo essere di programmazione lineare e quindi risolubile attraverso un algoritmo polinomiale.

Poichè però la sezione di capacità minima dipende dal nodo sorgente  $s$  e dal nodo di destinazione  $t$ , si devono in realtà risolvere **n-1** problemi di max flow: per tale ragione è stato preferito utilizzare una porzione del software **Concorde** che offre, attraverso le proprie librerie, la possibilità di risolvere tale problema con tempi di esecuzione molto brevi ed allo stesso tempo di alleggerire il nostro carico di lavoro che in ogni caso non avrebbe prodotto risultati migliori.

Concorde è un software, sviluppano in linguaggio **C** da David Applegate, Robert E. Bixby, Vašek Chvátal, e William J. Cook, specializzato nella risoluzione ottimizzata di istanze del problema del commesso viaggiatore. Per fini accademici la distribuzione e l'utilizzo è fornita in modo gratuito e le librerie possono essere scaricate direttamente dal seguente indirizzo:

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

Come accennato poco fa il linguaggio utilizzato da Concorde non è **C#** bensì **C** pertanto una implementazione diretta del software non è possibile. La soluzione da noi adottata è la seguente: abbiamo creato un nuovo progetto Visual Studio in linguaggio **C/C++** ed al suo interno abbiamo creato un codice che soddisfacesse unicamente alla funzionalità ambedue i tipi di callback proposti interfacciandosi alle librerie di Concorde per trovare i vincoli di subtour elimination ed aggiungerli al modello matematico. Successivamente il tutto è stato impacchettato all'interno di una **DLL** compatibile con il linguaggio **C#**. Il processo di creazione della DLL è stato spiegato nel seguente paragrafo LINK!!!!!!!!!!!!!!!!!!!!, d'ora in avanti quindi ci focalizzeremo unicamente nel contenuto della libreria dinamica da noi creata.

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

- **allocrus.c**
- **connect.c**
- **cut\_st.c**
- **mincut.c**
- **shrink.c**
- **sortrus.c**
- **urandom.c**

- **cut.h**
- **machdefs.h**
- **macrorus.h**
- **util.h**
- **end**

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

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

## LAZYCONSTRAINTCALLBACK IN C

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

---

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

---

Dove:

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

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

---

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

---

Passiamo ora a descrivere la funzione *myLazyCallBack* che identifica i subtour ed aggiunge i relativi vincoli al modello, ricordando che la sua firma deve rispettare specifici parametri definiti da CPLEX stesso<sup>39</sup>:

---

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

---

Dove:

- **env**: rappresenta l'istanza dell'environment con il quale stiamo lavorando. DIVERSO DA SOPRA!!!!!!!
- **cbdata**: come accennato poco fa questo parametro è quello specificato dall'utente durante l'installazione della callback, non essendo noto a priori il tipo di dato che l'utente desidera ricevere si utilizza **void**;
- **wherefrom**: definisce da che punto del *BC* è stata invocata la funzione, ai fini pratici tale parametro, per la lazy callback è risultato irrilevante;
- **cbhandle**: puntatore a dati privati utilizzato da CPLEX;
- **useraction\\_p**: puntatore ad un intero utilizzato dall'utente per comunicare a CPLEX diverse informazioni. Tale parametro può assumere i seguenti tre valori:
  - **0**: avente come costante simbolica `CPX_CALLBACK_DEFAULT` comunica a CPLEX che fino a quel punto la callback non ha aggiunto tagli al modello;
  - **1**: avente come costante simbolica `CPX_CALLBACK_FAI` impone a CPLEX di uscire dall'ottimizzazione;
  - **2**: avente come costante simbolica `CPX_CALLBACK_SET` comunica a CPLEX che sono stati aggiunti tagli;

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

---

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

---

Successivamente è necessario assegnare al parametro **\*useraction\\_p** il valore **CPX\_CALLBACK\_DEFAULT**. Per ottenere la soluzione del rilassamento continuo è necessario utilizzare il metodo **CPXgetcallbacknodex** avente come intestazione:

---

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

---

Dove:

---

<sup>39</sup>Essendo tali funzioni invocate automaticamente da CPLEX i tipi di parametri che esse ricevono sono stati definiti a priori e non risultano modificabili

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

Nel nostro caso dato che vogliamo conoscere tutte le variabili, assegniamo i valori 0 e  $[n*(n-1)/2]^{\sim}1$  rispettivamente ai parametri **begin** ed **end**.

All'atto dell'invocazione del metodo *CPXgetcallbacknode* non viene passato come parametro l'array **bestLb** contenuto in *inst* ma viene creato un opportuno array chiamato **xstar**:

---

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

---

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

## USERCUT CALLBACK IN C

Come già anticipato nei precedenti paragrafi questo tipo di callback sono utilizzate per gestire soluzioni frazionarie ottenute per i vari nodi dell'albero decisionale durante una risoluzione di tipo *B&C* per problemi di programmazione lineare da parte di CPLEX. A livello concettuale sono del tutto simili a quanto visto nel paragrafo precedente per le *lazy callback* in linguaggio **C**, è raccomandata una lettura del paragrafo precedente a loro dedicato in quanto di seguito saranno esposti estensivamente solamente i dettagli riguardanti la gestione dei tagli.<sup>40</sup>

L'installazione delle callback avviene tramite la funzione **CPXsetusercutcallbackfunc**:

---

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

---

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

---

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

---

<sup>40</sup>Notiamo in realtà che l'implementazione delle *usercut callback* avviene sempre in concomitanza all'implementazione delle *lazy callback* per tanto il settaggio del numero di thread è necessario solamente una volta

CPLEX, una volta calcolata una soluzione frazionaria, genera in automatico dei propri tagli<sup>41</sup>. Quando il parametro *wherefrom* risulta pari a **CPX\_CALLBACK\_MIP\_CUT\_LAST** significa che l'iterazione successiva da parte di CPLEX consisterebbe nell'operazione di branching sul nodo in questione: solo in questa condizione risulta conveniente generare i propri vincoli caratteristici del problema che si sta risolvendo. Qualora il parametro *wherefrom* assuma invece altri valori, si effettua una semplice **return 0** senza eseguire alcuna operazione, altrimenti come già discusso per le lazy, è necessario recuperare il puntatore all'istanza. Riprendendo quanto detto nel paragrafo LINK PARAGRAFO CALLBACK|||| è sconsigliato aggiungere *manualmente* ad ogni nodo dell'albero decisionale dei tagli in quanto il loro numero complessivo risulterebbe troppo elevato andando quindi a **peggiorare** le prestazioni di CPLEX. Per tale ragione, dopo alcuni test e secondo le linee guida discusse durante il corso, si è deciso che solamente con una probabilità del 10% la *usercut callback* da noi definita entra in gioco. Dato che l'id numerico assegnato ai nodi dell'albero decisionale, ottenuto attraverso la funzione **CPXgetcallbacknodeinfo**, non ha alcuna relazione diretta alla probabilità che venga generata una soluzione intera oppure frazionaria, è sufficiente effettuare una operazione di modulo dieci a tale valore: nel caso in cui il risultato sia pari a zero si procede con il calcolo dei tagli. Successivamente, invocando la nota funzione **CPXgetcallbacknodex** si ottiene la soluzione frazionaria. Prima di procedere con la parte principale di questo metodo, per ragioni di chiarezza riportiamo il codice che esegue quanto finora descritto:

---

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

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

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

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

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

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

---

Da questo momento inizieremo ad utilizzare le funzionalità offerta da *Concorde*, tutte le funzioni il cui nome inizia con la sigla "**CC**" sono importate da quest'ultimo. L'aggiunta di eventuali

---

<sup>41</sup>Ad esempio taglio di *Gomory*

vincoli di *subtour elimination* avviene tramite l'invocazione in un primo momento della funzione **CCcut\_connect\_components** la quale identifica le componenti connesse della soluzione ricevuta come parametro indipendentemente dal fatto che sia intera o frazionaria.

Di seguito sono riportati nel dettaglio tutti i parametri che tale funzione vuole ricevere in ingresso, si osserva che mentre i primi 4 costituiscono l'effettivo input della funzione i rimanenti 3 sono in realtà settati al suo interno e quindi possono essere visti come parametri di output:

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

---

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

---

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

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

---

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

---

La chiamata alla funzione risulta quindi essere:

---

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

---



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

---

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

---

Dove:

- **env, cbdata, wherefrom**: parametri noti già discussi nella callback *myLazyCallback*;
- **nzcnt**: numero di coefficienti diversi da zero del vincolo;
- **rhs**: definisce il termine noto del vincolo;
- **sense**: può assumere i seguenti valori:
  - **cutind**: array di *nzcnt* elementi contenenti gli indici delle variabili presenti nel vincolo;
  - **cutval**: array di *nzcnt* elementi contenenti i corrispondenti valori dei coefficienti;
  - **purgeable**: valore intero che specifica come CPLEX deve trattare il taglio:
    - \* **CPX\_USECUT\_FORCE**: il taglio una volta aggiunto al rilassamento non può essere più rimosso;
    - \* **CPX\_USECUT\_PURGE**: il taglio è aggiunto al rilassamento ma può essere eliminato in un secondo momento se giudicato inefficiente;
    - \* **CPX\_USECUT\_FILTER**: il taglio deve essere trattato come se generato da CPLEX il quale prima di aggiungerlo al rilassamento lo analizza e può quindi decidere di abortire l'operazione di aggiunta. nel rilassamento(per esempio è già presente un taglio più efficiente);

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

---

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

    for (int c = 0; c < ncomp; c++)
    {
        int dimIndexValue = compscount[c] * (compscount[c] - 1) /
            2;
```

```

    int *cutind = (int*)malloc(dimIndexValue * sizeof(int));
    double *cutval = (double*)malloc(dimIndexValue *
        sizeof(double));
    int nzcnt = 0;

    k2 += compscount[c];

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

    k1 = k2 + 1;

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

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

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

return 0;
}

```

---

Nel caso in cui la soluzione presenti una sola componente connessa, come in Fig. X, invocando la funzione **CCcut\_violated\_cuts** di *Concorde* è possibile individuare gli insiemi  $S$  che soddisfino la disuguaglianza (10): noto  $S$  risulta poi banale inserire il relativo vincolo di subtour. In particolare *CCcut\_violated\_cuts* è una funzione in grado di individuare sezioni di capacità inferiori ad una certa soglia. Descriviamo quindi i 7 parametri che tale funzione riceve in input:

- **int ncount, int ecount, int \*elist**: il loro significato è già stato descritto per la funzione *CCcut\_connect\_components*;
- **dlen**: vettore contenente la capacità di ogni lato;
- **cutoff**: [Questo è il termine noto della disequazione f2, non ho capito perchè devo togliere a 2 un EPSILON, così è scritto nel pdf condiviso dal prof che si chiama RO2\_TSPutilities]

- **(\*(doit\_fn)(double, int, int \*, void \*))** : è una funzione creata da noi che risulta essere una vera e propria callback: ogniqualvolta *Concorde* individua un insieme  $S$  cercato tale funzione viene invocata. Al suo interno, grazie ai parametri forniti<sup>42</sup> è nostro compito procedere all'ampliamento del pool di tagli di *CPLEX*.
- **pass\_param**: puntatore ad una struttura dati contenente variabili e puntatori che devono essere accessibili all'interno della callback;

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

---

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

---

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

---

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

---

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

---

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

---

Dove:

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

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

---

<sup>42</sup>Maggiori dettagli riguardo i quattro parametri di ingresso saranno forniti a breve durante la descrizione di come l'implementazione di tale funzione è stata da noi realizzata.

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

## SEZIONE OTTO

52

come abbiamo già potuto vedere nelle varianti del metodo **Loop** presentate, prendono il nome di algoritmi euristici: la soluzione che offrono sarà sempre ammissibile ma non viene garantita la sua ottimalità, al contrario nella maggior parte dei casi, soprattutto per istanze complesse, questa non viene quasi mai raggiunta. È inoltre tenere ben presente che esistono molteplici algoritmi euristici più o meno potenti, essendo inoltre tecniche non esatte, è possibile trovarne infinite varianti per ognuno di essi. In generale, come meglio vedremo nel seguito del testo, possiamo classificarli in **costruttivi**, **migliorativi**, **metaeuristici** e **matheuristics**. Quanto esposto fino ad ora ci porta intuitivamente a pensare che la bontà di un metodo proposto può subire enormi variazioni in base a quali istanze su cui viene applicato.

## ALGORITMI COSTRUTTIVI GREEDY

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

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

## ALGORITMO NEAREST NEIGHBOR

L'algoritmo **nearest neighbour** è stato uno dei primi algoritmi utilizzati per risolvere istanze del problema del commesso viaggiatore. La sua dimostrazione di correttezza non viene qui riportata in quanto non risulta di interesse ai fini del nostro progetto ed è inoltre ampiamente discussa in letteratura. Limitiamoci quindi ad una descrizione del concetto fondamentale alla base dell'algoritmo: dato un qualsiasi percorso (soluzione) parziale, il modo migliore per ampliarlo è banalmente attraverso l'aggiunta di un nuovo arco a costo **minore** avente come estremi uno dei due nodi liberi<sup>43</sup> del percorso stesso mentre il secondo sia uno ancora disponibile<sup>44</sup>.

Entriamo ora più nel dettaglio nella realizzazione di questo algoritmo descrivendone i passaggi fondamentali. Noteremo che sono state fatte delle scelte di programmazione che in un primo momento potrebbero apparire in motivate se considerate limitatamente a quanto visto fino ad ora. L'algoritmo preso in analisi produce una soluzione valida molto velocemente ma che risulta essere scadente per quanto riguarda il suo costo, nel nostro progetto di conseguenza il *nearest neighbour* viene unicamente utilizzato per fornire una o più soluzioni di partenza per altri algoritmi euristici. Risulta

<sup>43</sup>Nodi attraversati del percorso parziale ma aventi un solo lato incidente.

<sup>44</sup>Non ancora attraversato dal circuito parziale.

```

Procedure Greedy( $E, F, S$ ):
  begin
     $S := \emptyset; Q := E;$ 
    repeat
       $e := \text{Best}(Q); Q := Q \setminus \{e\};$ 
      if  $S \cup \{e\} \in F$ 
        then  $S := S \cup \{e\}$ 
    until  $Q = \emptyset;$ 
  end.

```

Fig. 6: Algoritmo Greedy

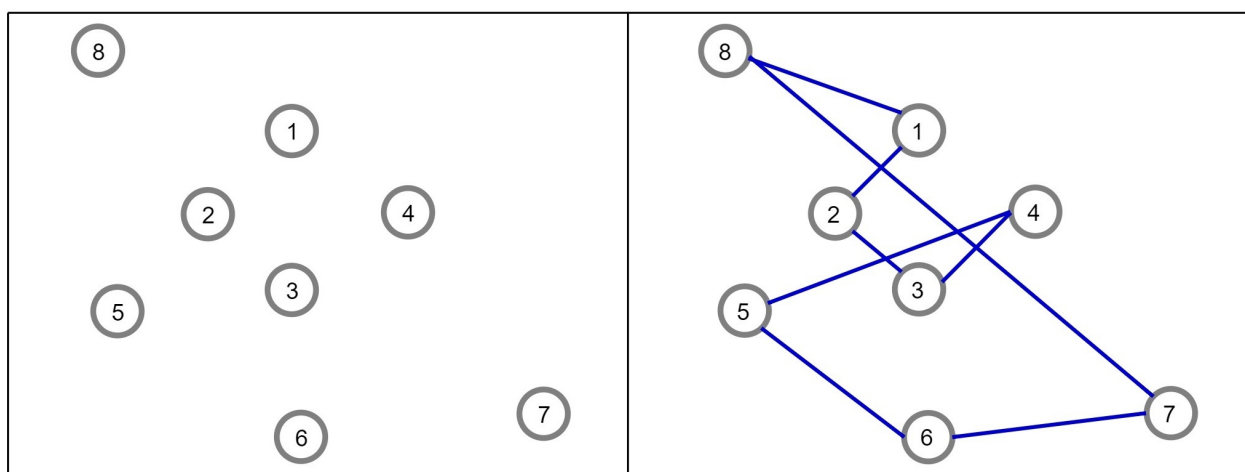


Fig. 7: Esempio di algoritmo Nearest Neighbour

pertanto di maggior importanza che al suo interno sia introdotta una certa randomicità nelle scelte che effettua in modo tale che multipli utilizzi sulla stessa istanza del TSP producano soluzioni tra loro scorrelate. Algoritmi che presentano tale caratteristica possono essere trovati in letteratura con la dicitura **GRASP**<sup>45</sup>

- Innanzi tutto, partendo da una soluzione vuota è necessaria una operazione preliminare prima di innescare la ricorsività dell'algoritmo. In altre parole dobbiamo fornire un nodo iniziale il cui lato incidente a costo minore diventa il la prima vera componente della soluzione che quindi da vuota diviene parziale. La scelta di quale debba essere il nodo di partenza avviene in modo casuale;
- Come sarà più chiaro in seguito, l'algoritmo *nearest neighbour* non produce mai soluzioni soddisfacenti ed è solamente utilizzato come punto di partenza per algoritmi più complessi. Questi ultimi necessitano in genere di una struttura dati che veda il circuito prodotto come un percorso orientato. Seguendo questa linea di pensiero, supponendo che l'ultima iterazione abbia introdotto il nodo  $j$  nel percorso parziale, il lato successivo che andremo a selezionare dovrà sempre essere incidente in  $j$ . In questo modo risulta molto più semplice tenere traccia della soluzione come un vero e proprio percorso orientato, essendo questo utilizzato solamente come input per altri metodo più complessi la nostra scelta non risulta in alcun modo limitante;
- Per introdurre un ulteriore livello di casualità nell'algoritmo, la scelta di quale lato entra a far parte della soluzione parziale non ricade sempre in quello a costo minore. In particolare dopo varie prove si è deciso che quest'ultima opzione avviene con una percentuale del 90%, mentre con il 9% la scelta ricade nel secondo miglior lato e con il restante 1% nel terzo miglior lato;
- Spesso può capitare che il lato designato per essere aggiunto al percorso causerebbe la presenza di un cappio al suo interno. Naturalmente la soluzione finale risulterebbe non valida e quindi tale situazione deve essere evitata. Banalmente, nel caso in cui il lato in questione sia la  $i$ -esima scelta migliore, questo viene sostituito dalla  $(i+1)$ -esima miglior scelta. Naturalmente il tutto viene ripetuto iterativamente fino a che non si trova un lato accettabile;

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

---

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

    int currentIndex = rnd.Next(instance.NNodes);
    int startIndex = currentIndex;
```

---

<sup>45</sup>Greedy Randomly Adaptive Search Procedure.

```

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

availableIndexes[currentIndex] = true;

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

    int plus = RndPlus(rnd);

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

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

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

    return new PathGenetic(heuristicSolution, distHeuristic);
}

```

---



## ALGORITMI MIGLIORATIVI

Gli algoritmi euristici migliorativi si basano su un'idea estremamente semplice ed intuitiva: data una soluzione ammissibile  $\mathbf{x}$ , relativa ad un problema di ottimizzazione, viene esaminato se attraverso minime variazioni questa risulta migliorabile in termini di funzione obiettivo. In gergo più tecnico si parla di ricercare soluzioni *vicine* a quella attuale ma migliorative. Per poter definire il concetto di "vicinanza" è necessario discutere quello di **mossa**. Questa è una operazione di modifica (caratteristica dell'algoritmo migliorativo) che viene eseguita su  $\mathbf{x}$  e che ha come conseguenza la generazione di un **insieme** di soluzioni ammissibili le quali costituiscono un intorno di  $\mathbf{x}$ , indicato con  $N(\mathbf{x})$ . Si parla allora di  $\mathbf{y}$  vicina ad  $\mathbf{x}$  se e solo se differiscono tra loro per una sola mossa e quindi  $\mathbf{y} \in N(\mathbf{x})$ .

Una volta definito  $N(\mathbf{x})$  questo viene esplorato secondo due possibili strategie che sono **first improvement** e **steepest descent**. Nel primo caso l'esplorazione dell'intorno termina non appena si trova una soluzione migliore di quella corrente. Nel secondo caso, invece, l'esplorazione è completa e viene trovato il miglioramento più consistente.

Qualora esista una soluzione  $\mathbf{y}$  migliore di  $\mathbf{x}$ , il procedimento viene iterato esplorando  $N(\mathbf{y})$ ; viceversa l'algoritmo si arresta. Giunti a questo punto il risultato finale può essere una soluzione *localmente ottima* oppure, più raramente, globalmente ottima<sup>46</sup>. Poiché da un punto di vista matematico il processo di ricerca analizza, ad ogni interazione, un intorno della soluzione corrente, gli algoritmi migliorativi vengono anche chiamati *algoritmi di ricerca locale*.

Tranne alcuni casi particolari in cui la funzione obiettivo ha determinate caratteristiche di convessità, nella maggior parte dei problemi reali questa presenta un grande numero di minimi locali che spesso si discostano totalmente dell'ottimo globale. In effetti, una delle fortunate eccezioni è il metodo del simplesso per la programmazione lineare che si pone alla base degli studi in questo settore: esso fornisce sia un metodo per analizzare in un numero finito di passi tutti gli ottimi locali del problema e soprattutto se questi sono anche globali.

Tra i più famosi algoritmi migliorativi applicabili al problema del commesso viaggiatore troviamo i **K-Opt** dove **K** è in genere un numero intero superiore a 2. Procediamo quindi ad una loro descrizione generale seguita da una implementazione particolare della tecnica del **2-Opt**.

## ALGORITMO K-OPT

Gli algoritmi **K-Opt**, sigla inglese per K-Ottimalità, fanno parte della categoria degli algoritmi migliorativi e sono caratterizzati da una mossa, applicata ad un circuito hamiltoniano, consistente nello **scambio** di **K** archi con altrettanti non facenti parte del percorso producendone uno *vicino*, migliore e chiaramente valido. Come specificato nel paragrafo precedente **K** può assumere qualsiasi valore superiore a 2<sup>47</sup> ma in genere è proprio  $K = 2$  l'unica variante realmente utilizzata. Risulta infatti facilmente verificabile che più il suo valore è alto, più salgono sia la complessità computazionale che di scrittura

progettazione dell'algoritmo<sup>48</sup> perdendo così i vantaggi offerti dall'utilizzo di tecniche euristiche. Concentriamoci quindi unicamente nella variante *2-Opt*: presa una qualsiasi coppia di archi distinti

---

<sup>46</sup>Da notare che un ottimo è globale se lo è anche localmente

<sup>47</sup>Chiaramente **K** non può superare il numero di archi che costituiscono la soluzione.

<sup>48</sup>Vedremo in seguito che questa affermazione è valida solo per algoritmi che applicano direttamente la definizione di K-Ottimalità. Esistono infatti metodi che la ottengono indirettamente se sono caratterizzati da tempi di esecuzione molto buoni.

$([i, j]; [h, k])$ <sup>49</sup> è possibile sostituirli correttamente con una sola delle combinazioni  $([i, k]; [h, j])$  e  $([i, h]; [j, k])$ . Una di queste due, infatti, trasforma la soluzione in una seconda contenente due subtour. A discapito quindi di un concetto molto basilare, l'algoritmo *2-opt*<sup>50</sup> introduce la difficoltà di verificare quale delle due possibili sostituzioni è valida. L'approccio migliore in questi casi è di introdurre un fittizio ordinamento nel circuito attraverso una struttura dati di supporto apposita. Come è possibile infatti vedere dalla figura X, se gli archi sono ordinati, e quindi lo è anche il circuito, la difficoltà appena discussa è facilmente risolvibile:

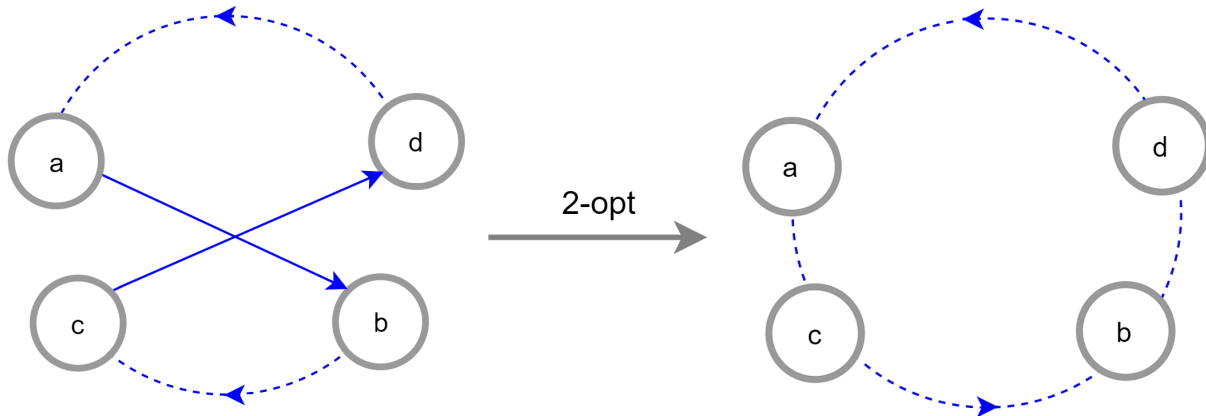


Fig. 8: Esempio di azione nell'algoritmo 2-opt

Viene da sé che applicare l'approccio *2-Opt* a qualsiasi coppia di archi non garantisce un miglioramento del costo della soluzione. Questo però avviene nel 100% dei casi se i due archi in questione visualmente si incrociano come mostrato nella figura X, tale affermazione è facilmente verificabile matematicamente. Nel complesso, nel caso in cui si voglia trovare l'operazione di due ottimalità migliore<sup>51</sup> è necessario confrontare tutte le coppie possibili ottenendo quindi una complessità computazionale pari a  $O(n^2)$  rispetto al numero di nodi mentre nel caso di un generico valore di  $K$  si passa ovviamente a  $O(n^3)$ . Nel caso del *TSP*, alcuni esperimenti svolti verso la fine degli anni '50 hanno mostrato che il passaggio da due ottimalità a tre ottimalità porta un miglioramento sensibile della qualità della soluzione trovata, che giustifica pienamente l'aumento di carico computazionale ma non il costo di scrittura

progettazione dell'algoritmo se paragonato ad altri metodi euristici che mostreremo nel seguito del testo. Miglioramenti praticamente trascurabili si hanno invece per  $K > 3$ . Concludiamo le nozioni riguardanti i metodi migliorativi di  $K$  ottimalità nel seguente paragrafo dove vengono mostrati i dettagli implementativi dell'algoritmo *2-Opt* da noi progettato

## METODO TwoOpt

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

<sup>49</sup>Notiamo che questi non possono mai essere presi consecutivi e cioè con un vertice in comune in quanto non ci sarebbe modo di produrre una soluzione *vicina* valida.

<sup>50</sup>In generale lo stesso discorso può applicarsi anche a tutto il resto della famiglia di algoritmi.

<sup>51</sup>Cioè trovare la soluzione vicina a costo più basso

---

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

---

Dove:

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

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

---

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

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

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

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

        if (distAC + distBD < distTotABCD)
        {
            Utility.SwapRoute(c, b, pathG);
            pathG.path[a] = c;
            pathG.path[b] = d;
            pathG.cost = pathG.cost - distTotABCD + distAC + distBD;
            indexStart = 0;
        }
    }
}
```

```

        cnt = 0;
        found = true;
        //"break" = "first improvement" technique
        break;
    }

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

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

} while (cnt < instance.NNodes);

```

---

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

## MULTISTART

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

---

```

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

```

```

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

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

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

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

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

```

---

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

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

```

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

```

---

I numeri random sono generati, a partire da un valore d' inizializzazione chiamato **seme**, da un algoritmo matematico. Per sua natura l' algoritmo è deterministico: se si fornisce in input lo stesso seme genererà sempre la medesima sequenza di numeri. Per tale ragione il valore del seme viene derivato dall'orologio di sistema all' atto della creazione dell' istanza della classe `Random` qualora si utilizza il costruttore di default. In questo modo, non essendo predicibile il valore del seme, la sequenza di numeri generati dall' algoritmo risulta essere sistematicamente casuale.

L' orologio di sistema non sempre risulta un buon valore da utilizzare per settare il seme. Si supponga di avere la necessità di creare due oggetti diversi della classe `Random`: qualora quest' ultimi siano creati uno di seguito all' altro avranno entrambi il medesimo seme poichè l' orologio di sistema risulta lo stesso nel lasso di tempo che il processore impiega ad eseguire le due istruzioni.

Per constatare ciò si è realizzato il seguente programma:

```

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

```

---

```

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

```

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

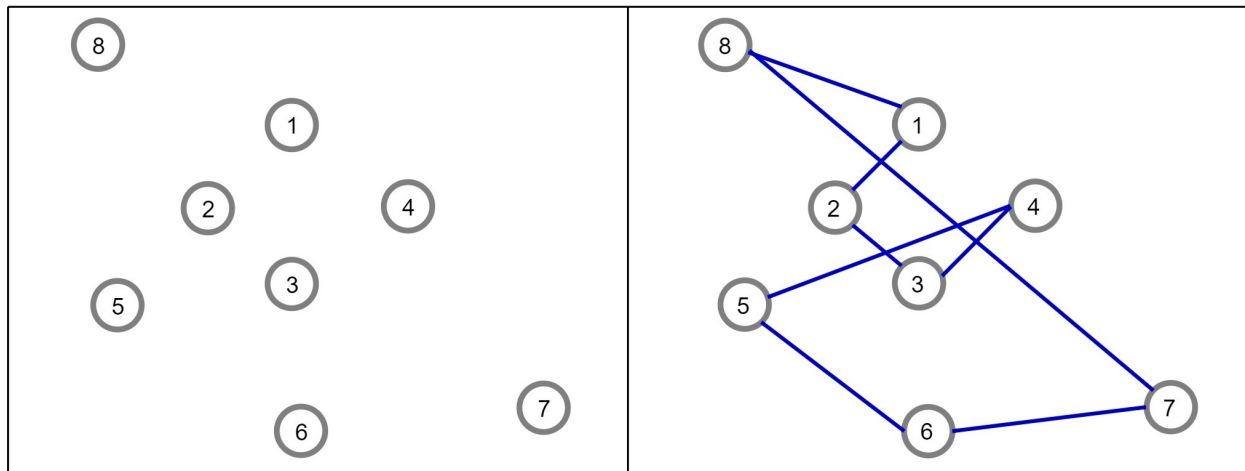


Fig. 9: Output

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

## METAEURISTICI

### TABU

### VNS

## ALGORITMI GENETICI

Gli Algoritmi Genetici (*AG*), proposti nel 1975 da J.H. Holland, sono un modello computazionale idealizzato dall'evoluzione naturale darwinista. L'aggettivo "genetico" deriva dal fatto che il modello evolutivo darwiniano trova spiegazioni nella branca della biologia detta genetica e dal fatto che tali algoritmi attuano meccaniche concettualmente simili a quelli dei processi biochimici scoperti da questa scienza. I principi fondamentali che consentono la nascita e lo sviluppo di un processo evolutivo che porta all'evoluzione di una specie sono la **selezione naturale** e la **varietà del genotipo**<sup>52</sup> della popolazione. La selezione naturale è il meccanismo grazie al quale si ha un progressivo e cumulativo aumento della frequenza degli individui aventi caratteristiche ottimali per l'ambiente in cui essi vivono poiché solo quelli che meglio si adattano ad un certo habitat riescono a sopravvivere e a riprodursi. I meccanismi generatori della variazione del genotipo della popolazione sono sostanzialmente due:

<sup>52</sup>Il termine *genotipo* indica la costituzione genetica di un organismo o di un gruppo di individui

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

I cambiamenti che si verificano da una generazione all'altra risultano essere molto piccoli ma, dato che sopravvivono soprattutto quelli positivi, un loro accumulo porta nel tempo a grandi cambiamenti. La ricerca parte da una popolazione iniziale di individui, detti cromosomi, che rappresentano ipotetiche soluzioni al problema dato. Ogni individuo della popolazione viene codificato da un vettore<sup>53</sup> i cui elementi contengono simboli appartenenti ad un alfabeto finito, detti geni. Ad ogni soluzione è associato un valore determinato da una funzione chiamata **Fitness** il cui scopo è di determinare la bontà di un individuo nel risolvere il problema in questione. Così come nella natura solamente gli individui che meglio si adattano all'ambiente sono in grado di sopravvivere e riprodursi, anche negli algoritmi genetici le soluzioni migliori sono quelle che hanno la maggiore probabilità di trasmettere i propri geni alle generazioni future. Come vedremo in seguito sono fondamentalmente tre le caratteristiche determinanti per un algoritmo genetico: determinare quale funzione di fitness si andrà ad utilizzare, partendo dalla attuale generazione decidere come creare un pull di possibili candidati per quella successiva ed infine come selezionare tra questi ultimi quelli che sopravviveranno. Essendo la definizione delle funzione di fitness direttamente dipendente da quale tipo di problema si desidera studiare, concludiamo questa introduzione elencando solamente quali operatori genetici è possibile applicare per definire le restanti due caratteristiche di un algoritmo.

## OPERATORI GENETICI

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

### OPERATORE DI CROSSOVER

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

- **Crossover ad un punto:** date due soluzioni si tagliano i loro vettori di codifica in un punto casuale o predefinito per ottenere due teste  $\{H_a, H_b\}$  e due code  $\{T_a, T_b\}$ , si possono costruire quindi altrettante soluzioni distinte combinando la testa di un genitore con la coda dell'altro  $S_1 = H_a \cup T_b, S_2 = H_b \cup T_1$ ;
- **Crossover a due punti:** date due soluzioni si tagliano i loro vettori di codifica in due punti predefiniti o casuali al fine di ottenere una coppia di teste  $\{H_a, H_b\}$ , parti centrali  $\{I_a, I_b\}$  ed code  $\{T_a, T_b\}$ . Le due soluzioni sono ottenute scambiando le due parti centrali nei genitori  $S_1 = H_a \cup I_b \cup T_a, S_2 = H_b \cup I_a \cup T_b$ ;

---

<sup>53</sup>Oltre alla codifica vettoriale in letteratura è possibile trovare anche quella ad albero. Tuttavia essa viene utilizzata per codificare gli individui della popolazione nell'ambito della programmazione genetica (che è?????????).

- **Crossover uniforme:** consiste nello scambiare casualmente elementi tra le soluzioni candidate all'evoluzione;
- **Crossover aritmetico:** consiste nell'utilizzare un'operazione aritmetica per creare la nuova soluzione, ad esempio eseguendo una *XOR* o una *AND* tra elementi dei genitori se interpretati come una sequenza binaria;

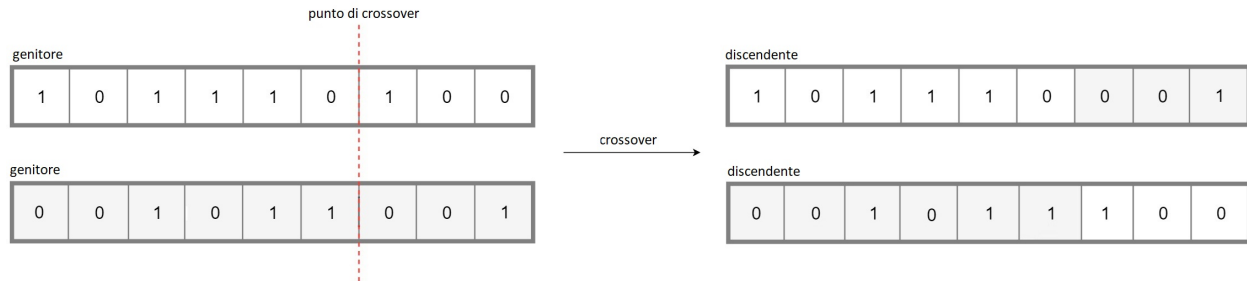


Fig. 10: Esempio di operatore di crossover

## OPERATORI DI SELEZIONE

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

- **Selezione a roulette:** la probabilità che una soluzione venga scelta per far parte della successiva generazione è direttamente proporzionale al valore restituito dalla funzione di fitness. Immaginiamo quindi di avere a disposizione una roulette la cui ruota viene divisa in sezioni tutte assegnate ai vari candidati, la loro grandezza è quindi proporzionale all'idoneità dell'individuo. La selezione è banalmente ottenuta con molteplici rotazioni della roulette tenendo conto che un individuo non può essere selezionato più volte. Questa tecnica presenta dei problemi nel caso in cui le sezioni della ruota risultino tra loro eccessivamente sbilanciate in ampiezza, le soluzioni peggiori vengono selezionate troppo raramente e questo per quanto già esposto non è necessariamente un bene;
- **Selezione di Boltzmann:** le soluzioni vengono scelte con un grado di probabilità che, agli inizi dell'algoritmo, favorisce l'*esplorazione* mentre più avanti tende a stabilizzarsi. Questa tecnica ritiene utile, in un primo momento, consentire agli individui meno idonei di riprodursi quasi quanto quelli migliori, e far procedere lentamente la selezione così da mantenere una certa diversità all'interno della popolazione. In seguito si rafforza la selezione per favorire maggiormente gli individui ad alta idoneità, presumendo che la fase iniziale, con grande diversità e poca selezione, abbia consentito alla popolazione di individuare la zona giusta nello spazio di ricerca;



- **Selezione a torneo:** da un pool di possibili soluzioni, nel caso più comune vengono scelti in modo del tutto casuale sia due individui che un numero  $c \in [0, 1]$ . Se quest'ultimo risulta minore di un parametro  $k \in [0, 1]$  fissato, si seleziona il più idoneo tra i due candidati, altrimenti la scelta ricade sul peggiore. Naturalmente si procede fino a quando non ho tutti gli elementi per la generazione successiva.

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

## OPERATORI DI MUTAZIONE

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

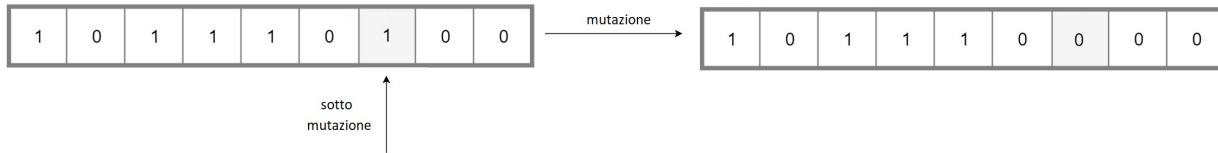


Fig. 11: Esempio di mutazione

### 0.1 ALGORITMO GENETICO TSP

Gli *AG* risolvono un determinato problema generando sempre nuove **popolazioni** di soluzioni dove in genere troviamo una fitness media piuttosto bassa, giungendo solamente dopo diverse generazioni a valori più elevati. Per poter applicare un algoritmo genetico, occorre anzitutto codificare numericamente le soluzioni e individuare una opportuna funzione di fitness. La codifica vettoriale dei cromosomi più adatta per i problemi di TSP risulta essere un vettore di interi dove ogni elemento identifica in maniera univoca una delle città da visitare mentre il suo posizionamento identifica l'ordine di visita. La funzione di fitness realizzata riceve in ingresso una soluzione **ammissibile**<sup>54</sup> e restituisce un valore reale pari al reciproco del suo costo PERCHE'????? LO DICIAMO DOPO?????????. La prima generazione viene ottenuta attraverso il metodo **NearestNeighborGenetic** della classe **Utility**. Come facilmente intuibile genera soluzioni che tendono a collegare nodi tra loro vicini, per maggiori dettagli si consulti l'apposita sezione ad esso dedicata nella appendice LINKK!!!!!!!!!!!!!! La generazione di un *figlio* a partire da due *genitori* avviene attraverso il crossover ad un punto già presentato dove però solamente una delle due soluzioni ottenute entra a far parte del pool di candidati per la successiva generazione. Il crossover viene fornito dal metodo **GenerateChild**, sempre appartenente alla classe **Utility**, che viene descritto nell'apposito paragrafo LINK!!!!!!!!, mentre i restanti condidati alla nuova generazione sono gli elementi stessi della generazione precedente (motivo?!?!?!). Come operatore di selezione, si è deciso di utilizzare la *selezione a*

<sup>54</sup>Per qualsiasi generazione non sono quindi accettabili elementi non validi per il problema in questione. La funzione XXXX LINK!!! è stata realizzata a a tale scopo.

*roulette* dato che le soluzioni hanno lo stesso ordine di grandezza per quanto riguarda i loro costi QUI NON E' CHIARO CHE SIGNIFICA—INTENDI CHE LE FETTE CHE VENGONO FUORI NON SONO TROPPO DIVERSE??????. Infine la mutazione avviene con probabilità  $p_{mutex}$  pari all'1%. Quanto descritto viene per la maggior parte gestito attraverso il metodo NOME!!!!!!!!!!!!!! con l'ausilio della classe **pathgenetic** descritta nella apposita sezione LINK!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!.

## FUNZIONE GENETICALGORITHM

Questa classe ha il compito di amministrare tutto tutte le fasi dell'algoritmo genetico. Per prima cosa si procede con una dichiarazione ed inizializzazione delle varie strutture dati necessarie. Tra queste troviamo due liste di *PathGenetic* chiamate **OriginallyPopulated** e **ChildPoulation** che, durante tutto il processo, contengono rispettivamente l'insieme dei circuiti hamiltoniani che compongono la generazione  $i$  –esima ed i figli da loro generati. La prima generazione viene ottenuta attraverso il metodo **NearestNeighborGenetic**, discusso in maggiore dettagli nella appendice LINK!!!!, che, ricevendo la lista completa per ogni nodo di quali sono ad esso più vicini<sup>55</sup>, costruisce le varie istanze tendendo a collegare i nodi più vicini tra loro<sup>56</sup>. Durante la generazione dei nuovi figli, poiché l'indice in cui è memorizzato un circuito all'interno di *OriginallyPopulated*, dovuto in genere all'estrazione della roulette, è casuale<sup>57</sup> si è deciso di accoppiare circuiti memorizzati in celle adiacenti; la casualità di *OriginallyPopulated* consente di combinare fra loro soluzioni buone con altre meno buone e ciò statisticamente risulta particolarmente vantaggioso. Una volta prodotti i figli è necessario procedere con la creazione della nuova generazione padre: questo viene eseguito dal metodo **NextPopulation** descritto in seguito. Per identificare il miglior circuito della generazione corrente si è realizzato il metodo **BestSolution**, qualora il valore ottenuto risulti minore dell'incumbent<sup>58</sup> quest'ultimo viene aggiornato. L'algoritmo termina quando scade il time limit fornito dall'utente. Riportiamo di seguito il codice realizzato:

---

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

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

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

//Generate the first population
for (int i = 0; i < sizePopulation; i++)
    OriginallyPopulated.Add(Utility.NearestNeighborGenetic(instance, rnd,
        true, listArray));
do
{
    //Generate the children
    for (int i = 0; i < sizePopulation; i++)
    {
```

---

<sup>55</sup>Funzione già descritta qui LINK!!!!!!!!!!!!!!.

<sup>56</sup>Il numero di componenti per ogni generazione viene chiesto in input all'utente e passato come parametro di ingresso alla funzione *GeneticAlgorithm*.

<sup>57</sup>Con casuale si intende che non è presente alcuna forma di correlazione fra l'indice e il costo della soluzione.

<sup>58</sup>Con incumbent si intende la miglio soluzione fin ora calcolata.

```

if (i % 2 != 0)
ChildPoulation.Add(Utility.GenerateChild(instance, rnd,
    OriginallyPopulated[i], OriginallyPopulated[i - 1], listArray));
}

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

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

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

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

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

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

```

---

## NEARESTNEIGHTBORGNETIC

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

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

L' intestazione del metodo risulta essere:

```

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

```

---

Dove:

- **instance**: oggetto della classe *Instance* contenente tutti i dati che descrivono l'istanza del problema del Commesso Viaggiatore fornita in ingresso dall' utente;

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

Il circuito prodotto dal metodo viene memorizzato all'interno del vettore **heuristicSolution** avente una dimensione pari al numero di nodi del grafo. Poiché si vogliono soluzioni che siano il più possibile dissimili tra loro, è consigliabile fare in modo che *rndStartPoint* sia posta a *true*<sup>59</sup>. Il vettore **VisitedNodes** di tipo bool è un vettore di supporto che memorizza all'indice **i** il valore logico true se il nodo **i** è già stato visitato, false altrimenti.

---

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

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

int firstNode = 0;

//rndStartPoint define if the starting point is random or always the node
0
if (rndStartPoint)
firstNode = rnd.Next(0, instance.NNodes);

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

---

Una volta definito il nodo di partenza i restanti nodi, che se visitati rispettando il loro ordine compongono un circuito hamiltoniano, son ottenuti attraverso un ciclo *for*: alla generica iterazione **i** del ciclo, sfruttando la struttura dati listArray e la funzione **RndGenetic** si memorizza all'interno della variabile **nextNode** il nodo successivo visitato dal percorso sempre che questo sia ancora disponibile. Per verificarne la disponibilità si utilizza l'array *VisitedNodes*, qualora non sia possibile utilizzare tale nodo si passa al successivo più vicino.

(la variabile contatore de ciclo for è inizializzata al valore 1, quindi il heuristicSolution[i-1] è memorizzato l' ultimo nodo visitato) QUESTO VA COME COMMENTO NEL CODICE

---

```
for (int i = 1; i < instance.NNodes; i++)
{
bool found = false;
int candPos = RndGenetic(rnd);
int nextNode = listArray[heuristicSolution[i - 1]][candPos];
```

---

<sup>59</sup>Da notare che tale parametro non è settata runtime ma solamente via hardcoded.

```

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

```

---

## GENERATECHILD

Per generare un figlio si è realizzato il metodo **GenerateChild**, appartenente alla classe *Utility*, avente la seguente intestazione:

---

```

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

```

---

Dove:

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

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

---

```

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

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

```

---

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

---

```

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

```

---

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

---

```

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

```

---

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

## NEXTPOPULATION

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

---

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

---

Dove:

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

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

---

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

---

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

---

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

---

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

---

```
List<int> NumbersExtracted = new List<int>();
```

```

bool find = false;
int numberExtracted;

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

```

---

## MATHEURISTICS

Gli algoritmi **matheuristics** nascono con l'obiettivo di migliorare una soluzione di partenza ammissibile  $\mathbf{x}$  sfruttando il **modello matematico** del problema (ogni tanto parli di MIP, ma si intende modelli misti interi e frazionari, è giusto usarlo???) che si vuole risolvere. Analogamente a quanto visto per gli algoritmi di ricerca locale e metaeuristici, i matheuristici tentano di individuare una soluzione migliore  $\mathbf{x}^*$  all'interno di un intorno  $N(\mathbf{x})$  ottenuto, in questo ambito, modificando opportunamente il modello matematico di partenza.

L'esplorazione di  $N(\mathbf{x})$  non avviene per enumerazione come visto in precedenza ma tramite appositi solver come ad esempio CPLEX: grazie alla loro sempre maggiore ottimizzazione, questa operazione risulta quindi essere molto veloce.

I matheuristici possono essere applicati a qualunque soluzione ammissibile  $\mathbf{x}$ ; ciò nonostante per apprezzarne davvero la potenza è consigliabile utilizzare un punto di partenza già buono. A tale scopo si può quindi pensare di concatenarli all'esecuzione di un algoritmo migliorativo o, meglio ancora, al termine del multistart o di un metaeuristico. Questi infatti arrivano molto frequentemente a saturare in prossimità di una soluzione molto buona  $\bar{\mathbf{x}}$  ma non ottima.

Sfruttando questa combinazione di algoritmi è stato dimostrato che già in un breve lasso di tempo si riescono ad ottenere sostanziali miglioramenti ed in alcuni casi è possibile raggiungere anche l'ottimo globale. Si osserva che esiste una notevole differenza fra l'ottenere quest'ultimo risultato grazie ad un algoritmo esatto e seguendo il procedimento precedentemente discusso: mentre nel primo caso è sempre certificato, nel secondo questo non avviene<sup>60</sup> e non si ha nemmeno modo di certificarlo a meno che non lo si conosca a priori.

---

<sup>60</sup>Il solver utilizzato garantisce solamente l'ottimalità del modello da noi fornitogli che chiaramente non può essere quello originale del problema.



A questo punto è chiaro come il fattore caratterizzante di un algoritmo matheuristico sia come questo definisce l'intorno della soluzione di partenza sul quale si suppone possano essere localizzati dei miglioramenti.

Uno schema comunemente utilizzato prende il nome di **hard variable fixing** e prevede di fissare il valore di un certo numero di variabili, rendendole di fatto costanti, mentre le restanti vengono lasciate libere. Lo svantaggio principale derivante da questa tecnica è che a priori non esistono indicazioni su cosa è più opportuno fissare e cosa lo è meno. Di conseguenza esiste una sua variante detta **soft variable fixing** che lascia al solver questa responsabilità indicandogli solamente con quale proporzione deve avvenire la cosa.

Nei prossimi paragrafi sono riportanti alcuni esempi di algoritmi matheuristici che si basano sui concetti appena esposti ricordando che come per qualsiasi metodo euristico non esistono varianti più o meno efficaci in assoluto ma molto dipende dal problema che si sta analizzando e da fattori casuali non calcolabili.

## HARD FIXING

L'hard fixing è una tecnica euristica che, partendo da una qualsiasi soluzione ammissibile  $x_H$ , tenta di trovarne una migliore all'interno di un suo intorno. Quest'ultimo viene definito dall'algoritmo stesso attraverso il fissaggio a priori di alcune variabili del modello matematico, rendendole così di fatto costanti, utilizzando i valori che queste assumo in  $x_H$ .

Come già discusso nel paragrafo precedente, una volta definito l'intorno e quindi un nuovo modello matematico, la sua risoluzione viene affidata ad un solver esterno pertanto l'unica fase saliente dell'algoritmo è la scelta stessa di quale porzione di  $x_H$  mantenere valida e quale no.

Esistono chiaramente molteplici approcci ammissibili e nessuno di questi risulta migliore in assoluto. Come spesso accade negli algoritmi euristici la scelta più semplice produce risultati molto buoni e quindi alcune varianti dell'*hard fixing* semplicemente fissano un qualunque lato con probabilità  $p$  o meno con probabilità  $1 - p$ <sup>61</sup>.

Altri approcci invece tentano di definire un sistema di ranking come ad esempio dare più valore agli elementi con costo minore.

Indipendentemente da quale tecnica si decida di utilizzare maggiore è il numero dei lati fissati, minore è il carico di lavoro per risolvere il rilassamento continuo derivante ma allo stesso tempo diviene minore anche l'ampiezza dell'intorno che si va a sondare.

Nel complesso quindi è buona norma tentare sì molteplici fissaggi ma anche variarne il numero cercando di trovare la miglior combinazione per il problema in questione.

Nel nostro caso si è deciso di optare per la prima variante andando progressivamente a diminuire il valore di  $p$ . Il motivo di tale scelta è che il nostro algoritmo non ha un target specifico di istanze aventi una struttura ben definita dove diventa quindi evidente un sistema di ranking efficiente.

Preferiamo quindi ampliare il range di possibili intorni da visitare dando però sempre la possibilità all'algoritmo di trovare la soluzione ottima. Di conseguenza più è alto il tempo limite che gli andiamo a concedere maggiori sono le probabilità di avvicinarsi al risultato desiderato.

## WARM-START

Tra le svariate funzionalità offerta da CPLEX possiamo trovarne una in particolare che è sempre consigliato utilizzare quando possibile e nello specifico risulta essere una naturale aggiunta alla

---

<sup>61</sup>Ovviamente  $p$  è direttamente proporzionale a quanti lati sul loro totale vogliamo fissare.

tecnica dell'**hard-fixing**. Stiamo parlando della possibilità di fornire a CPLEX una soluzione al problema che dovrà andare a risolvere in modo tale che, una volta controllata la sua validità durante la fase di preprocessing, possa essere utilizzata come incumbent. Questa soluzione prende il nome di **warm-start** o **MIP-start** e chiaramente quando si procede al fissaggio di alcune variabili in accordo ad  $x_H$ , questa rimane ovviamente valida per il problema e quindi è una naturale candidata per questo ruolo.

CPLEX trae diversi potenziali vantaggi grazie ad un **warm-start**: prima di tutto sfruttando il criterio di bounding, avere a disposizione un buon incumbent fin dai primi momenti della tecnica del *branch&cut* può velocizzare la costruzione dell'albero decisionale dichiarandone alcuni nodi come sondati; In secondo luogo permette di applicare particolare tecniche euristiche che necessitano di questa condizione di partenza, parliamo ad esempio degli algoritmi **RINS** e **polishing** che saranno discussi successivamente anche in questa tesi.

In linguaggio **C#** l'operazione di warm-start avviene attraverso il metodo non statico **AddMIP-Start** della classe *CPLEX* la cui firma è:

---

```
public virtual int AddMIPStart(INumVar[] vars, double[] values)
```

---

Dove:

- **vars**: Vettore contenente nella posizione  $i$  la  $i$  – esima variabile del modello;
- **values**: Vettore contenente nella posizione  $i$  il valore della  $i$  – esima variabile del modello;

Si osserva inoltre che nel caso in cui si attuino consecutive esecuzione dell'*hard-fixing* non è più necessario aggiornare manualmente il valore di *warm-start*<sup>62</sup>: variare solamente i fissaggi delle variabili, senza modificare ulteriormente il modello matematico, non viene considerata da CPLEX una operazione *drastica* al punto di abbandonare i risultati ottenuti fino a quel momento.

In altre parole, nel caso in cui non sia trovata una soluzione migliore rispetto al *warm-start* attuale la validità di quest'ultimo viene automaticamente ricontrollata durante la successiva fase di preprocessing<sup>63</sup> effettuata dal solver. Altrimenti nel caso in cui l'ultimo rilassamento continuo abbia individuato un nuovo valore per l'incumbent, e quindi un nuovo *warm-start*, CPLEX provvede autonomamente al suo aggiornamento.

## PREPROCESSINGTSP

Una volta definito l'intorno  $N(x_I)$  e quindi aver fissato il *lower bound* di alcune variabili a 1 è possibile procedere immediatamente alla risoluzione del problema modificato attraverso il solver, nel nostro caso CPLEX. Come ben noto, nel caso più generico, il pool di tagli è inizialmente vuoto e solo quelli necessari, non determinabili a priori, sono individuati attraverso le callback installate. Nel caso dell'**hard-fixing** questa affermazione non è del tutto esatta in quanto **alcuni** dei tagli **necessari**<sup>64</sup> possono essere trovati molto velocemente attraverso una analisi del fissaggio effettuato. Una loro aggiunta preventiva risulta quindi molto vantaggiosa in quanto si evitano certamente delle iterazioni al solver risparmiando nel complesso diverso tempo. In **FiguraX** troviamo una generica

---

<sup>62</sup>Se non alla prima run.

<sup>63</sup>La sua validità è sempre garantita in quanto siamo nel caso in cui si applichiamo un fissaggio diverso ma rispetto la stessa soluzione di partenza dato che non ne sono state trovate di migliori.

<sup>64</sup>Per necessari si intende che sicuramente

situazione dove gli archi in *blu* sono stati *fissati* mentre quelli in *azzurro* no. Proprio questi ultimi però verrebbero immediatamente selezionati dal solver in quanto la soluzione con i due subtour indicati risulta banalmente la meno costosa. Il compito del preprocessingTSP (**ppTSP**) da noi progettato è quindi proprio questo, trovare tutti quei lati che, se anche selezionati singolarmente, generano un subtour<sup>65</sup> e di conseguenza forzarli a valore 0.

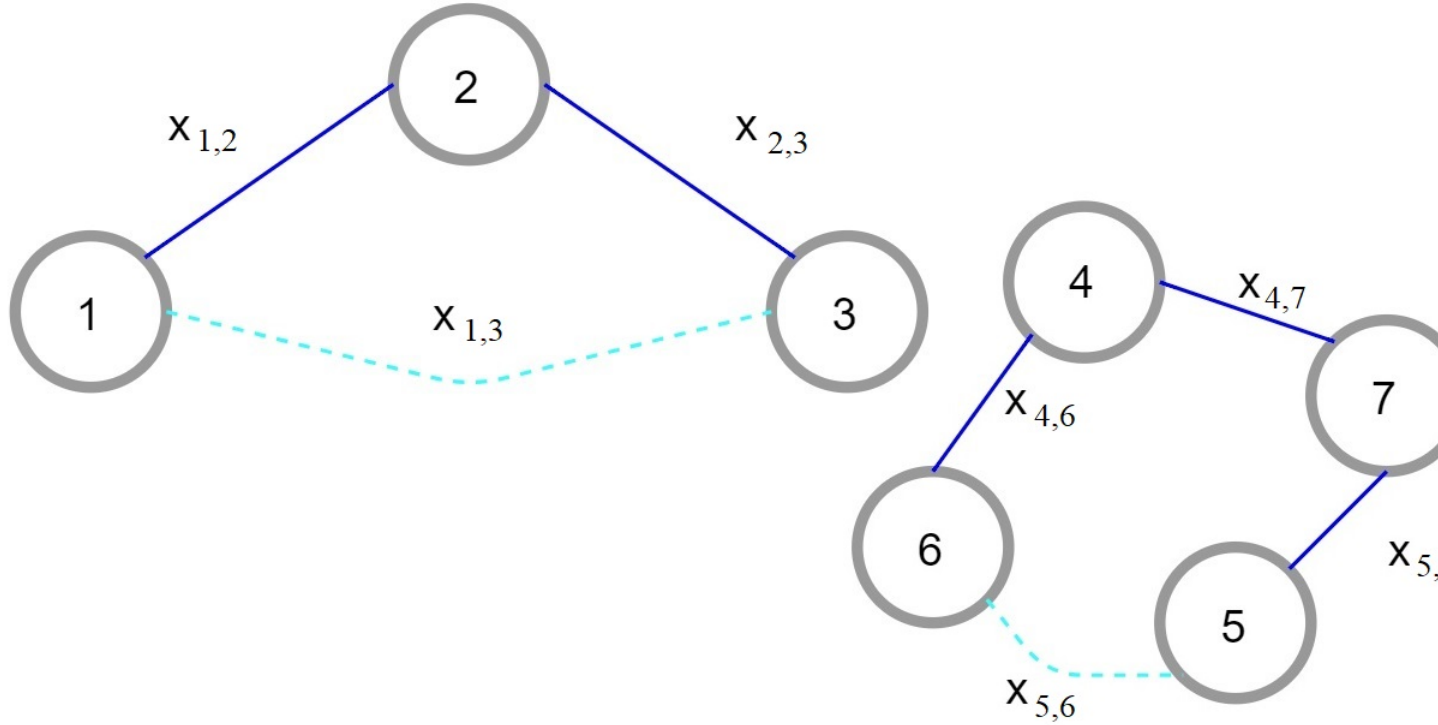


Fig. 12: Preprocessing

A livello algoritmo l'operazione di *ppTSP* è molto semplice da realizzare. Sfruttiamo la stessa tecnica utilizzata nel corso del progetto per individuare le componenti connesse di una **soluzione** proposta. Queste nel nostro caso non risulteranno mai essere un ciclo completo per costruzione e quindi si possono sempre individuare le coppie di nodi posti alle loro estremità: i lati che collegano i due elementi appartenenti alla stessa coppia sono esattamente quelli da forzare a valore nullo<sup>66</sup>. Una definizione equivalente di quanto stiamo cercando sono tutti quei nodi posti alla estremi di un solo lato che è stato fissato. Proponiamo ora il codice commentato che sfrutta quest'ultima affermazione e permette di attuare la completa fase di *ppTSP*:

---

```
public static void PreProcessingTSP(Instance instance, INumVar[] x)
```

---

<sup>65</sup>Dato che non ha senso fissare  $n - 1$  lati su un totale di  $n$ , se la singola selezione di un lato genera un subtour, qualsiasi siano le restanti selezioni, la soluzione così prodotta avrà sempre almeno un secondo subtour e quindi risulta invalida.

<sup>66</sup>Questo discorso è sempre valido a meno che la componente connessa non ciclica trovata sia composta da un solo lato, in quel caso non è richiesta nessuna operazione.

Dove:

- **instance**: riferimento all' oggetto contenente tutte le informazioni relative all' istanza del Problema del Commesso Viaggiatore corrente;
- **x**: vettore contenente le variabili del modello matematico;

---

```
//Questo vettore tiene traccia all'indice i-esimo di quante volte il nodo
i è estremo di un lato fissato, opzioni valide sono 0, 1, 2
int[] cntNode = new int[instance.NNodes];

//Questo vettore tiene traccia di quale componente connessa fa parte ogni
nodo
int[] compConn = new int[instance.NNodes];

//Questo vettore di liste di interi è la struttura dati di supporto dove
infine ogni lista contenente due elementi sarà relativa ad una singola
//componente connessa e immagazzinerà al suo interno gli indici dei nodi
posti alle sue estremità. Chiaramente tale informazione è facilmente
//ricavabile dai due precedenti vettori cntNode e compConn ma con un
costo complessivo più alto.
List<int>[] externalNodes = new List<int>[instance.NNodes];

//Classica inizializzazione delle componenti connesse, ad ogni nodo è
assegnato ne è assegnata una propria
Utility.InitCC(compConn);

//Inizializzazione di externalNodes
for (int i = 0; i < instance.NNodes; i++)
    externalNodes[i] = new List<int>();

for (int i = 0; i < instance.NNodes; i++)
{
    for (int j = i + 1; j < instance.NNodes; j++)
    {
        //Trovo indice corretto del lato con estremi i nodi di indice i e
        j
        int position = Utility.xPos(i, j, instance.NNodes);

        //Nel caso sia stato fissato dall'hard-fixing lo si analizza
        if (x[position].LB == 1)
        {
            //Aggiornamento del contatore relativo ai nodi i e j
            cntNode[i] += 1;
            cntNode[j] += 1;

            //Aggiorno componenti connesse seguendo la tecnica di Kruskal
            for (int k = 0; k < instance.NNodes; k++)
            {
                if ((k != j) && (compConn[k] == compConn[j]))
```

```

        compConn[k] = compConn[i];
    }

    compConn[j] = compConn[i];
}
}

//Popoliamo correttamente externalNodes in modo tale che solamente i due
//nodi estremi della stessa componente connessa
//siano inserite all'interno della stessa lista
for(int i = 0; i < instance.NNodes; i++)
{
    if (cntNode[i] == 1)
        externalNodes[compConn[i]].Add(i);
}

//Controlliamo quali elementi del vettore di liste appena popolato
//risultano avere esattamente due elementi ed eseguiamo
//l'operazione fondamentale del preprocessing discussa
for (int i = 0; i < instance.NNodes; i++)
{
    if (externalNodes[i].Count == 2)
    {
        //Calcolo l'indice del lato di cui voglio forzare l'upper bound a
        0
        int pos = xPos(externalNodes[i][0], externalNodes[i][1],
            instance.NNodes);

        //Se il lato analizzato era in precedenza stato fissato ad uno
        //dall'algoritmo di hard fixing, quindi il suo lower bound
        //è pari ad 1, non lo andiamo a toccare in quanto significa che
        //la componente connessa non ha altri elementi
        if (x[pos].LB == 0)
            x[pos].UB = 0;
    }
}

```

---

## IMPLEMENTAZIONE HARD FIXING

---

```

static void HardFixing(CPLEX cplex, Instance instance, Process process,
    Random rnd, Stopwatch clock)

```

---

Dove:

- **cplex**: riferimento all'oggetto contenente il modello da risolvere;
- **instance**: riferimento all'oggetto contenente tutte le informazioni relative all'istanza del Problema del Commesso Viaggiatore corrente;

- **process**: riferimento all'oggetto necessario per la stampa dei dati attraverso *GNUPlot*;
- **rdn**: istanza della classe Random utilizzata per la generazione dei numeri casuali;
- **clock**: riferimento all'oggetto utilizzato come cronometro, deve già essere stato avviato;;

Di seguito troviamo il codice del metodo **commentato**. Facciamo però presente che l'*hard fixing* può ricevere in ingresso una qualsiasi soluzione di partenza, preferibilmente già buona, sulla quale vogliamo definire un intorno di ricerca. Nella versione base dell'algoritmo riportato viene utilizzato il modo più veloce per ottenere tale risultato e quindi attraverso la funzione **NearestNeighbor LINK!!!!!!!!!!!!!!** con conseguente algoritmo di due ottimalità.

Nei test a fine capitolo vedremo invece una variante che sfrutta soluzioni già precedentemente ottenute da altri algoritmi euristici attraverso loro tempi di esecuzione infinitamente più alti.

---

```
//Oggetto utilizzato per la scrittura su file dei vari incumbent trovati
//in modo tale che GNUPlot possa stamparne il ciclo a video
StreamWriter file;

//Vettore che vuole essere sempre aggiornato al miglior ciclo trovato
double[] currentIncumbentSol = new double[(instance.NNodes - 1) *
    instance.NNodes / 2];

//Variabile sempre aggiornata al costo del miglior ciclo trovato
double currentIncumbentCost = Double.MaxValue;

//All'indice i-esimo troviamo in ordine crescente quali sono i nodi più
//vicini all'i-esimo nodo
List<int>[] listArray = Utility.BuildSLComplete(instance);

//Variabile booleana che comunicherà alla lazy callback utilizzata di non
//stampare a video tutte gli aggiornamenti dell'incumbent che trova
bool BlockPrint = false;

//Indica dopo quante consecutive esplorazioni di diversi intorni sia
//necessario ampliare l'ampiezza dei successivi
const int VALUECONSITENOTIMPROV = 3;

//Variabile costantemente aggiornata a quante esplorazioni consecutive
//non hanno portato ad una soluzione migliore, inizialmente si parte da
//VALUECONSITENOTIMPROV
int consecutiveiterationNotImprov = VALUECONSITENOTIMPROV;

//La percentuale iniziale p che ha ogni lato di essere fissato
double percentageFixing = 0.8;

//Semplice inizializzazione del vettore che andrà a contenere la
//selezione o meno di ogni lato nella soluzione migliore finale
instance.BestSol = new double[(instance.NNodes - 1) * instance.NNodes /
    2];

//Creo il modello matematico di partenza
```

```

INumVar[] x = Utility.BuildModel(cplex, instance, -1);

//Utilizzo del NearestNeighbor per iniziare a costruire la soluzione di
partenza
PathStandard heuristicSol = Utility.NearestNeighbor(instance, rnd,
    listArray);

//Applicazione del 2-Opt per completare la soluzione di partenza
TwoOpt(instance, heuristicSol);

//La soluzione di partenza è settata come attuale incumbent
for (int i = 0; i < instance.NNodes; i++)
{
    int position = Utility.xPos(i, heuristicSol.path[i], instance.NNodes);
    currentIncumbentSol[position] = 1;
}
//Il costo della soluzione di partenza è settato come il miglior costo
attuale
currentIncumbentCost = heuristicSol.cost;

//Installazione lazy callback per la risoluzione del modello
TSPLazyConsCallback tspLazy = new TSPLazyConsCallback(cplex, x, instance,
    process, BlockPrint);
cplex.Use(tspLazy);

//Forniamo a CPLEX il warm-start
cplex.AddMIPStart(x, currentIncumbentSol);

//Setto i thread utilizzati da CPLEX pari al numero di core virtuali
della macchina utilizzata, il codice della Lazy è thread-safe
cplex.SetParam(CPLEX.Param.Threads, cplex.GetNumCores());

//La politica da noi scelta prevede che al primo miglioramento ottenuto
durante l'esplorazione dell'intorno attuale questa venga fermata in
modo da ripetere il procedimento
//in modo da ripetere il procedimento con il nuovo valore di incumbent.
Per ottenere questo risultato comunichiamo a CPLEX che alla seconda
soluzione intera trovata fermi
//la propria risoluzione. Notiamo che la prima soluzione intera è il
warm-start
cplex.SetParam(CPLEX.LongParam.IntSolLim, 2);

do
{
    //Modifichiamo il modello introducendo il fissaggio di alcune
    variabili creando in questo modo un intorno della soluzione
    migliore attuale
    Utility.ModifyModel(instance, x, rnd, percentageFixing,
        currentIncumbentSol);

    //Risolviemo il modello attuale

```

```

cplex.Solve();

//Se otteniamo un miglioramento
if (currentIncumbentCost > cplex.GetObjValue(CPLEX.IncumbentId))
{
    //Preparo il file
    file = new StreamWriter(instance.InputFile + ".dat", false);

    //Aggiorno le variabili con l'attuale percorso e costo incumbent
    currentIncumbentCost = cplex.GetObjValue(CPLEX.IncumbentId);
    currentIncumbentSol = cplex.GetValues(x, CPLEX.IncumbentId);

    //Stampa del nuovo percorso incumbent su file
    for (int i = 0; i < instance.NNodes; i++)
    {
        for (int j = i + 1; j < instance.NNodes; j++)
        {
            int position = Utility.xPos(i, j, instance.NNodes);

            if (currentIncumbentSol[position] >= 0.5)
            {
                file.WriteLine(instance.Coord[i].X + " " +
                    instance.Coord[i].Y + " " + (i + 1));
                file.WriteLine(instance.Coord[j].X + " " +
                    instance.Coord[j].Y + " " + (j + 1) + "\n");
            }
        }
    }
    //Chiusura del file di tipo StreamWriter, il flush dal buffer
    viene eseguito
    file.Close();

    //Stampa attraverso GNUPlot del nuovo incumbet path
    Utility.PrintGNUPlot(process, instance.InputFile, 1,
        currentIncumbentCost, -1);

    //Resettiamo il numero di successivi non miglioramenti
    consecutiveiterationNotImprov = VALUECONSITENOTIMPROV;
}
else
{
    //Il numero di consecutivi non miglioramenti è decrementato
    consecutiveiterationNotImprov--;
}

//Se ho raggiunto il limite di consecutivi non miglioramenti
if (consecutiveiterationNotImprov == 0)
{
    //Se la percentuale di fissaggio di un nodo è ancora superiore al
    20% la diminuisco
    if (percentageFixing > 0.2)

```



```

    {
        //Diminuzione della percentuale di fissamento di un nodo di
        //un 10%
        percentageFixing -= 0.1;
        //Resetto il numero di consecutivi non miglioramenti in
        //seguito alla variazione dell'ampiezza dell'intorno da
        //analizzare
        consecutiveIterationNotImprov = VALUECONSITENOTIMPROV;
    }
}

//Elimino l'ultimo fissaggio delle variabili in preparazione al
//prossimo
for (int i = 0; i < x.Length; i++)
{
    x[i].LB = 0;
    x[i].UB = 1;
}

} while (clock.ElapsedMilliseconds / 1000.0 < instance.TimeLimit);
//Ripeto il tutto fino a che non scade il timelimit, l'ultima iterazione
//può sfolarlo fino a che non è completa

//Aggiorno la variabili instance con la soluzione migliore finale ed il
//relativo costo
instance.XBest = currentIncumbentCost;
instance.BestSol = currentIncumbentSol;

//Stampo righe vuota nell'output standard di CPLEX
cplex.Output().WriteLine();
cplex.Output().WriteLine();
//Stampo il costo della miglior soluzione trovata nell'output standard di
//CPLEX
cplex.Output().WriteLine("x = " + instance.XBest + "\n");

```

---

## LOCAL BRANCH

Terminata l'analisi di una possibile variante della tecnica *hard-fixing* proseguiamo con la tipologia gemella del **soft-fixing**: si ricordi che la principale differenza tra i due approcci è che il secondo lascia al solver, nel nostro caso CPLEX, anche l'operazione di fissaggio indicandogli solamente la forma che questo deve assumere.

Nello specifico prendiamo in considerazione una particolare tipologia di algoritmi facente parte del ramo *soft-fixing* e cioè il **Local Branch**: proposto nell'anno 2002 dai docenti universitari **Matteo Fischetti** <sup>67</sup> e **Andrea Lodi** <sup>68</sup>, questa variante mira ad ottenere l'esplorazione di un intorno  $N(x, r)$  di una soluzione  $x$  con un costo computazionale estremamente inferiore a  $O(n^r)$  che invece abbiamo visto essere necessario per un classico algoritmo di *r-ottimalità*.

<sup>67</sup>Docente presso l' Università di Padova, Dipartimento di Ingegneria dell' Informazione.

<sup>68</sup>Docente presso l' Università di Bologna, Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione.

Il *Local Branch* non utilizza la classica tecnica di esplorazione per enumerazione di  $N(x, r)$  ma introduce una specifica disequazione al modello matematico del problema *TSP*:

$$\sum_{e : x_e=1} x_e^* \geq n - r \quad (11)$$

Attraverso questo vincolo è il solver stesso a generare un intorno  $N(x, r)$  sul quale cercare la soluzione migliore al suo interno. In altre parole, nel nostro esempio, sfruttiamo la velocità offerta da CPLEX nel risolvere un modello matematico e costruiamo quest'ultimo così che ammetta tutte le soluzioni a loro volta valide per un algoritmo di *r-ottimalità*.

Prendendo più in esame (11), è possibile notare come la sua elevata potenza sia in pieno contrasto alla sua estrema semplicità. Tra tutte le variabili disponibili, sono considerate solamente quelle facenti parte della soluzione  $x$  attuale<sup>69</sup> e si impone che almeno  $n - r$  dovranno far parte della nuova soluzione prodotta dal solver. Sono quindi ammessi fino a  $r$  scambi tra variabili in soluzione e non, producendo di fatto *gratuitamente* un'operazione di *r-ottimalità*.

La forma espressa dalla (11) prende il nome di formulazione **asimetrica** in quanto non appaiono direttamente le variabili non selezionate da  $x$ . È presente anche una sua variante **simmetrica** che risulta essere sia più esplicita che maggiormente *complessa*:

$$\underbrace{\sum_{j : x_j=0} x_j^*}_{\# \text{ variabili che passano da 0 a 1}} + \underbrace{\sum_{j : x_j=1} (1 - x_j^*)}_{\# \text{ variabili che passano da 1 a 0}} \leq r \quad (12)$$

Come si può vedere da una breve analisi le due forme (11) e (12) sono del tutto equivalenti, nello specifico (12) definisce la massima distanza di **Hamming**<sup>70</sup> che può sussistere fra  $x^*$  e  $x$ <sup>71</sup> rispettivamente la nuova e la attuale soluzione.

Esponiamo due concetti riguardanti le due formulazioni proposte. Come prima cosa (12), al contrario di (11), nel caso in cui voglia simulare una operazione  $k-Opt$  deve porre  $r = 2 * k$  in quando devo indicare sia i lati che aggiungo rispetto a  $x$  ma anche quelli che tolgo da quest'ultima.

In secondo luogo la decisione di quale delle due formulazioni adottare dipende dal tipo di problema: notiamo infatti che in (12) appaiono tutte le variabili del problema mentre in (11) solamente quelle facenti parte della soluzione attuale. Nel caso in cui queste due quantità risultino paragonabili allora le due formulazioni sono entrambe valide. In caso contrario, come ad esempio i modelli di TSP analizzati in questa tesi dove la differenza è di un ordine di grandezza, (12) risulta molto più densa di (11) e quindi più difficilmente gestibile dal solver.

Concludiamo questa introduzione teorica con una considerazione sui valori assegnabili ad  $r$  nella (11). Sappiamo che una operazione di  $k-ottimalità$  comprende anche tutte le possibili  $m-Opt$  dove  $k > m$  pertanto più alto viene posto il valore di  $r$  più diventa ampio l'intorno  $N(x, r)$  dove andiamo a cercare la nuova soluzione e quindi le possibilità di ottenere un risultato migliore. Naturale conseguenza è anche una maggiore complessità computazionale sia nel caso di una banale esplorazione per enumerazione di  $N(x, r)$  sia attraverso l'uso del *Local Branching*. Non è mai pertanto consigliabile oltrepassare certi limiti, direttamente dipendenti dal tipo di macchina sulla quale vengono fatti

<sup>69</sup>Il cui numero totale pari al numero di nodi indicato con  $n$ .

<sup>70</sup>La distanza di *Hamming* fra due vettori di pari dimensione, corrisponde al numero di posizioni aventi simboli corrispondenti diversi.

<sup>71</sup>,

eseguire gli algoritmi, in quanto si viene a parte la fondamentale caratteristica degli euristici: la loro velocità.

## IMPLEMENTAZIONE LOCAL BRANCHING

Presentiamo ora il codice **commentato** che realizza la tecnica *Local Branching* in versione **asimmetrica** per le motivazioni esposte nel precedente paragrafo.

Come per il metodo **hard fixing** qui viene riportato un versione basilare del codice che crea una soluzione di partenza attraverso la applicazione consecutiva del NN LINK!!!!!! e  $2 - Opt$  LINK!!!!. Nella parte finale della tesi dove sono presentati vari test, saranno invece utilizzate soluzioni migliore ottenute in precedenza dall'applicazione di altri metodi euristici.

Infine riprendendo quanto detto alla fine dell'ultimo paragrafo specifichiamo che i valori di  $r$  utilizzati<sup>72</sup> sono solamente 3, 5, 7, 10, raggiungibili in progressione una volta che i precedenti falliscono<sup>73</sup>.

---

```
static void LocalBranching(CPLEX cplex, Instance instance, Process
    process, Random rnd, Stopwatch clock)
```

---

Dove:

- **cplex**: riferimento all'oggetto contenente il modello da risolvere;
- **instance**: riferimento all'oggetto contenente tutte le informazioni relative all'istanza del Problema del Commesso Viaggiatore corrente;
- **process**: riferimento all'oggetto necessario per la stampa dei dati attraverso *GNUPlot*;
- **rdn**: istanza della classe Random utilizzata per la generazione dei numeri casuali;
- **clock**: riferimento all'oggetto utilizzato come cronometro, deve già essere stato avviato;

---

```
//Vettore contenente i possibili valori del raggio r che definisce
    l'intorno nel quale il metodo cerca
//una soluzione migliore della attuale
int[] possibleRadius = {3, 5, 7, 10};

//Variabile che memorizza l'indice dove trovare l'attuale raggio in
    possibleRadius, inizializzata a 0
int currentRange = 0;

//Variabile booleana utilizzare per comunicare alla Lazy Callback di
    CPLEX se deve stampare la soluzione
//intera trovata nel caso migliore quella incumbent
bool BlockPrint = false;

//Inizializzazione del vettore contenente la soluzione incumbent
```

---

<sup>72</sup>Dove  $r$  determina quale operazione di  $r - Opt$  si ottiene.

<sup>73</sup>Cioè se non si trova una soluzione migliore di quella già disponibile all'interno del suo intorno  $N(x, r)$ .

```

double[] incumbentSol = new double[(instance.NNodes - 1) *
    instance.NNodes / 2];

//Inizializzazione variabile contenente il costo della soluzione incumbent
double incumbentCost = double.MaxValue;

//Inizializzazione del vettore presente in instance sul quale viene
    memorizzata la soluzione finale migliore
instance.BestSol = new double[(instance.NNodes - 1) * instance.NNodes /
    2];

//Creo il modello iniziale e assegno ad x il riferimento alle sua
    variabili
INumVar[] x = Utility.BuildModel(cplex, instance, -1);

//All'indice i-esimo troviamo in ordine crescente quali sono i nodi più
    vicini all'i-esimo nodo
List<int>[] listArray = Utility.BuildSLComplete(instance);

//Prima parte della creazione della soluzione di partenza, si utilizza la
    tecnica del nearest neigh
PathStandard heuristicSol = Utility.NearestNeighbor(instance, rnd,
    listArray);

///Seconda parte della creazione della soluzione di partenza, si applica
    un semplice 2-Opt
TwoOpt(instance, heuristicSol);

//Aggiornamento del vettore incumbentSol secondo la soluzione iniziale
    appena prodotta
for (int i = 0; i < instance.NNodes; i++)
{
    int position = Utility.xPos(i, heuristicSol.path[i], instance.NNodes);

    incumbentSol[position] = 1;
}

//Installazione Lazy Callback
cplex.Use(new TSPLazyConsCallback(cplex, x, instance, process,
    BlockPrint));

//Settaggio per il multi-thread di CPLEX, si utilizzano tanti thread
    quanti i core virtuali della macchina
cplex.SetParam(CPLEX.Param.Threads, cplex.GetNumCores());

//Aggiunta di un warm-start per CPLEX
cplex.AddMIPStart(x, incumbentSol);

//Inizio creazione del vincolo asincrono caratterizzante il Local
    Branching

```

```

//Creao il contenitore per l'espressione del vincolo di Local Branching
ILinearNumExpr expr = cplex.LinearNumExpr();

//Popolo il contenitore secondo le variabili utilizzate dalla soluzione
di partenza appena creata
for (int i = 0; i < instance.NNodes; i++)
    expr.AddTerm(x[Utility.xPos(i, heuristicSol.path[i],
        instance.NNodes)], 1);

//Dovendo ad ogni applicazione del Local Branching sostituire il vincolo
caratterizzante, si memorizza
//quello attuale in una variabile apposita in modo tale da facilitare la
sua rimozione futura
IAddable localBranchConstraint = cplex.Ge(expr, instance.NNodes -
    possibleRadius[currentRange]);

//Aggiungo il vincolo creato non come un qualsiasi taglio ma come una
vero e proprio vincolo
//del modello matematico, la risoluzione da parte di CPLEX è nettamente
migliorata
cplex.Add(localBranchConstraint);

//Inizio ciclo do-while che ripete la tecnica del Local Branching fino al
termine del tempo limite
//indicato dall'utente o quando il raggio r=10 non produce più
miglioramenti
do
{
    //CPLEX risolve l'attuale modello
    cplex.Solve();

    //Se trovo un miglioramento rispetto alla soluzione incumbent
    if (incumbentCost > cplex.GetObjValue())
    {
        //Sostituisco l'incumbent attuale con i valori appena trovati da
        CPLEX
        incumbentCost = cplex.ObjValue;
        incumbentSol = cplex.GetValues(x);

        //Rimozione del vincolo caratterizzante attualmente utilizzato
        nel modello
        cplex.Remove(localBranchConstraint);

        //Preparo il nuovo vincolo caratterizzante sfruttando
        l'esplorazione della nuova soluzione
        //per la sua stampa a video attraverso GNUPlot

        expr = cplex.LinearNumExpr();

        StreamWriter file = new StreamWriter(instance.InputFile + ".dat",

```

```

        false);

//Scandisco ogni variabile per vedere se fa parte della nuova
//soluzione
for (int i = 0; i < instance.NNodes; i++)
{
    for (int j = i + 1; j < instance.NNodes; j++)
    {
        int position = Utility.xPos(i, j, instance.NNodes);

        //Testo se l'attuale variabile è stata selezionata nella
        //nuova soluzione
        if (incumbentSol[position] >= 0.5)
        {
            //Stampo su file il percorso della nuova soluzione
            //per GNUPlot
            file.WriteLine(instance.Coord[i].X + " " +
                instance.Coord[i].Y + " " + (i + 1));
            file.WriteLine(instance.Coord[j].X + " " +
                instance.Coord[j].Y + " " + (j + 1) + "\n");

            //In contemporanea aggiungo il termine
            //all'espressione per il nuovo vincolo
            //caratterizzante
            expr.AddTerm(x[position], 1);
        }
    }
}

//Lancio la stampa attraverso GNUPlot e chiudo il flusso del file
//utilizzato
file.Close();
Utility.PrintGNUPlot(process, instance.InputFile, 1,
    incumbentCost, -1);

//Completo la creazione del nuovo vincolo caratterizzante
localBranchConstraint = cplex.Ge(expr, instance.NNodes -
    possibleRadius[currentRange]);

//Aggiungo il nuovo vincolo caratterizzante al modello matematico
cplex.Add(localBranchConstraint);
}
else
{
    //Nel caso in cui non si sia trovata una soluzione migliore testo
    //se posso aumentare il range r
    if (possibleRadius[currentRange] != 10)
    {
        //Aumento l'indice di possibleRadius sui cui trovare il nuovo
        //raggio da utilizzare
        currentRange++;
    }
}

```

```

//Rimuovo il precedente vincolo
cplex.Remove(localBranchConstraint);

//Aggiorno il range r del vincolo caratterizzante mantenendo
    chiaramente le variabili
//coinvolte dato che la soluzione incumbent rimane la
    medesima della precedente iterazione
localBranchConstraint = cplex.Ge(expr, instance.NNodes -
    possibleRadius[currentRange]);

//Aggiungo il vincolo caratterizzante aggiornato al modello
    matematico
cplex.Add(localBranchConstraint);
}
else
{
    //Nel caso non sia possibile aumentare ulteriormente il
        raggio r termino il ciclo do-while
    break;
}
}
} while (clock.ElapsedMilliseconds / 1000.0 < instance.TimeLimit);

//Memorizzo nelle apposite variabili di instance la soluzione finale
    trovata ed il relativo costo
instance.BestSol = incumbentSol;
instance.BestLb = incumbentCost;

```

---

## POLISHING

L'algoritmo **polishing**<sup>74</sup> è di fatto l'ultima tecnica presentata in questa tesi e fa parte degli algoritmi **matheuristici**.

*Polishing* non introduce di per sè novità ma bensì combina le migliori caratteristiche degli algoritmi **genetici** LINK!!!! e di **Relaxation Induced Neighborhood Search** conosciuto più comunemente con la sigla **RINS**. Quest'ultimo nasce nel 2004 per opera di **Emilie Danna**, **Edward Rothberg**<sup>75</sup> e **Claude Le Pape**. È stato ampiamente discusso nei precedenti paragrafi come nei metodi **matheuristici** il fattore caratterizzante sia la realizzazione del *fissaggio* di alcune variabili del modello matematico in modo tale da limitare le possibili soluzioni e quindi i tempi di esecuzione da parte del solver. *RINS* definisce una regola molto semplice a tale scopo: date due soluzioni  $x_1$  e  $x_2$ , valide per il **MIP** preso in analisi, ogni variabili che assume lo stesso valore in entrambe deve essere fissata nel nuovo modello a tale valore. Naturalmente più  $x_1$  ed  $x_2$  sono *simili* tra loro, più variabili fisso e quindi minore diventa la dimensione dell'intorno che il solver esplora.

*RINS* sembra quindi un buon metodo da applicare in presenza di due soluzioni molto simili a quella ottima  $x_{opt}$ , è plausibile assumere che in tal caso il fissaggio vada a riguardare in gran parte valori

---

<sup>74</sup>Polishing è utilizzato anche all'interno di CPLEX stesso ed è inoltre brevettato da quest'ultimo, la realizzazione è da attribuirsi ad Edward Rothberg

<sup>75</sup>Uno dei fondatori di **GuRoBi**, software solver concorrente a **CPLEX**.

concordi ad  $x_{opt}$ . La proprietà più importante è da vedersi nella estrema velocità di questa tecnica nel caso di problemi come il **TSP**, dove con un ordine del numero di variabili pari a  $O(n^2)$  solamente **n** assumo valore 1 in una qualsiasi soluzione mentre le restanti 0<sup>76</sup>. Come conseguenza diretta si ha che indipendentemente da come sono state ottenute  $x_1$  e  $x_2$  la probabilità che una qualsiasi variabili assuma valore 0 in entrambe è estremamente elevata.

Ed è proprio in questo punto che entrano in gioco i metodi euristici **genetici**, durante la produzione di ogni nuova generazione si presenta la necessità di creare un gran numero di nuove soluzioni a partire da quelle già presenti. Questi algoritmi affrontano quindi la possibilità di applicare *RINS* in modo continuo su soluzioni via via più *vicine* a quella ottima. Quanto appena esposto si pone alla base degli algoritmi **polishing**.

## PROBLEMATICHE DELLA TECNICA POLISHING

*Polishing* eredita una problematica fondamentale dagli algoritmi *genetici* dai noi volutamente non risolta in questi ultimi in modo tale evidenziarne l'importanza: stiamo parlando della possibilità dell'algoritmo di saturare dopo un certo periodo inversamente proporzionale alla grandezza della popolazione scelta. Mano a mano che nuove generazioni di popolazione vengono create, la probabilità che queste contengano soluzioni sempre più simili tra loro cresce portando così ad una saturazione per quanto riguarda la ricerca di un miglioramento<sup>77</sup>.

Naturalmente questo fattore non è sempre un male in quanto se la saturazione dovesse avvenire nelle vicinanze del valore ottimo lo scopo dell'algoritmo potrebbe essere considerato raggiunto in quanto metodo euristico. Questo fatto diventa però più probabile all'aumento della grandezza della popolazione e quindi del numero di soluzioni facenti parte di ogni generazione con l'inevitabile aumento dei tempi di calcolo.

Come evidenziato dai test riguardanti l'algoritmo genetico da noi implementato LINK!!!!, la problematica esposta è molto evidente già dalle istanze con meno nodi vista necessità di utilizzare una popolazione molto piccola. La situazione per quanto riguarda l'algoritmo *polishing* sarebbe stata la medesima indipendentemente dal fatto che quest'ultimo ci permettesse nella media di triplicare la grandezza di ogni generazione grazie all'uso di *RINS*.

La necessità di creare un metodo **anti saturazione** ci ha portato a ragione sul perchè gli algoritmi genetici offrono potenzialmente ottimi risultati: combinare soluzioni *buone* con altre *peggiori* può portare, attraverso varie generazioni, a risultati molto migliori di quelli raggiungibili con il solo uso di *buone* soluzioni.

Abbiamo quindi pensato di riapplicare questo principio in caso di saturazione e cioè sostituire una parte della generazione attuale con nuovi elementi, nel nostro caso generati nello stesso modo di quelli appartenenti alla prima generazione e quindi attraverso l'uso del **Nearestneighbour**<sup>78</sup> LINK!!!!!!!. Ipoteticamente la nostra idea è che in caso di saturazione le soluzioni facenti parte della attuale generazione abbiano ormai raggiunto un discreto fissaggio delle variabili al loro interno. Una loro combinazione con soluzioni potenzialmente peggiori attraverso *RINS* dovrebbe portare ad una ulteriore selezione interna al fissaggio appena citato mantenendo valida così solo la sua porzione migliore.

<sup>76</sup>Naturalmente anche il caso opposto ha pari valenza

<sup>77</sup>Verrà indicata come **saturazione di primo tipo** nei commenti al codice.

<sup>78</sup>Vedremo come in realtà è stato effettuato del leggero ulteriore tuning di questo metodo in base al numero di saturazioni consecutive senza alcun miglioramento dell'incumbent.



Indipendentemente da quale istanza sia stata da noi testata i risultati sono stati sorprendenti, con il progredire delle varie risoluzioni la probabilità di ottenere un miglioramento solamente dopo una operazione di anti saturazione continuava ad aumentare.

Il successivo nodo da discutere è naturalmente come capire quando ci si trova in una situazione di saturazione. I due fattori da noi considerati sono stati i seguenti:

- **Numero di generazioni successive senza miglioramenti dell'incumbent:** il valore da noi considerato è stato di **10**;
- **Tempo impiegato per definire una nuova generazione:** questo parametro è più complesso del precedente in quanto dipende da diversi fattori quali l'hardware utilizzato e l'istanza utilizzata. Abbiamo quindi svolto diversi test e la formula che per noi ha prodotto il risultato desiderato per le istanze testate è

$$t_{GenTot}[s] = 0.085 * (\#Nodi/10) * grandezzaPopolazione \quad (13)$$

Esiste una seconda problematica molto meno evidente e che può essere ricondotta ad un diverso tipo di saturazione, questa volta interno ad ogni generazione<sup>79</sup>. Al crescere della complessità dell'istanza analizzata, in genere determinata dal numero di nodi  $n$  e quindi da quello dei lati/variabili, può capitare che pochi *figli* vengano generati in tempi molto più lunghi rispetto ai restati della stessa generazione. La conseguenza è quindi un *ritardo* dell'algoritmo nel passaggio alla iterazione successiva senza però nessuna garanzia di un guadagno finale in termini di bontà dell'incumbent.

La nostra scelta è stata quindi di limitare questa situazione preferendo produrre più generazioni possibili. Le modalità di attuazione per questa idea sono multiple e variano nella loro intrusività con la normale prosecuzione dell'algoritmo. Ad esempio, una volta calcolati un certo numero di figli, si potrebbe iniziare a tenere traccia dei loro tempi di generazione medi e limitare i successivi di conseguenza.

La tecnica proposta è leggermente diversa da quest'ultima, meno limitante e sfrutta la nozione fondamentale che la creazione dei nuovi figli, in ogni generazione esclusa la prima, avviene in multi threading: il calcolatore mette a disposizione un numero di thread pari ai propri processori virtuali, i quali generano in modo autonomo un nuovo *figlio*. Questo richiede di attivare multiple istanze di CPLEX contemporaneamente, tra loro scorrelate e che utilizzino un solo thread.

Fatta questa premessa, indicando con  $m$  il numero di *figli* da creare ad ogni generazione e con  $z$  il numero di thread a disposizione, la nostra tecnica prevede di tenere traccia di quanto tempo impiega l'algoritmo per ottenere  $m - z$  figli. Tale intervallo è pesato proprio su  $m - z$  ed in base al valore finale ottenuto si attuano limitazioni per i restanti  $z$  figli.

$$t_{mzP}[s] = t_{mz} / (m - z) \quad (14)$$

Entrando più nel dettaglio si limita il tempo per completare i restanti figli, per costruzione ora calcolabili tutti in parallelo, a :

- **10 secondi:** se  $t_{mzP}$  è inferiore a tale valore;
- $t_{mzP}$ : se questo è compreso tra 10 e 300 secondi;

---

<sup>79</sup>Verrà indicata come **saturazione di secondo** tipo nei commenti al codice.

- **300 secondi:** se  $t_{mzP}$  è superiore a tale valore;

La descrizione di come questa seconda tecnica anti saturazione è stata implementata viene rimandata alla appendice in quanto risulta particolarmente complessa ma di relativo interesse. Prima di passare alla sezione successiva dove viene discusso come attivare thread multipli ed assegnargli compiti specifici facciamo solamente notare che per come è stato costruito,  $t_{mzP}$  è pari, ma in genere **superiore** al tempo medio necessario per calcolare  $z$  figli parallelamente.

## UTILIZZARE MULTIPLE ISTANZE DI CPLEX IN PARALLELO

L'utilizzo di multiple istanze di cplex, tra loro indipendenti ed attive contemporaneamente su appositi threads viene introdotto solamente ora in quanto tutti gli algoritmi discussi precedentemente al **polishing** sono stati presentati nella loro versione più base dove era l'unica istanza di CPLEX a lavorare in più threads. In realtà i test che verranno presentati in questa tesi fanno tutti uso di una versione modificata delle tecniche proposte che appunto sfrutta più istanze di CPLEX attive contemporaneamente.

Il motivo di tale scelta è che permette di ottenere multipli output ma con prestazioni chiaramente limitate. Teniamo sempre ben presente che lo scopo delle varie prove proposte è il confronto delle tecniche studiate e non una loro analisi in termini di prestazioni assolute in quanto queste sono molto dipendenti dall'hardware utilizzato.

Questo ragionamento non riguarda gli algoritmi *genetico* e *polishing* dato che si è scelto di elaborare in parallelo solamente i *figli* di ogni generazione e quindi l'output prodotto è comunque unico.

Per quanto riguarda il *genetico* il motivo è molto semplice: il calcolo delle soluzioni *figlie* a partire da due soluzioni *genitori* è molto semplice e non impiega grandi risorse hardware. È quindi motivato un approccio multi-threading e dato che la bontà di tali algoritmi dipende molto dal numero di generazioni che si creano è preferibile velocizzare tale processo piuttosto che attivare più algoritmi genetici contemporaneamente.

D'altro canto l'algoritmo *polishing*, tenendo comunque valida la motivazione appena esposta per il *genetico*, si deve considerare che si utilizza CPLEX per generare un nuovo *figlio* e che tale solver non garantisce risultati migliori se gli si concede più threads<sup>80</sup>.

Vediamo quindi la struttura generale per attivare multipli threads in linguaggio C#. È possibile trovare una applicazione pratica nell'esposizione del codice relativo all'algoritmo *polishing* nella sezione successiva, mentre maggiori dettagli per tutti gli altri metodi risolutivi verranno forniti assieme ai test eseguiti.

Esistono molteplici metodi per ottenere il risultato desiderato, quello che esponiamo prevede l'uso della classe **ThreadPool**<sup>81</sup> la quale fornisce il metodo statico fondamentale **QueueUserWorkItem** che delega al sistema la creazione di un nuovo thread il quale esegue una qualsivoglia porzione di codice con a disposizione tutte le variabili del thread chiamante:

---

```
ThreadPool.QueueUserWorkItem(o =>
{
    \\Codice generico
});
```

---

<sup>80</sup> Visitare contemporaneamente più nodi dell'albero decisionale durante la fase di **Branch&Cut** non garantisce di terminare la risoluzione più velocemente rispetto all'analisi di un solo nodo ma a pieno potenziale hardware.

<sup>81</sup> Appartenente allo spazio dei nomi **System.Threading**.

Due semplici considerazioni: per prima cosa si nota che non è necessario istanziare un oggetto della classe **ThreadPool** ed in secondo luogo il nuovo thread viene gestito in background dal sistema ed attivato il prima possibile. Questo significa che il guadagno ottenuto nell'evitare qualsivoglia tipo di attesa, comporta la prosecuzione immediata del thread originale e soprattutto la mancanza di metodo diretto per conoscere quando le operazioni eseguite background terminano.

Per i nostri scopi queste mancanze devono essere obbligatoriamente risolte per entrambi gli algoritmi **genetico** e **polishing** che necessitano di sincronizzazione per il calcolo delle varie generazioni.

Introduciamo quindi le due classi **ManualResetEvent** e **WaitHandle** che una volta combinate rispondono esattamente alle nostre esigenze: un oggetto di tipo *ManualResetEvent* una volta istanziato può essere *gestito* dalla classe *WaitHandle* la quale dispone del metodo statico **WaitAll**. Fintanto che tutti gli oggetti da lei gestiti non chiamano il metodo **Set()** il thread su cui viene eseguita la chiamata *WaitAll(...)* è posto in attesa.

La generazione di multipli thread deve quindi essere gestita nel seguente modo:

---

```
var resetEventList = new List<ManualResetEvent>();

for (int i = 0; i < contatore; i++)
{
    resetEventList.Add(new ManualResetEvent(false));

    ManualResetEvent mRE = resetEventList[i];

    ThreadPool.QueueUserWorkItem(o =>
    {
        Generico codice
        mRE.Set();
    });
}

WaitHandle.WaitAll(resetEventList.ToArray());
```

---

Concludiamo infine con un commento tecnico al codice appena presentato. Come si può notare nel caso in cui si chiami **ThreadPool.QueueUserWorkItem** all'interno di un ciclo e, nello specifico, se il codice al suo interno dipende da quale iterazione genera la chiamata, è necessario creare un nuovo riferimento per tutte le variabili dipendenti da quanto appena detto. Come accennato in precedenza, l'operazione *QueueUserWorkItem* programma la creazione di un nuovo thread ed i riferimenti alle variabili utilizzate sono assegnati unicamente al suo effettivo avvio. Se quest'ultima operazione viene quindi posticipata durante successive iterazione del ciclo si incorre inevitabilmente in errori multipli.

Apparentemente la soluzione proposta non dovrebbe risolvere tale problema: si tratta però di un *pattern* automaticamente riconosciuto dal compilatore presente nelle ultime versioni del framework C#, introdotto appositamente per tale scopo.

## IMPLEMENTAZIONE POLISHING

In questa sezione vengono presentate le parti più importanti del codice di programmazione che realizza il metodo *matheuristic polishing* discusso. Data la natura di quest'ultimo, la struttura generica del codice è molto simile a quella del metodo genetico LINK!!!!!!!, pertanto non sono discussi nel dettaglio gli aspetti comuni tra i due algoritmi.

Iniziamo con la classica firma del metodo:

---

```
static void Polishing(Instance instance, Random rnd, Process process, int
    sizePopulation, Stopwatch clock)
```

---

Dove:

- **instance**: riferimento all'oggetto contenente tutte le informazioni relative all'istanza del Problema del Commesso Viaggiatore corrente;
- **rnd**: istanza della classe Random utilizzata per la generazione dei numeri casuali, un eventuale seme random deve essere già stato assegnatogli;
- **process**: istanza della classe Process, utilizzata per la stampa su file e a video tramite GNUPlot delle soluzioni trovate, deve essere già stata inizializzata dal metodo chiamante;
- **sizePopulation**: Dimensione della popolazione di ogni generazione, in altre parole il numero di soluzioni facenti parte di queste ultime;
- **clock**: Oggetto che funge da cronometro;

Come prima operazione, trattandosi di un metodo matheuristico è necessario preparare i riferimenti a CPLEX e quindi costruire il modello di partenza ovviamente privo di alcun taglio. Ricordiamo inoltre che il codice presentato utilizza il multithreading per genere più *figli* contemporaneamente e pertanto sono necessarie multiple istanze di CPLEX e relativi modelli matematici:

---

```
//Lista contenente i riferimenti alle istanze degli oggetti di tipo CPLEX
List<CPLEX> CPLEXList = new List<CPLEX>();
```

```
//Lista contenente i riferimenti alle variabili utilizzate dagli oggetti
    di tipo CPLEX nel proprio modello matematico
List<INumVar[]> xList = new List<INumVar[]>();
```

```
//Data una popolazione di n elementi si creano n/2 figli
for (int i = 0; i < sizePopulation / 2; i++)
{
    //Inizializzazione degli oggetti di tipo CPLEX
    CPLEXList.Add(new CPLEX());
    //Settaggio di un seme random basato sui ticks di sistema
    rappresentanti l'istante attuale
    CPLEXList[i].SetParam(CPLEX.Param.RandomSeed,
        (int)(DateTime.Now.Ticks & 0x0000FFFF));

    //Per ogni oggetto CPLEX inizializzato se ne definisce il modello
    matematico, viene inoltre aggiunto
    //il riferimento delle variabili utilizzate alla apposita lista in
    modo tale che il generico j-esimo
    //oggetto CPLEX in CPLEXList abbia le proprie nella j-esima posizione
    di xList
    xList.Add(Utility.BuildModel(CPLEXList[i], instance, -1));

    //Installazione lazyconstraint callback
```

```

CPLEXList[i].Use(new TSPLazyConsCallback(CPLEXList[i], xList[i],
    instance, false));

//Inizializzazione di un aborter per ogni oggetti CPLEX, utilizzato
    per gestire la saturazione di
//secondo tipo
CPLEXList[i].Use(new CPLEX.Aborter());
}

```

Devono essere inizializzati anche gli oggetti che andranno a contenere i *percorsi* delle varie soluzioni trovate. Viene utilizzata a tale scopo la classe **PathGenetic** LINK!!!!!! dove in questo caso il percorso è memorizzato sempre all'interno della variabile **path** ma rispettando la struttura delle variabili di CPLEX: molto semplicemente *path[i]* assume valore 1 se e solo se la variabile di indice *i* – *esimo* della soluzione a cui si fa riferimento vale anch'essa 1, in caso contrario deve valere 0.

```

//Definisco l'oggetto che codifica la soluzione incumbent (miglior
    soluzione in assoluto fra tutte le generazioni)
PathGenetic incumbentSol = new PathGenetic();

//Miglior soluzione all'interno della nuova generazione calcolata ad ogni
    iterazione
PathGenetic currentBestPath = null;

//Popolazione della generazione attuale
List<PathGenetic> CurrentPopulation = new List<PathGenetic>();

//Figli della generazione attuale
List<PathGenetic> ChildPoulation = new List<PathGenetic>();

```

La prima generazione, come anche l'anti saturazione di primo tipo, sfrutta una nuova versione del metodo **Nearest Neighbour** visto LINK!!!!!! dove sono introdotte tre variazioni fondamentali:

- Il circuito deve essere memorizzato all'interno degli oggetti *PathGenetic* seguendo la struttura descritta in precedenza;
- Il nodo iniziale per la costruzione del circuito è sempre quello di indice 0 così che l'algoritmo *RINS* abbia più possibilità di fissare anche variabili comuni di valore 1;
- La probabilità di utilizzare il lato più corto disponibile oppure il successivo, dipende dal numero di nodi della istanza analizzata e da quante saturazioni di primo tipo successive avvengono; In particolare si ha che la probabilità di selezione il lato migliore cresce al crescere dei nodi della istanza in quanto creare soluzioni di partenza più dissimili tra loro aumenta ulteriormente i tempi di generazione dei *figli*. Infine questa probabilità è abbassata del 5% ad ogni saturazione di primo tipo consecutiva (*malus* resettato al primo miglioramento trovato) così da variare progressivamente le soluzioni introdotte;

I dettagli implementativi riguardanti il metodo **NearestNeighborPolishing** sono presenti LINK!!!!!! all'interno della appendice.

```

//listArray[i][y] = z ; implica che il nodo di indice z è l'y-esimo più
    vicino a quello di indice i

```

```

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

for (int i = 0; i < sizePopulation; i++)
{
    //Generazione di una soluzione e conseguente aggiunta del suo
    //riferimento nella apposita lista
    CurrentPopulation.Add(Utility.NearestNeighborPolishing(instance,
        rnd, listArray, 0));

    //Aggiornamento valore incumbent
    if (currentBestPath == null || currentBestPath.Cost >
        CurrentPopulation[i].Cost)
        currentBestPath = CurrentPopulation[i];
}

```

---

Per poter cominciare la generazione ciclica delle varie generazioni sono necessarie inizializzazioni di variabili di *servizio*, il settaggio del numero di **Threads** massimosi utilizzabile dalla classe *ThreadPool* e lo start del cronometro. I dettagli riguardanti l'anti saturazione di primo tipo sono riportati nella apposita sezione LINK!!!! della appendice.

---

```

//ThreadPool può programmare un numero massimo di threads pari ai core
//virtuali della macchina per la //generazione di nuovi figli, ed un
//thread aggiuntivo di servizio utilizzato per evitare la saturazione
//di secondi tipo e cioè evitare che alcuni figli impieghino troppo tempo
//a generarsi rispetto agli
//altri della stessa generazione. Questo thread rimane per la maggior
//parte del tempo in attesa
ThreadPool.SetMaxThreads(CPLEXList[0].GetNumCores() + 1,
    CPLEXList[0].GetNumCores() + 1);

//Questa variabile memorizza il tempo impiegato per generare i figli
//della attuale generazione
double timeC;

//Contatore di generazioni successive che non migliorano l'incumbent e
//generate in tempi validi per
//contribuire alla saturazione di primo tipo
int cntSat = 0;

//Contatore di saturazioni di primo tipo consecutive
int cntSatProgr = 0;

//Oggetti necessari per l'anti saturazione di secondo tipo, utilizzata
//per impedire che alcuni figli
//impieghino troppo tempo per la loro creazione rispetto ad altri della
//stessa generazione
object lockPol;
EventWaitHandle ewhCnt;
EventWaitHandle ewhCntComplete;

//Eventuale start del cronometro
if (!clock.IsRunning)

```

```
clock.Start();
```

La creazione delle varie generazioni di soluzioni per il **TSP** corrente avviene all'interno di un ciclo **do-while** di durata temporale pari al tempo limite indicato da parte dell'utente. Di seguito è riportato il codice commentato che gestisce la creazione della nuova generazione. Il metodo **GenerateChildPolish** è analizzato a fine sezione.

```
//Inizializzazione variabili che gestiscono la saturazione di secondo tipo
lockPol = new object();
EventWaitHandle ewhCnt = new EventWaitHandle(false,
    EventResetMode.AutoReset);
EventWaitHandle ewhCntComplete = new EventWaitHandle(false,
    EventResetMode.AutoReset);

//Aggiornamento dei time limit di CPLEX in modo tale che venga sempre
    lasciato almeno un secondo e mezzo
//per la generazione di ogni figlio
for (int i = 0; i < sizePopulation / 2; i++)
{
    if (instance.TimeLimit - (clock.ElapsedMilliseconds / 1000.0) > 1.5)
    {
        CPLEXList[i].SetParam(CPLEX.Param.TimeLimit, instance.TimeLimit -
            (clock.ElapsedMilliseconds / 1000.0));

        CPLEXList[i].SetParam(CPLEX.Param.DetTimeLimit,
            Program.TICKS_PER_SECOND *
            CPLEXList[i].GetParam(CPLEX.Param.TimeLimit));
    }
    else
    {
        CPLEXList[i].SetParam(CPLEX.Param.TimeLimit, 1.5);

        CPLEXList[i].SetParam(CPLEX.Param.DetTimeLimit,
            Program.TICKS_PER_SECOND *
            CPLEXList[i].GetParam(CPLEX.Param.TimeLimit));
    }
}

//resetEventC contiene la lista di ManualResetEvent utilizzati per
    segnalare quando un figlio è stato
//creato
List<ManualResetEvent> resetEventC = new List<ManualResetEvent>();

//resetEventSat è utilizzato per segnalare quando il thread di supporto
    per gestire la durata dei figli
//termina
ManualResetEvent resetEventSat = new ManualResetEvent(false);

//resetEventList viene popolata di tutti i riferimenti di resetEventC e
    resetEventSat ed questo oggetto ad
//essere gestito dalla classe WaitHandle
List<ManualResetEvent> resetEventList = new List<ManualResetEvent>();
```

```

//index è una variabile di supporto utilizzata per accoppiare
    correttamente i genitori dei figli
int[] index = new int[sizePopulation / 2];
//Popolamento di index e di resetEventList, un oggetto ManualResetEvent
    per ogni figlio
for (int i = 0; i < sizePopulation/2; i++)
{
    index[i] = i * 2 + 1;
    resetEventList.Add(new ManualResetEvent(false));
}

//Richiesta di attivavazione del thread di servizio che monitora e
    gestisce il tempo con cui vengono
//generati i figli, è compito del metodo MonitorCPLEX gestire quindi la
    saturazione di secondo tipo
ThreadPool.QueueUserWorkItem(o =>
{
    MonitorCPLEX(CPLEXList[0].GetNumCores(), CPLEXList, sizePopulation,
        ewhCnt, ewhCntComplete);
    resetEventSat.Set();
});

//resetEventSat è aggiunto alla lista resetEvent
resetEvent.Add(resetEventSat);

//Prima di proseguire è necessario che il thread di servizio venga creato
    e MonitorCPLEX abbia completato
//l'inizializzazione locale. Viene conferito un tempo di attesa limite di
    50 ms
ewhCntComplete.WaitOne(50);

//startG tiene traccia dell'istante iniziale relativo alla creazione
    della nuova generazione
double startG = clock.ElapsedMilliseconds / 1000.0;

//ThreadPool setta la coda di thread per la creazione in parallelo di
    tutti i figli, le variabili
//ManualResetEvent utilizzate per segnalare il completamente di questi
    ultimi sono aggiunte a resetEvent
foreach(int i in index)
{
    int y = i;
    ManualResetEvent mRE = resetEventList[(y - 1) / 2];

    ThreadPool.QueueUserWorkItem(o =>
    {
        ChildPoulation.Add(Utility.GenerateChildPolish(CPLEXList[(y - 1)
            / 2], (Instance)instance.Clone(), xList[(y - 1) / 2],

```



```

        CurrentPopulation[y - 1], CurrentPopulation[y], lockPol,
        ewhCnt, ewhCntComplete));
    mRE.Set();
});

    resetEvent.Add(mRE);
}

//Fino a che tutti i thread richiesti da ThreadPool, e quindi la
//generazione di tutti i figli ed il thread
//di servizio, terminano non si prosegue
WaitHandle.WaitAll(resetEvent.ToArray());

//Reset degli oggetti CPLEX.Aborter per ogni oggetto CPLEX
for (int i = 0; i < sizePopulation / 2; i++)
{
    CPLEXList[i].Use(new CPLEX.Aborter());
}

//La effettiva nuova generazione viene creata sfruttando la precedente ed
//i suoi figli nello stesso modo
//del metodo genetico
CurrentPopulation = Utility.NextPopulation(instance, sizePopulation,
    CurrentPopulation, ChildPopulation);

//Tempo totale misurato in secondi impiegato per la creazione della nuova
//generazione
timeC = clock.ElapsedMilliseconds / 1000.0 - startG;

//Aggiorno il riferimento alla soluzione migliore della nuova generazione
currentBestPath = Utility.BestSolution(CurrentPopulation);

```

---

La corrente iterazione del ciclo *do-while* termina con la verifica del miglioramento dell'incumbent, in tal caso si procede anche alla stampa della nuova soluzione ed al reset dei contatori per la saturazione di primo tipo.

---

```

//Aggiornamento della soluzione incumbent
incumbentSol = (PathGenetic)currentBestPath.Clone();

//Stampa della nuova miglior soluzione
Utility.PrintGeneticSolution(instance, process, incumbentSol);

//Reset contatori saturazione di primo tipo
cntSat = 0;
cntSatProgr = 0;

```

---

Nel caso in cui non ci sia un miglioramento inizia il processo di anti saturazione di primo tipo.

---

```

//Verifica se il tempo impiegato rientra nei parametri di saturazione
if (timeC < 0.085 * (instance.NNodes / 10) * sizePopulation)
    cntSat++;
else

```

```

        cntSat = 0;

//Nel caso in cui 10 generazioni consecutive non migliorative rientrano
    nei parametri di saturazione
//avvio il metodo anti saturazione di primo tipo
if (cntSat == 10)
{
    //Calcolo la diminuzione della probabilità che il metodo
        NearestNeighbor tenti di selezionare ad ogni
    //iterazione il lato meno costoso
    double minus = 0.05 * cntSatProgr;

    //Rimozione di un quarto delle soluzioni della corrente generazione,
        più precisamente sono sempre
    //rimosse quelle di indice 0, 4, 8, ecc...
    for (int i = 0; i < sizePopulation / 4; i++)
    {
        CurrentPopulation.RemoveAt(sizePopulation - (4 * (i + 1)));
    }

    //Rimpiazzo le soluzioni rimosse con delle nuove generate dal Nearest
        Neighbour
    for (int i = 0; i < sizePopulation / 4; i++)
    {
        CurrentPopulation.Insert(4 * i,
            Utility.NearestNeighborPolishing(instance, rnd, listArray,
                minus));

        //Nel caso improbabile che una delle nuove soluzioni migliori
            l'incumbent aggiornino tale valore
        if (CurrentPopulation[4 * i].Cost < currentBestPath.cost)
        {
            currentBestPath = CurrentPopulation[CurrentPopulation.Count -
                1];
        }
    }

    //Aggiorno i contatori di saturazione di primo tipo
    cntSatProgr++;
    cntSat = 0;
}

```

---

Si conclude questa sezione con la dovuta analisi del metodo **GenerateChildPolish** che utilizza l'algoritmo *RINS* per creare un *figlio* a partire da due genitori.

---

```

public static PathGenetic GenerateChildPolish(CPLEX cplex, Instance
    instance, INumVar[] x, PathGenetic mother, PathGenetic father, object
    lockPol, EventWaitHandle ewhCnt, EventWaitHandle ewhCntComplete)

```

---

- **cplex**: oggetto della classe CPLEX utilizzato per risolvere il modello;
- **instance**: riferimento alla classe Instance, accesso solo in lettura;

- **x**: vettore contenente i riferimenti alla variabili del modello matematico di cplex;
- **mother**: primo dei due circuiti genitori;
- **father**: secondo dei due circuiti genitori;
- **lockPol**, **ewhCnt**, **ewhCntComplete**: oggetti necessari per segnalare in modo il completamento della generazione della soluzione *figlio*, maggiori dettagli LINK!!!!!!;

Il contenuto di questa funzione è molto semplice, si procede inizialmente al fissaggio secondo la tecnica *RINS* delle variabili a valore comune nei due circuiti *genitori* ed inoltre il migliore tra questi viene settato come **warm-start**<sup>82</sup>:

---

```
//Il percorso della soluzione figlio viene inizializzato
int[] path = new int[mother.Path.Length];

//Fissaggio delle variabili comuni tra i due genitori a valore 1
for (int i = 0; i < mother.path.Length; i++)
{
    if (mother.Path[i] == 1 && father.Path[i] == 1)
        x[i].LB = 1;
}

//Fissaggio delle variabili comuni tra i due genitori a valore 0
for (int i = 0; i < mother.path.Length; i++)
{
    if (mother.Path[i] == 0 && father.Path[i] == 0)
        x[i].UB = 0;
}

//cplex.AddMIPStart accetta unicamente un vettore di double come
//warm-start mentre all'interno
//degli oggetti di tipo PathGenetic è utilizzato un vettore di interi
if (mother.Cost > father.Cost)
    cplex.AddMIPStart(x, ConvertIntArrayToDoubleArray(father.Path));
else
    cplex.AddMIPStart(x, ConvertIntArrayToDoubleArray(mother.Path));

//Il modello matematico definito viene risolto da CPLEX
cplex.Solve();
```

---

Terminata la risoluzione del modello matematico, operazione che può anche essere stata forzata con un abort nel caso in cui si ricada nella saturazione di secondo tipo, è necessario risvegliare il thread di servizio in modo tale da comunicargli il completamento della generazione di un figlio.

Tale thread può gestire una sola comunicazione per volta, è necessario creare quindi un sistema di serializzazione: i threads che gestiscono la creazione di un figlio devono essere certi che il thread di servizio non stia gestendo altre segnalazioni di completamento.

---

<sup>82</sup>La definizione di un warm-start è necessaria in quanto può capitare che CPLEX non generi una soluzione valida entro il tempo limite fissato causando successivamente una eccezione.

---

```

//Se la terminazione non è avvenuta in seguito ad un abort dovuto a
    saturazione di secondo tipo
//si comunica al thread di servizio il completamento della generazione di
    un figlio
if (!cplex.GetAborter().IsAborted())
{
    //lockPol viene utilizzato per serializzare delle comunicazioni di
        completamento
    //se non è disponibile si attende la chiamata Monitor.Pulse(lockPol)
        da parte di un altro thread
    lock (lockPol)
    {
        //Viene comunicata il completamento del figlio ed atomicamente si
            attende una segnalazione
        //da parte del thread di servizio: solo dopo tale segnalazione è
            possibile liberare il lock su
        //lockPol
        WaitHandle.SignalAndWait(ewhCnt, ewhCntComplete, 1000, false);
        //Monitor.Pulse(lockPol) risveglia un qualsiasi thread in attesa
            su lockPol
        Monitor.Pulse(lockPol);
    }
}

```

---

Completata la fase di sincronizzazione non rimane altro che leggere la soluzione trovata da CPLEX, popolare un apposito oggetto **PathGenetic** contenente il nuovo figlio e restituirlo, non prima di aver eliminato il fissaggio delle variabili secondo *RINS*.

---

```

//Si ottiene il valore delle variabili nella soluzione figlio trovata
double[] pathChild = cplex.GetValues(x);

//Popolamento del vettore path
for (int i = 0; i < mother.path.Length; i++)
{
    if (pathChild[i] >= 0.5)
        path[i] = 1;
    else
        path[i] = 0;
}

//Viene creato l'oggetto di tipo PathGenetic per il figlio generato
child = new PathGenetic(path, cplex.GetObjValue());

//Reset delle variabili modificate secondo la tecnica RINS
for (int i = 0; i < mother.path.Length; i++)
{
    x[i].LB = 0;
    x[i].UB = 1;
}

```

```
return child;
```

---

## APPENDICE

### CLASEE STOPWATCH

### GNUPLOT

### METODO INITPROCESS

### METODO MODIFYMODEL

Il metodo ModifyModel fissa in soluzione, con una certa percentuale p, lati che appartengono ad una soluzione ammissibile. La sua firma risulta essere:

---

```
public static void ModifyModel(Instance instance, INumVar[] x, Random
    rnd, double percentageFixing, double[] solution)
```

---

Dove:

- **instance**: riferimento all' oggetto contenente tutte le informazioni relative all' istanza del Problema del Commesso Viaggiatore corrente;
- **x**: vettore contenente le variabili del modello;
- **rnd**: istanza della classe Random utilizzata per la generazione dei numeri casuali;
- **percentageFixing**: Probabilità con cui un lato viene fissato in soluzione;
- **solution**: Vettore che codifica la soluzione ammissibile su cui si fissano in soluzione i lati;

L' idea dell' algoritmo consiste nel scandire tutti i lati appartenenti alla soluzione ammissibile fornitagli in ingresso, invocare per ognuno lato il metodo **RandomSelect** e se ritorna 1 viene fissato il lato altrimenti no. Per fissarlo si pone semplicemente il LB della variabile ad esso associata a 1. Il ciclo do - while serve per ripetere il fissaggio qualora si fissino tutti i lati o non si vincoli solo un lato poichè non avrebbe senso a questo punto far partire cplex. Al termine del metodo viene effettuato il preprocessing discusso del paragrafo precedente.

---

```
//Stored the number of variable fixed
int nVariabileFix = 0;

do
{
    nVariabileFix = 0;

    //Scan all edge that belong to the current heuristic solution
    for (int i = 0; i < x.Length; i++)
    {
        if ((solution[i] == 1))
        {
```

```

//Whit a percentageFixing probability fix a edge belong to the current
    solution
if (RandomSelect(rnd, percentageFixing) == 1)
{
x[i].LB = 1;
nVariabileFix++;
}
}
}

} while (nVariabileFix >= instance.NNodes - 1);

Utility.PreProcessingTSP(instance, x);

```

---

## FILL ROULETTE

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

```

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

```

---

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

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

```

int sizeRoulette = 0;

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

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

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

```

```
}  
return sizeRoulette;
```

---

## ESTIMATE

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

---

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

---

Dove:

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

## REPAIR

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

Discutiamo ora il codice prodotto. Costruiamo due liste di interi chiamate **isolatedNodes** e **nearestIsolatedNodes** dove rispettivamente contengono all'i-esimo elemento l'indice dell'i-esimo nodo isolato e l'indice del nodo ad esso più vicino<sup>83</sup>. Per popolare tali liste utilizziamo i metodi **FindIsolatedNodes** LINK!!!!!! e **FindNearestNode** LINK!!!!!!!!!!!!. Dichiariamo le due liste **pathIncomplete** e **pathComplete**, nella prima andiamo ad inserire il percorso originale privo degli elementi che lo rendono non hamiltoniano mentre nella seconda costruiamo il percorso completo di tutti i nodi ed hamiltoniano. Per ottenere quest'ultimo risultato procediamo ciclicamente con la copia di ogni elemento appartenente a *pathIncomplete* in **pathComplete** facendo però in modo che, se per un qualsiasi  $i, j, m$  vale  $pathIncomplete[i] = nearestIsolatedNodes[j]$ , allora nel caso andiamo a porre  $pathComplete[m] = pathIncomplete[i]$  dovrà anche valere  $pathComplete[m+1] = isolatedNodes[j]$  e  $pathComplete[m+2] = pathIncomplete[i+1]$  simili a quanto avviene nella Figura 3!!!!!!!!!!!!!!!!!!!!!!.

---

```
int positionInsertNode = 0;
```

---

<sup>83</sup>Ci sono in realtà delle eccezioni a quest'ultima affermazione che vedremo in seguito

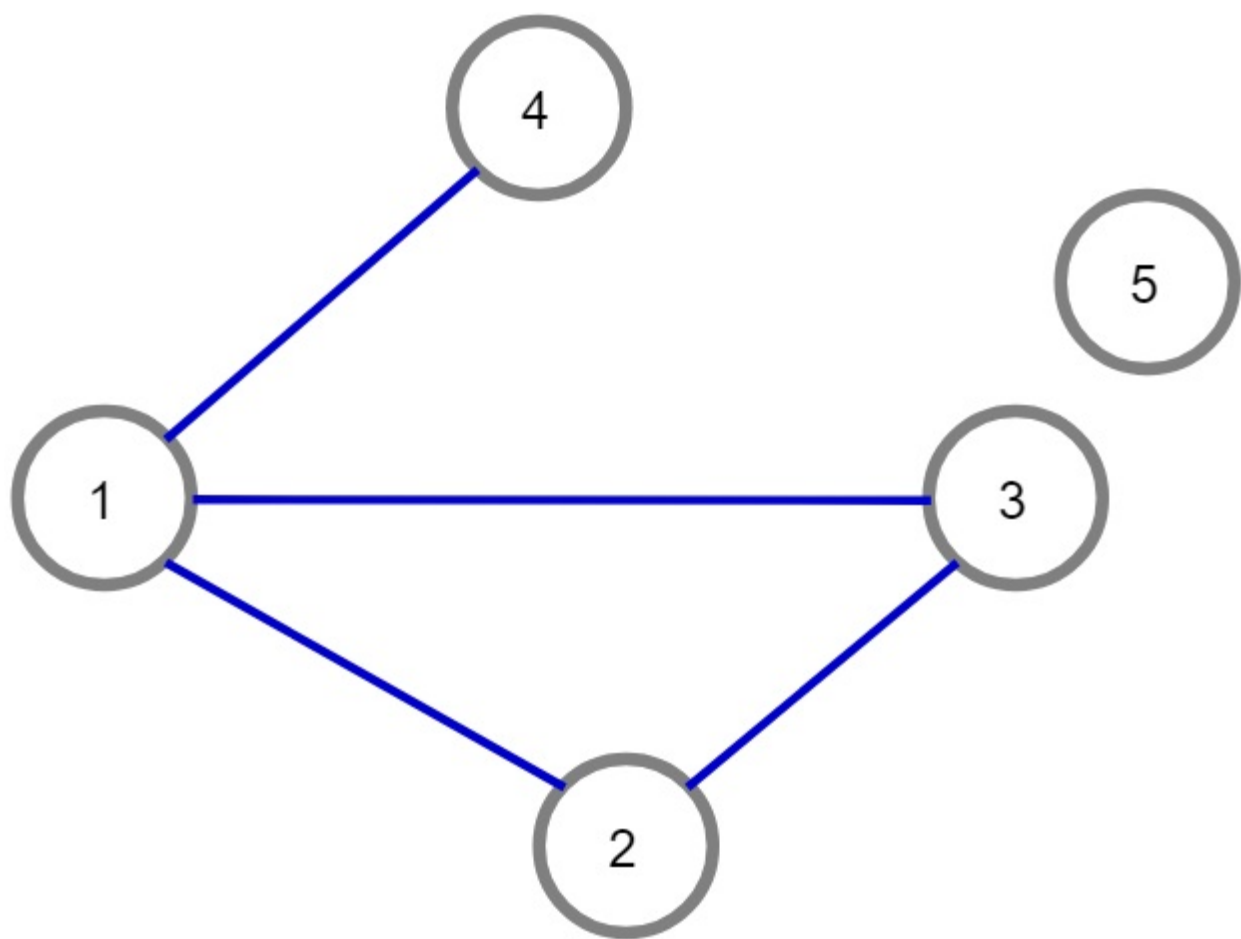


Fig. 13: Circuito non hamiltoniano figlio



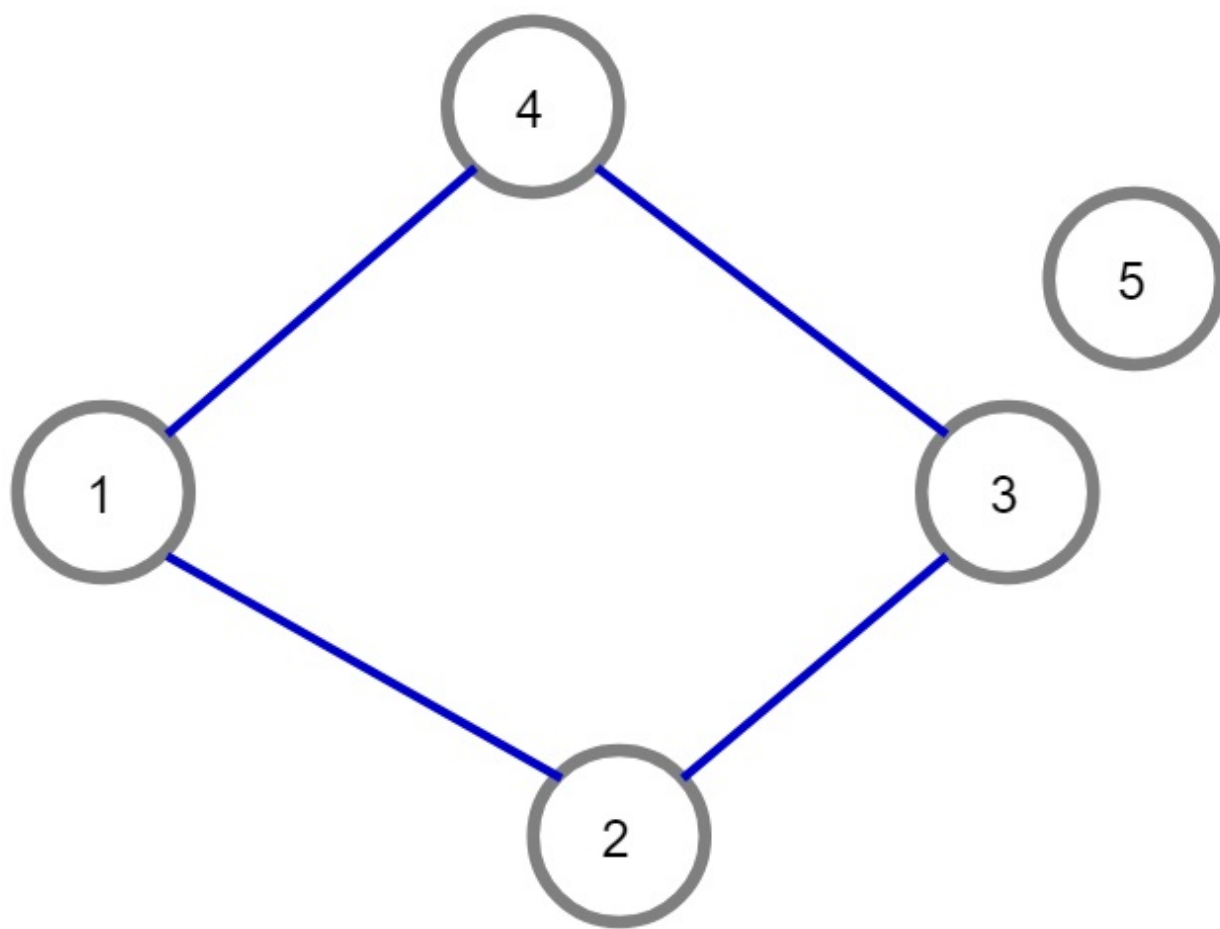


Fig. 14: Circuito hamiltoniano incompleto

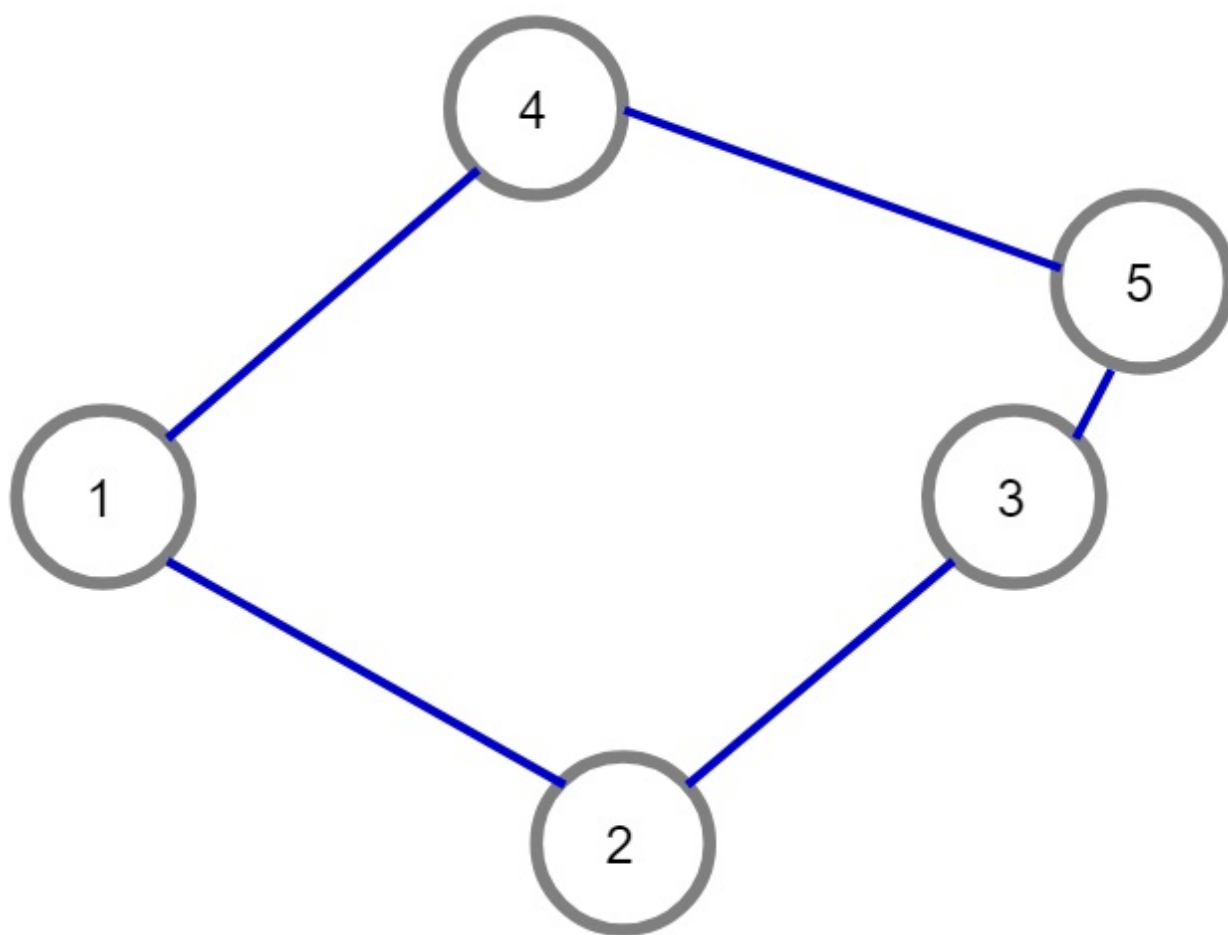


Fig. 15: Circuito hamiltoniano figlio

```

for (int i = 0; i < pathIncomplete.Count; i++)
{
    if (nearestIsolatedNodes.Contains(pathIncomplete[i]))
    {
        pathComplete[positionInsertNode] = pathIncomplete[i];
        pathComplete[positionInsertNode + 1] =
            isolatedNodes[nearestIsolatedNodes.IndexOf(pathIncomplete[i])];
        positionInsertNode++;
    }
    else
        pathComplete[positionInsertNode] = pathIncomplete[i];

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

```

---

## FINDISOLATEDNODES

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

```

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

```

---

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

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

---

```

bool nodeIsVisited = false;

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

    //If the node has never been visited is a isolated noode
    if (nodeIsVisited == false)
        isolatedNodes.Add(i);
}

```

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

---

## FINDNEARESTNODE

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

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

Il codice commentato della funzione viene riportato di seguito.

COMMENTA CODICE!!!!!!!!!!!!!!!!!!!!!!!!!!!! RIPORTA TUTTO IL METODO!!!!!!!!!!!!!!!!!!!!!! PUBLIC ...

---

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

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

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

---

## BESTSOLUTION

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

---

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

---

Dove:

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

Di seguito il codice:

---

```
PathGenetic currentBestPath = population[0];

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

return currentBestPath;
```

---

## NEARESTNEIGHBORPOLISHING & RNDPLUSPOLISHING

Il metodo **NearestNeighborPolishing** applica una versione dell'algoritmo **NearestNeighbor** LINK!!!! compatibile con il metodo matheuristico **Polishing**. Viene memorizzato il circuito in modo compatibile alle variabili di CPLEX<sup>84</sup>, utilizza come nodo di partenza sempre e solo quello di indice 0 ed utilizza una propria versione del metodo **RndPlus**<sup>85</sup> chiamata **RndPlusPolishing**.

---

```
public static PathGenetic NearestNeighborPolishing(Instance instance,
    Random rnd, List<int>[] listArray, double minus)
```

---

Dove:

- **instance**: riferimento alla classe `Instance` contenente le informazioni riguardanti l'istanza presa in analisi;
- **rnd**: variabile utilizzata per ottenere valori random, precedentemente inizializzata;
- **listArray**: `listArray[i][j] = z`; implica che il nodo di indice  $z$  è il  $j$ -esimo più vicino a quello di indice  $i$  nella attuale istanza;
- **minus**: valore necessario per il metodo **RndPlusPolishing**;

---

<sup>84</sup>Se la variabile di indice  $i$  ha valore  $x$  allora il vettore che descrive il circuito sarà  $vect[i] = x$ .

<sup>85</sup>**RndPlus** determina se la selezione del lato migliore possibile oppure uno dei successivi ad ogni iterazione dell'algoritmo **NearestNeighbor**.

Di seguito si riporta il codice commentato

---

```
//Vettore in cui memorizzo il nuovo circuito, (instance.NNodes - 1) *
//instance.NNodes / 2 corrisponde
//al numero totale di lati presenti
int[] heuristicSolutionCPLEX = new int[(instance.NNodes - 1) *
instance.NNodes / 2];

//Costo del circuito prodotto
double cost = 0;

//Memorizza i nodi che delimitano il lato preso in analisi
int[] currentNodes = new int[2];

//Nodi facenti parte del circuito
List<int> notAvailableNodes = new List<int>();

//Il nodo di partenza è sempre quello di indice 0
currentNodes[0] = 0;

//Il nodo di indice 0 è non più disponibile per costruzione
notAvailableNodes.Add(currentNodes[0]);

//I lati da inserire nel circuito sono in numero pari al numero di nodi
//della istanza
for (int i = 1; i < instance.NNodes; i++)
{
    //Se falso significa che l'attuale iterazione del ciclo non ha ancora
    //aggiunto un nuovo lato al circuito
    bool found = false;
    //Dal nodo corrente si tenta di passare al nodo plus-iesimo più vicino
    int plus = RndPlusPolishing(instance, rnd, minus);
    int nextNode = listArray[currentNodes[0]][plus];

    do
    {
        //Si controlla se il nextNode selezionato è ancora disponibile
        if (notAvailableNodes.Contains(nextNode) == false)
        {
            //Il lato da aggiungere ha estremi i nodi di indice
            //currentNodes[0] e currentNodes[1]
            currentNodes[1] = nextNode;
            //Si trova l'indice utilizzato da CPLEX per memorizzare la
            //variabile relativa al nodo
            //selezionato
            int pos = xPos(currentNodes[0], currentNodes[1],
instance.NNodes);

            //Il costo corrente del circuito è aggiornato
            cost += (int)Point.Distance(instance.Coord[currentNodes[0]],
instance.Coord[currentNodes[1]], instance.EdgeType);
        }
    }
}
```

```

        //Il vettore che descrive il circuito è aggiornato
        heuristicSolutionCPLEX[pos] = 1;

        //Aggiornamento dei nodi non disponibili e della nuova
        estremità del circuito
        notAvailableNodes.Add(nextNode);
        currentNodes[0] = nextNode;
        found = true;
    }
    else
    {
        //Nel caso il nodo selezionato non sia disponibile si passa al
        successivo
        //meno distante
        plus++;
        if (plus >= instance.NNodes - 1)
        {
            nextNode = listArray[currentNodes[0]][0];
            plus = 0;
        }
        else
            nextNode = listArray[currentNodes[0]][0 + plus];
    }
} while (!found);
}

//Il circuito viene effettivamente chiuso collegando l'ultimo nodo
selezionato a quello di indice 0
heuristicSolutionCPLEX[xPos(0, currentNodes[1], instance.NNodes)] = 1;
//Il costo viene aggiornato
cost += (int)Point.Distance(instance.Coord[0],
    instance.Coord[currentNodes[1]], instance.EdgeType);
//Una variabile di tipo PathGenetic viene restituita, Path e costo sono
settaggi di conseguenza
return new PathGenetic(heuristicSolutionCPLEX, cost);

```

---

Il metodo **RndPlusPolishing** agisce in base al numero di nodi della istanza considerata, i valori settati non seguono nessuna regola teorica ma sono il risultato di diversi tuning eseguiti durante la fase di test.

---

```

//Valore random tra 0 (compreso) ed 1 (non compreso)
double tmp = rnd.NextDouble();

if (instance.NNodes < 400)
{
    if (tmp < 0.75 - minus)
        //Significa che si tenta di collegare il nodo più vicino a quello
        attuale
        return 0;
    else

```

```

        //Significa che si tenta di collegare il secondo nodo più vicino a
        //quello attuale
        return 1;
    }else if (instance.NNodes < 600)
    {
        if (tmp < 0.8 - minus)
            return 0;
        else
            return 1;
    }
    else if (instance.NNodes < 800)
    {
        if (tmp < 0.9 - minus)
            return 0;
        else
            return 1;
    }
    else
    {
        if (tmp < 0.95 - minus)
            return 0;
        else
            return 1;
    }
}

```

---

## POLISHING - GESTIONE ALGORITMICA SATURAZIONE DI SECONDO TIPO

Ricordando che nell'algoritmo polishing, la saturazione di secondo tipo avviene quando gli ultimi  $m$ <sup>86</sup> figli non ancora determinati richiedono un tempo troppo elevato rispetto ai loro fratelli.

L'idea per gestire questa problematica è molto semplice: una volta terminata la creazione di tutti i figli, eccetto gli ultimi  $m$ , si attende ancora un certo tempo  $T_r$  LINK!!!! dopodiché si lancia una operazione di abort su tutte le istanze di CPLEX. Quelle che hanno già terminato la propria risoluzione non subiscono alcun effetto mentre le altre la terminano in modo sicuro.

Tutto questo viene gestito dal metodo **MonitorCPLEX** eseguito in background su un apposito thread. Le problematiche che incorrono sono principalmente due:

- La comunicazione tra i thread che generano i nuovi figli e *MonitorCPLEX* deve essere serializzata.
- Fare in modo che, se durante la attesa del tempo  $T_r$ , tutti i figli sono completati e quindi **non** è presente saturazione di secondo tipo, *MonitorCPLEX* termini immediatamente così che l'algoritmo di polishing possa proseguire senza inutili ritardi.

Per superare questi due ostacoli è necessario introdurre due classi **WaitHandle** e **EventWaitHandle**<sup>87</sup>: la prima dispone dell'essenziale metodo **SignalAndWait(WaitHandle toSignal,**

<sup>86</sup>Dove  $m$  è il numero di figli calcolati costantemente in modo parallelo da  $m$  threads.

<sup>87</sup>Entrambe le classi *WaitHandle* e *EventWaitHandle* fanno parte dello spazio dei nomi System.Threading.



**WaitHandle toWaitOn**) il quale atomicamente risveglia un thread in attesa su **toSignal** e pone il chiamante esso stesso in atteso su **toWaitOn**.

La soluzione proposta per ottenere una completa sincronizzazione tra tutti i threads in gioco è descritta dalla seguente immagine:

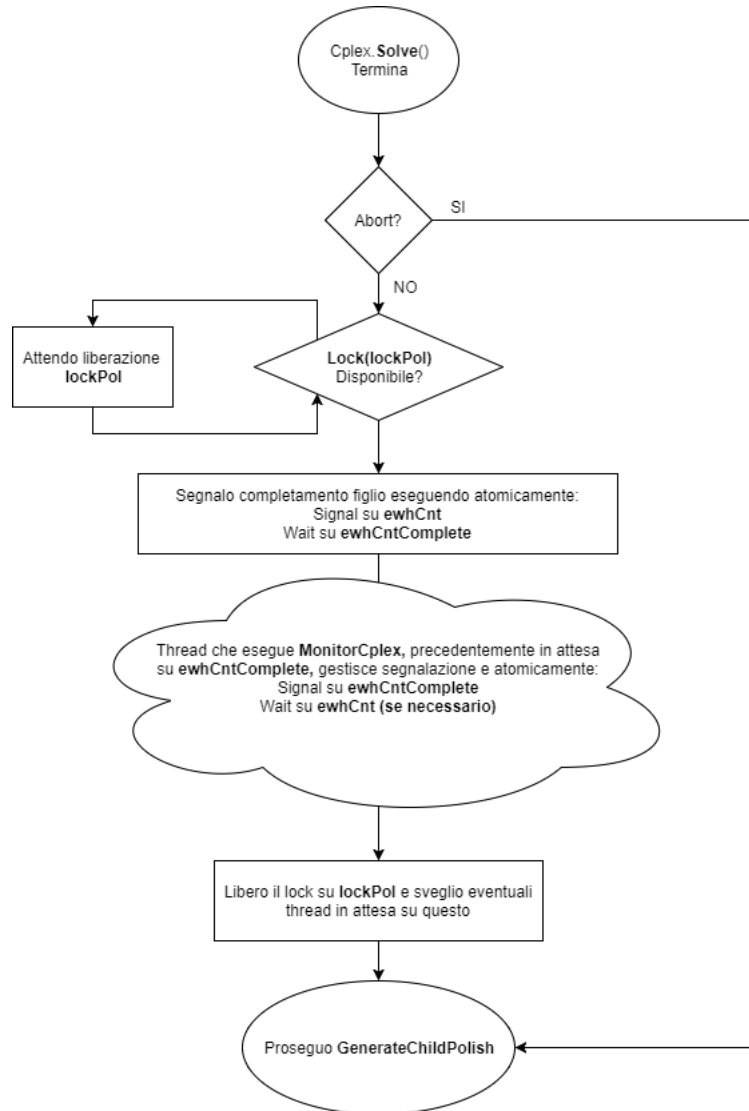


Fig. 16: Diagramma segnalazione serializzata completamento figlio

Banalmente come già visto nel paragrafo LINK!!!!!! il metodo **GenerateChildPolish** una volta terminato il comando **Cplex.Solve()** usa il codice:

```

if (!cplex.GetAborter().IsAborted())
{
    lock (lockPol)
    {
        WaitHandle.SignalAndWait(ewhCnt, ewhCntComplete, 1000, false);
    }
}
  
```

```

        Monitor.Pulse(lockPol);
    }
}

```

Dove l'operazione *WaitHandle.SignalAndWait* ha un timeout di un secondo<sup>88</sup> in quanto può capire che l'operazione di abort sia avviata tra il test della clausola **if** e l'esecuzione di **WaitHandle.SignalAndWait** stesso creando così una attesa infinita.

Completata la creazione del proprio figlio, il thread che esegue **GenerateChildPolish** effettua la segnalazione, attende la sua totale gestione da parte di **MonitorCPLEX** e libera **lockPol**.

Di seguito è riportato infine il codice completo e commentato del metodo *MonitorCPLEX*:

```

public static void MonitorCPLEX(int numThreadsPar, List<CPLEX> cplexList,
    int sizePop, EventWaitHandle ewhCnt, EventWaitHandle ewhCntComplete)
{
    //Cronometro che determina se è necessario un abort o meno di CPLEX
    Stopwatch clock = new Stopwatch();
    clock.Start();
    //Dati sizePop si creano sempre sizePop/2 figli
    int sizeCh = sizePop / 2;

    //Quanti figli è necessario che si completino prima di avviare il
    //controllo anti saturazione
    int maxWait = sizeCh - numThreadsPar;

    //Inizializzazione contatore figli completati
    int cnt = 0;

    //Inizializzazione tempo di attesa anti saturazione
    double tmpSingleThread = 0;

    //Si attendono maxWait segnalazioni di figli completati
    do
    {
        //Il thread corrente attende su ewhCnt e risveglia un thread, se
        //presente,
        //in attesa su ewhCntComplete
        WaitHandle.SignalAndWait(ewhCntComplete, ewhCnt);
        //Una volta risvegliato si procede in modo serializzato
        //all'incremento del contatore cnt
        cnt++;
        //Raggiunta la soglia fissata?
        if (cnt == maxWait)
        {
            //Si salva il tempo impiegato per la risoluzione completa di
            //maxWait figli
            tmpSingleThread = clock.ElapsedMilliseconds / cnt;
            clock.Stop();
        }
    }
}

```

<sup>88</sup>Questo intervallo di tempo è enormemente superiore ad una normale attesa e quindi assicura il corretto funzionamento della sincronizzazione. Essendo la situazione di timeout altrettanto rara non si hanno effetti reali per quanto riguarda un rallentamento dell'algoritmo *polishing*

```

    }
} while (cnt != maxWait);

//Il valore di attesa finale è settato nei seguenti millisecondi
if (tmpSingleThread < 10000) //se minore di 10 secondi lo impongo a
    10 secondi
    tmpSingleThread = 10000;
else if (tmpSingleThread > 300000) //se maggiore di 5 minuti secondi
    lo impongo a 5 minuti
    tmpSingleThread = 300000;

//Il clock viene riavviato
clock.Restart();

do
{
    //Si attendono le restanti segnalazioni aggiornando ad ognuna di
    esse il timeout
    if (WaitHandle.SignalAndWait(ewhCntComplete, ewhCnt,
        (int)(tmpSingleThread - clock.ElapsedMilliseconds), false))
    {
        cnt++;
    }
    else
    {
        clock.Stop();

        //Tutte le istanze di CPLEX subiscono un abort sicuro se
        ancora in fase risolutiva
        CPLEX.Aborter aborter;

        foreach (CPLEX cplex in cplexList)
        {
            aborter = new CPLEX.Aborter();
            cplex.Use(aborter);
            aborter.Abort();
        }
        //Gestita la saturazione il thread di supporto può essere
        terminato al più presto possibile
        return;
    }
} while (cnt != sizeCh); //Tutti i figli sono stati generati senza
    saturazione, esco senza ulteriore
//attesa

//Prima di uscire si sblocca il thread che ha inoltrato l'ultima
    segnalazione
//evito il timeout di un secondo
ewhCntComplete.Set();
}

```

---

## TEST E RISULTATI

In questa sezione sono presentati i test eseguiti sugli algoritmi discussi nel testo. La presentazione è suddivisa in tre sottosezioni:

- Istanze con numero di nodi inferiore a 200 + algoritmi esatti;
- Istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti;
- Istanze con numero di nodi compreso tra 300 e 999 + algoritmi euristici;

Essendo il codice progettato in Visual Studio utilizzando il linguaggio C#, non è stato possibile utilizzare il cluster di calcolo offerto dal dipartimento di studio DEI dell'università di Padova. Pertanto la macchina su cui i test sono stati eseguiti è un normale PC di utilizzo quotidiano: da pretest eseguiti durante la fase di sviluppo, si era già notato come istanze di tagli superiore ai 300 nodi richiedessero tempi troppo alti per essere risolte da algoritmi esatti; Tenendo presente inoltre che sarebbero state richiesti multipli tentativi di risoluzione per ogni coppia istanza/algoritmo, si è deciso di optare per la suddivisione appena descritta.

I dettagli hardware della macchina utilizzata sono riportati di seguito ma è necessario far notare che è stato imposto un limite massimo all'utilizzo della CPU pari al 75%: si è voluto sfruttare l'arco minimo di tempo per i test in modo tale che l'ambiente rimanesse il più omogeneo possibile, pertanto un uso superiore delle prestazioni del PC per 2/4 giorni di seguito<sup>89</sup> avrebbe potuto arrecarvi danni.


CPU   Caches   Mainboard   Memory   SPD   Graphics   Bench   About									
Processor									
Name	Intel Core i5 6600K								
Code Name	Skylake	Max TDP	95.0 W						
Package	Socket 1151 LGA								
Technology	14 nm	Core Voltage	0.976 V						
Specification	Intel® Core™ i5-6600K CPU @ 3.50GHz								
Family	6	Model	E	Stepping	3				
Ext. Family	6	Ext. Model	5E	Revision	R0				
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3, TSX								
Clocks (Core #0)									
Core Speed	2598.73 MHz								
Multiplier	x 26.0 ( 8 - 39 )								
Bus Speed	100.00 MHz								
Rated FSB									
Cache									
L1 Data	4 x 32 KBytes				8-way				
L1 Inst.	4 x 32 KBytes				8-way				
Level 2	4 x 256 KBytes				4-way				
Level 3	6 MBytes				12-way				
Selection <span>Socket #1</span> <span>Cores 4</span> <span>Threads 4</span>									

Fig. 17: CPU

Aggiungiamo solamente infine che il sistema operativo è Windows 10 Pro versione 1703 installato su una memoria a stato solido con velocità di lettura/scrittura fino a 535 MB sec/445 MB sec.

<sup>89</sup>Attualmente ci troviamo in stagione estiva con alte temperature.

CPU	Caches	Mainboard	Memory	SPD	Graphics	Bench	About
L1D-Cache							
Size	32 KBytes		x 4				
Descriptor	8-way set associative, 64-byte line size						
L1I-Cache							
Size	32 KBytes		x 4				
Descriptor	8-way set associative, 64-byte line size						
L2 Cache							
Size	256 KBytes		x 4				
Descriptor	4-way set associative, 64-byte line size						
L3 Cache							
Size	6 MBytes						
Descriptor	12-way set associative, 64-byte line size						
Size							
Descriptor							
Speed							

Fig. 18: Caches

CPU	Caches	Mainboard	Memory	SPD	Graphics	Bench	About
Motherboard							
Manufacturer	ASUSTeK COMPUTER INC.						
Model	Z170-A		Rev 1.xx				
Chipset	Intel	Skylake		Rev.	07		
Southbridge	Intel	Z170		Rev.	31		
LPCIO	Nuvoton	NCT6793/NCT5563					
BIOS							
Brand	American Megatrends Inc.						
Version	2202						
Date	09/19/2016						
Graphic Interface							
Version	PCI-Express						
Link Width	x16		Max. Supported	x16			
Side Band Addressing							

Fig. 19: Scheda madre

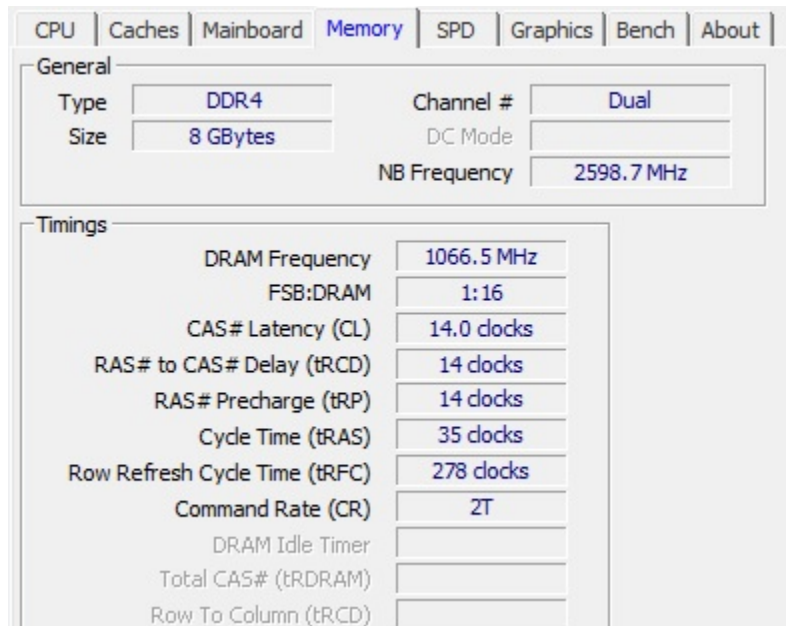


Fig. 20: RAM

Concludiamo questa introduzione spendendo alcune righe descrivendo alcuni aspetti comuni a tutti i test.

Nelle tre successive sottosezioni troviamo inizialmente una descrizione delle sigle utilizzate per identificare gli algoritmi utilizzati e l'indicazione del numero di run eseguite. Successivamente sono esposti i tempi medi di esecuzione, espressi in secondi, sotto forma tabellare utilizzando come tempo limite 30 minuti. Infine troviamo una serie di immagini che mostrano visivamente un confronto dei vari algoritmi per ogni istanza.

Per nessun test sono riportati i circuiti hamiltoniani trovati in quanto le istanze utilizzate sono tutte note in letteratura e già risolte all'ottimo. L'obiettivo che si vuole raggiungere è, per quanto riguarda gli algoritmi esatti un confronto dei loro tempi di esecuzione, mentre invece per quelli euristici un confronto del costo delle soluzioni da loro trovate e il valore ottimo noto entro un tempo limite fissato.

In ogni algoritmo sono state quindi rimosse tutte le stampe sia visuali attraverso GNUPlot che su file del circuito trovato ed anche la stampa su file del modello matematico finale comprendente i tagli generati. Le prove per gli algoritmi esatti sono state automatizzate in modo tale che per la stessa istanza venissero eseguiti in serie tutte le run necessarie ogni algoritmo<sup>90</sup>.

Nel caso in cui una o più run presentavano rallentamenti apparentemente non motivabili o improvvisi queste sono state ripetute per verificare che il problema non riguardasse fattori esterni determinati dalla macchina utilizzata. Tutti i valori riportati sono quindi verificati sotto questo punto di vista e dipendenti solamente dalla applicazione sviluppata.

<sup>90</sup>Ogni run è stata caratterizzata da un seme diverso sia per quanto riguarda CPLEX sia per quanto riguarda gli oggetti utilizzati per generare valori random.

## Istanze con numero di nodi inferiore a 200 + algoritmi esatti

Gli algoritmi esatti a disposizione sono i seguenti: Loop completamente esatto più le sue due varianti a prima fase euristica (linguaggio C#), utilizzo LazyConstraint Callback (linguaggio C#), utilizzo LazyConstraint e UserCut Callback (linguaggio C).

Il numero di run per ogni algoritmo è stato di 5: i valori riportati sono quindi una media aritmetica. Passiamo quindi alla descrizione delle sigle utilizzate:

- **LOOP**: metodo Loop completamente esatto;
- **L EGX**: metodo Loop con prima fase euristica dove EpGap è settato al  $X\%$ ;
- **L LAX**: metodo Loop con prima fase euristica dove sono abilitati i soli  $X$  lati a costo minore incidenti in un qualsiasi nodo;
- **L EGX LAY**: combinazione dei due punti precedenti;
- **LAZY**: utilizzo lazyconstraint callback in linguaggio C#;
- **USER**: utilizzo lazyconstraint callback e usercut callback in linguaggio C;

Notiamo che modificare un modello matematico attraverso l'eliminazione di alcune variabili può causare l'invalidazione di tutte le soluzioni originali: questi casi sono di seguito riconoscibili da un tempo di esecuzione medio pari a 0,00 secondi.

	LOOP	L EG5	L EG10	L LA5	L LA10	L EG5 LA5	L EG10 LA10	LAZY	USER
<b>berlin52</b>	0,111	0,083	0,073	0,092	0,053	0,096	0,051	0,028	0,024
<b>st70</b>	0,628	0,577	0,579	0,000	0,193	0,000	0,187	0,298	0,363
<b>eil76</b>	0,297	0,254	0,235	0,204	0,075	0,204	0,075	0,107	0,084
<b>pr76</b>	3,586	3,572	3,305	0,000	2,958	0,000	2,589	6,344	3,178
<b>rat99</b>	1,040	1,010	1,041	0,845	0,458	0,670	0,450	0,396	0,468
<b>kroA100</b>	2,009	1,959	1,891	1,544	0,770	1,504	0,777	0,915	1,508
<b>kroB100</b>	3,745	4,016	3,031	1,767	1,820	2,026	1,699	0,785	1,026
<b>kroC100</b>	2,306	1,850	1,908	2,197	0,823	1,878	0,821	0,648	0,996
<b>kroD100</b>	2,397	1,758	1,990	2,547	0,896	1,903	0,934	0,514	0,769
<b>kroE100</b>	2,017	2,243	2,290	2,168	1,137	1,793	0,936	1,084	0,900
<b>eil101</b>	0,952	0,851	0,802	0,426	0,370	0,433	0,414	0,363	0,342
<b>lin105</b>	1,402	0,897	0,959	0,664	1,000	0,625	0,588	0,545	1,071
<b>bier127</b>	1,862	2,228	3,077	1,251	0,981	1,222	0,905	1,093	1,636
<b>ch130</b>	2,013	2,213	2,198	1,625	0,980	1,911	1,052	1,373	1,589
<b>pr144</b>	10,247	6,362	4,023	0,000	7,427	0,000	5,703	3,300	2,020
<b>kroA150</b>	9,446	8,455	7,329	3,875	4,524	3,536	4,096	2,943	3,058
<b>kroB150</b>	15,815	10,842	12,213	9,041	7,643	5,165	8,266	8,473	5,067
<b>ch150</b>	4,563	5,158	6,103	2,348	2,612	3,034	3,987	2,413	2,481
<b>pr152</b>	4,806	3,961	3,610	0,000	0,000	0,000	0,000	2,638	3,130
<b>u159</b>	3,310	2,816	2,766	1,538	2,146	1,339	2,364	1,876	2,759
<b>rat195</b>	25,914	19,111	19,606	13,473	15,830	11,534	14,198	13,354	4,130

Tabella 1: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti



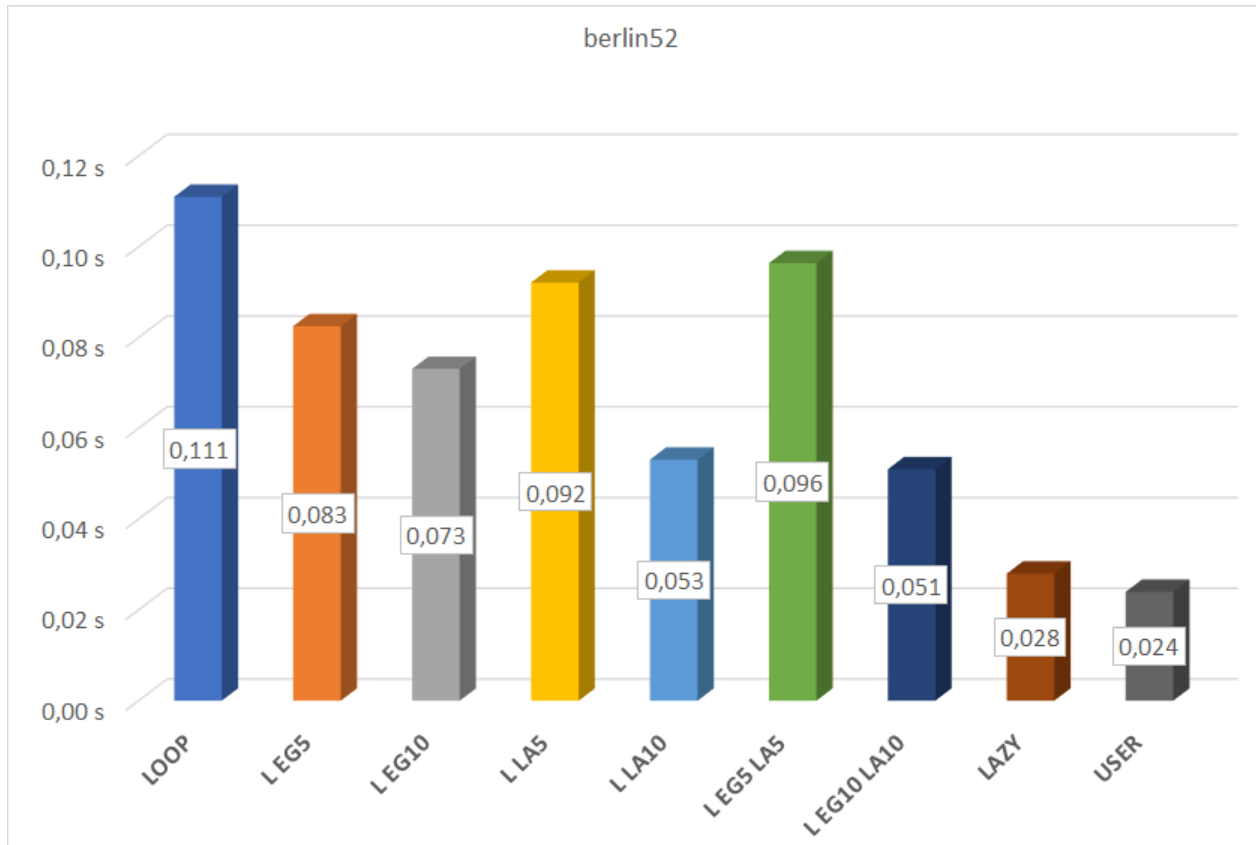


Fig. 21: berlin52

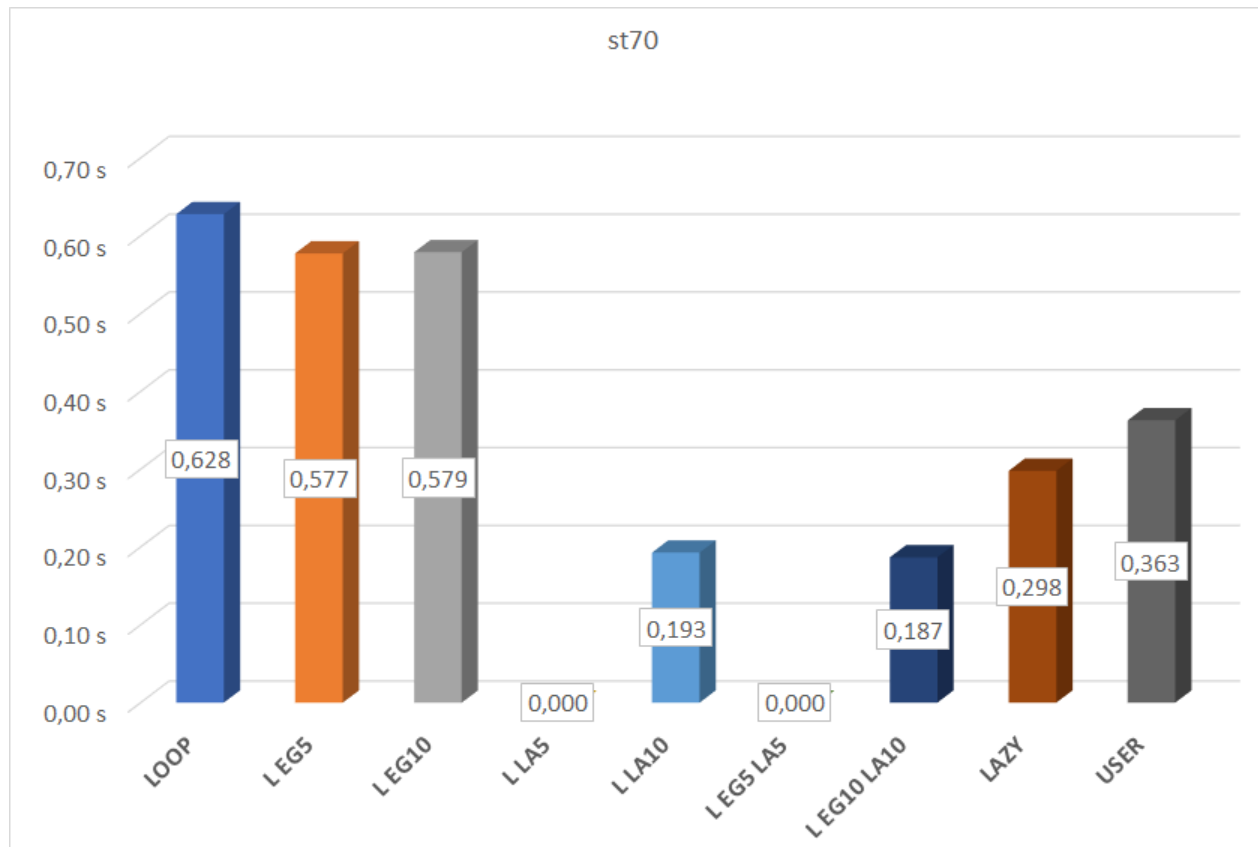


Fig. 22: st70

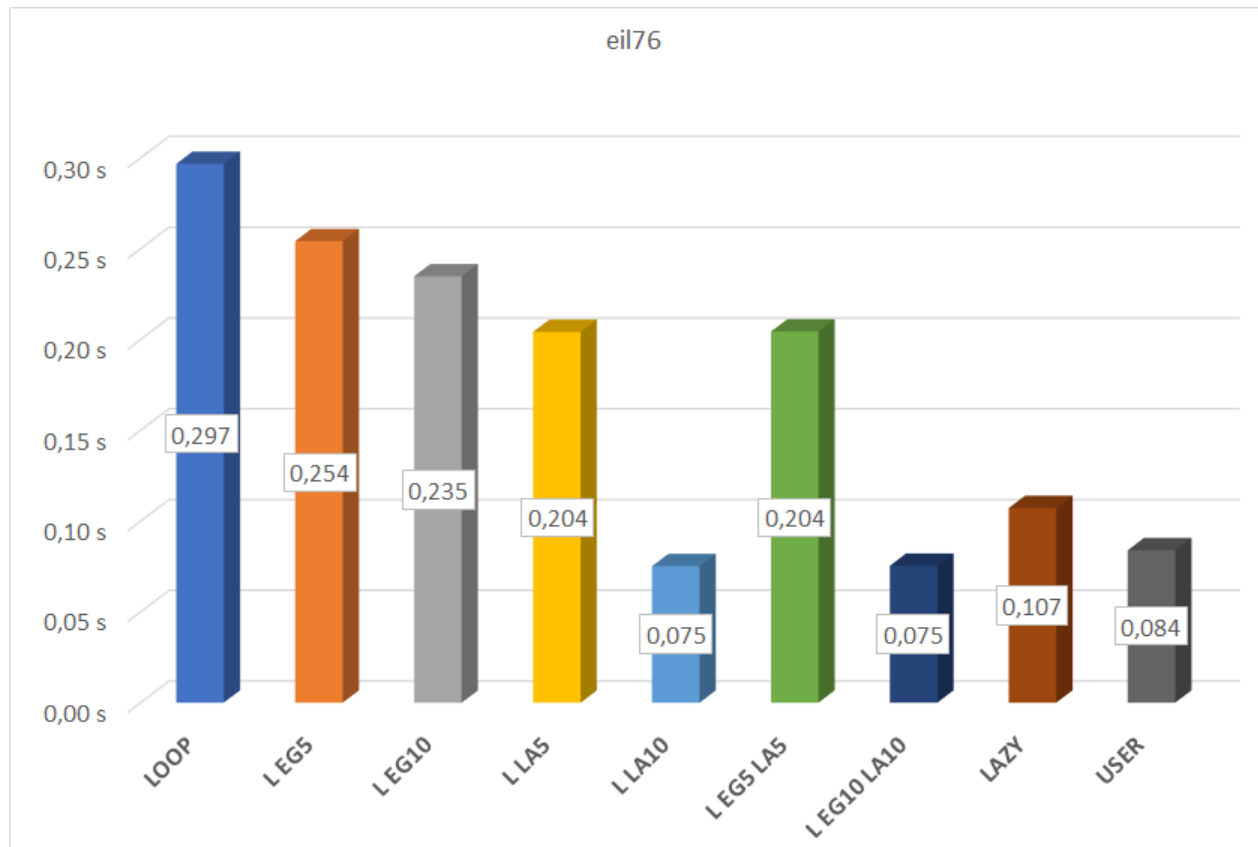


Fig. 23: eil76

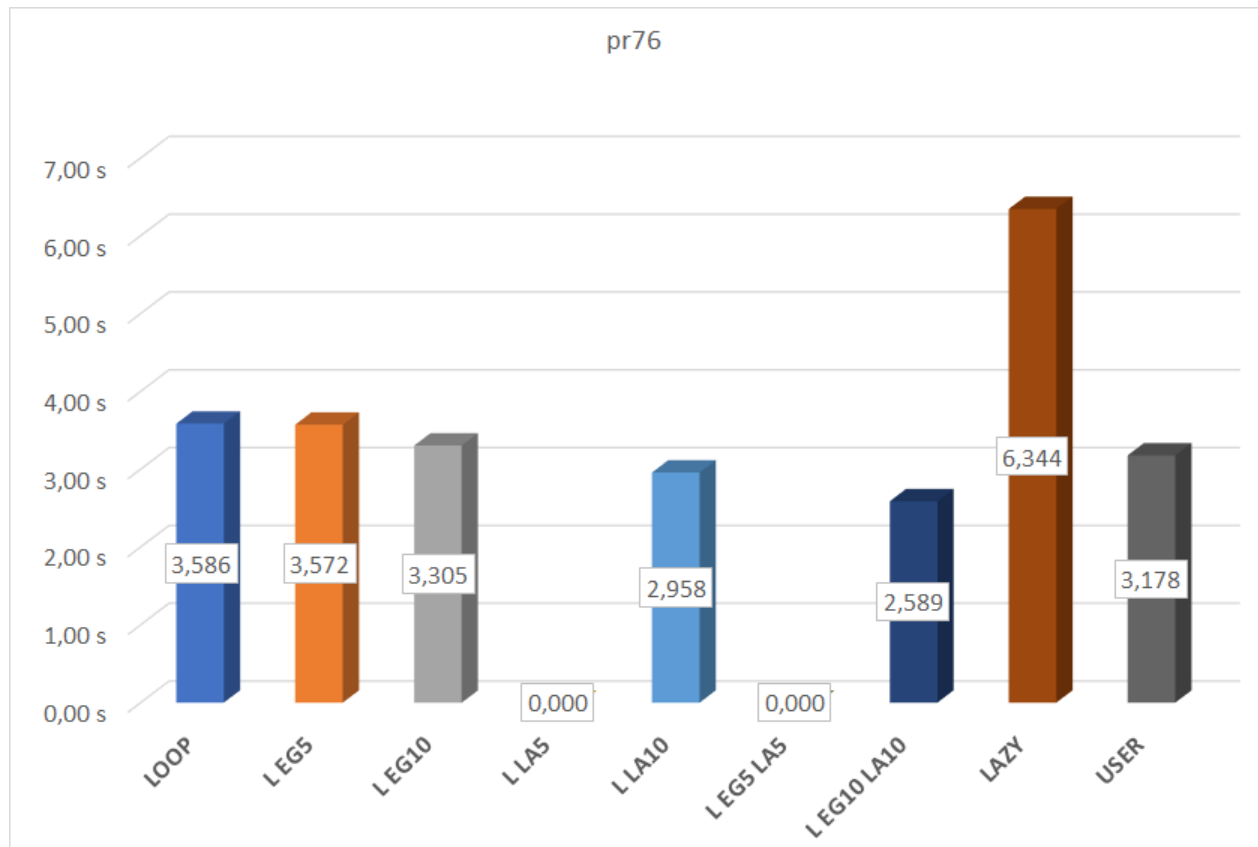


Fig. 24: pr76

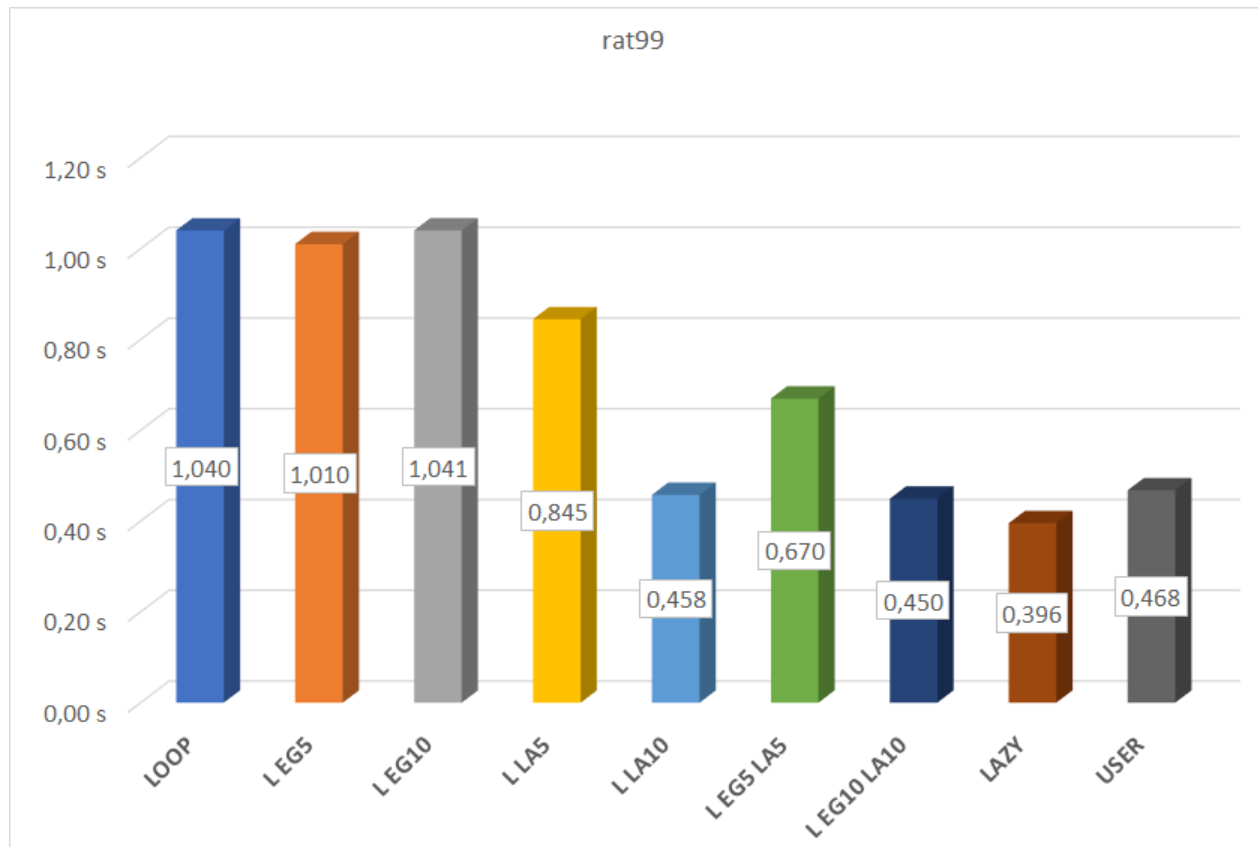


Fig. 25: rat99

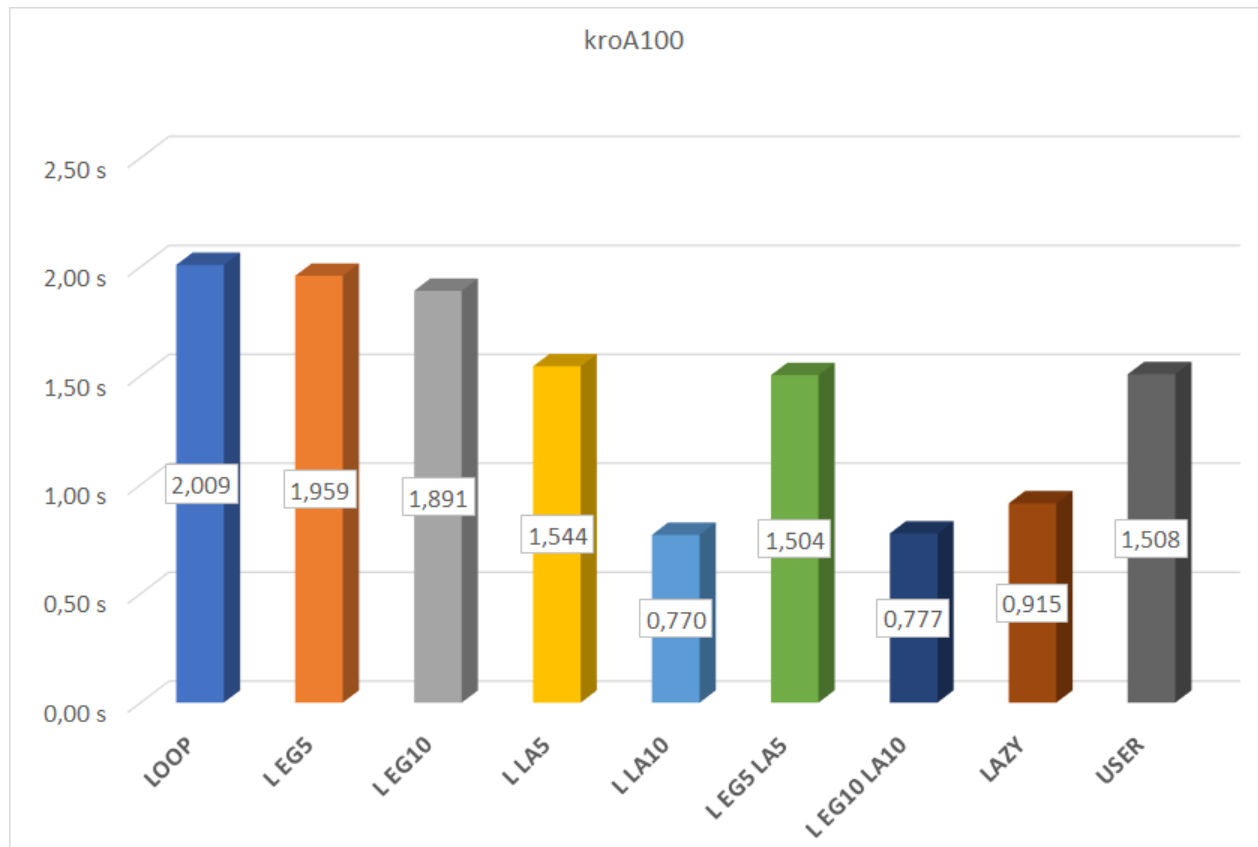


Fig. 26: kroA100

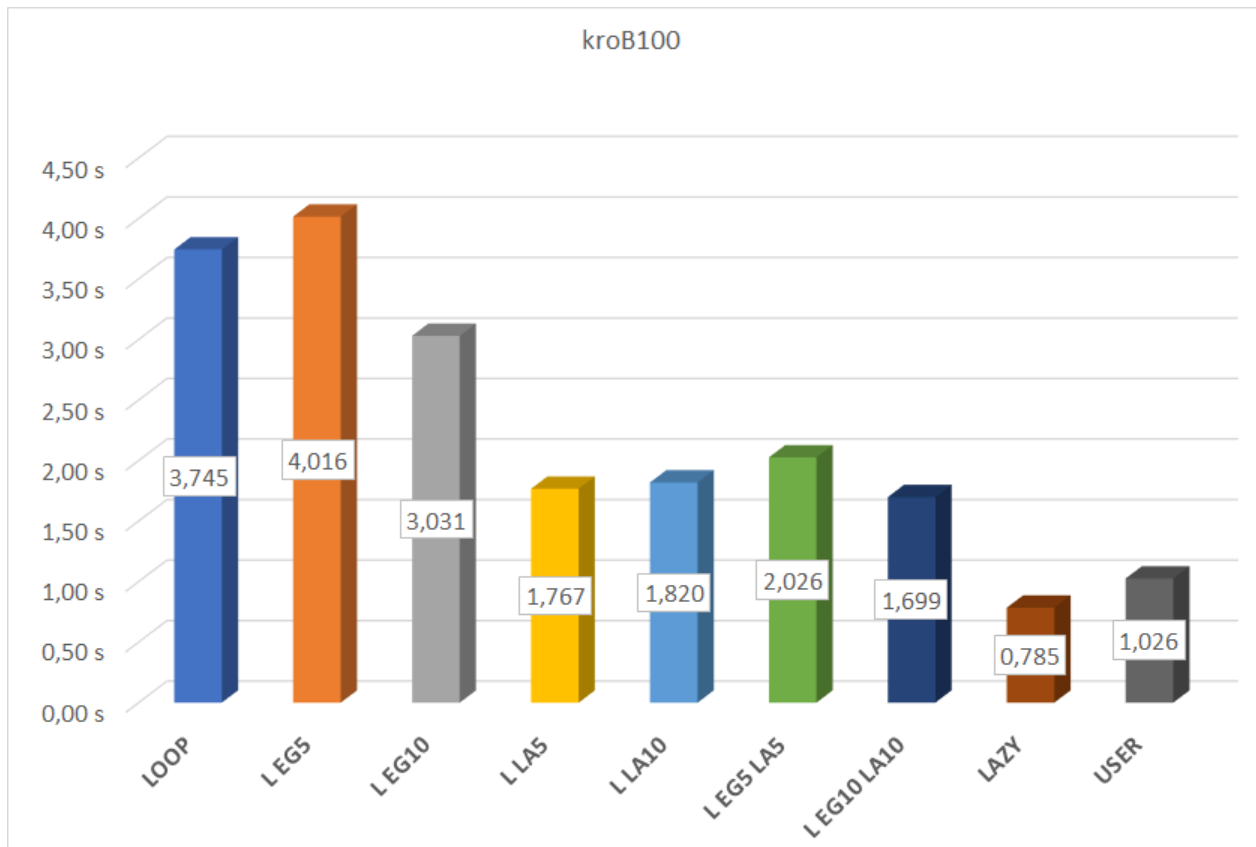


Fig. 27: kroB100

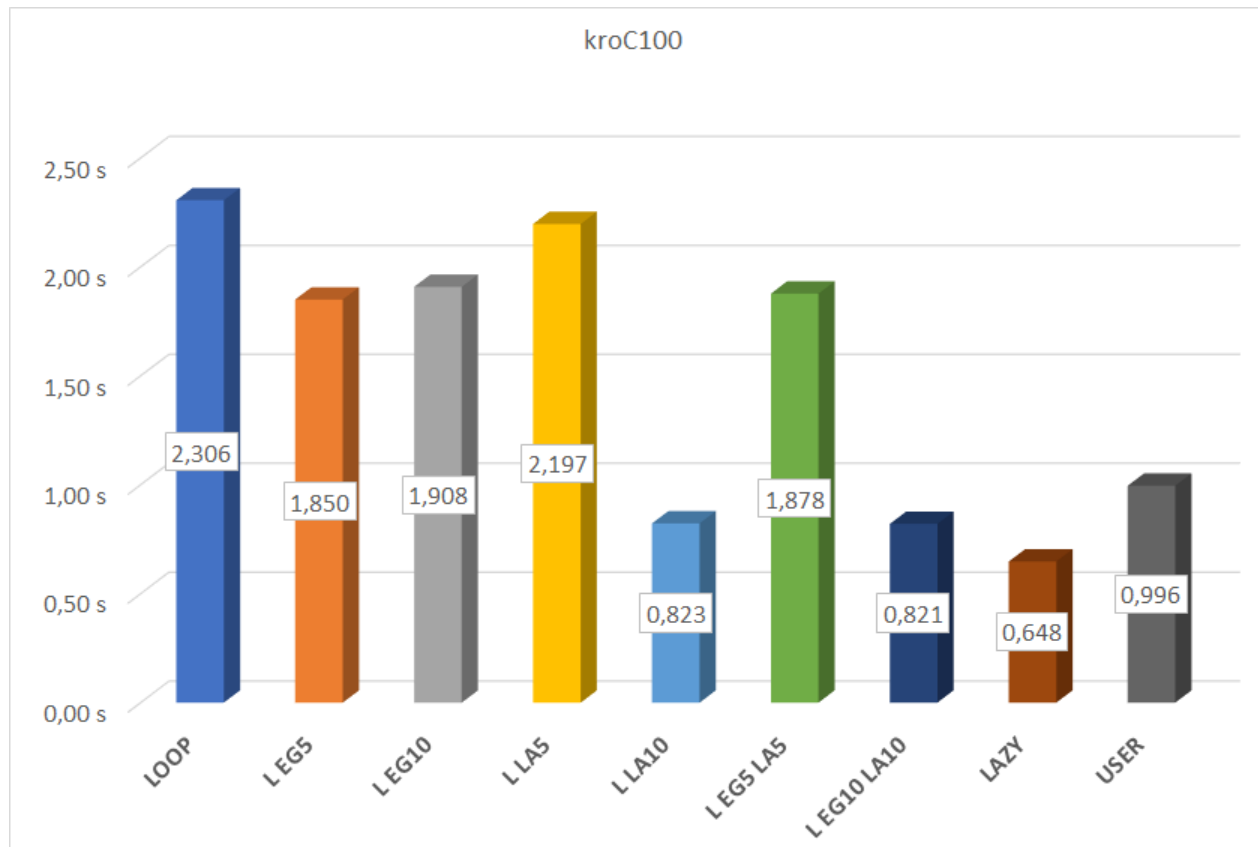


Fig. 28: kroC100



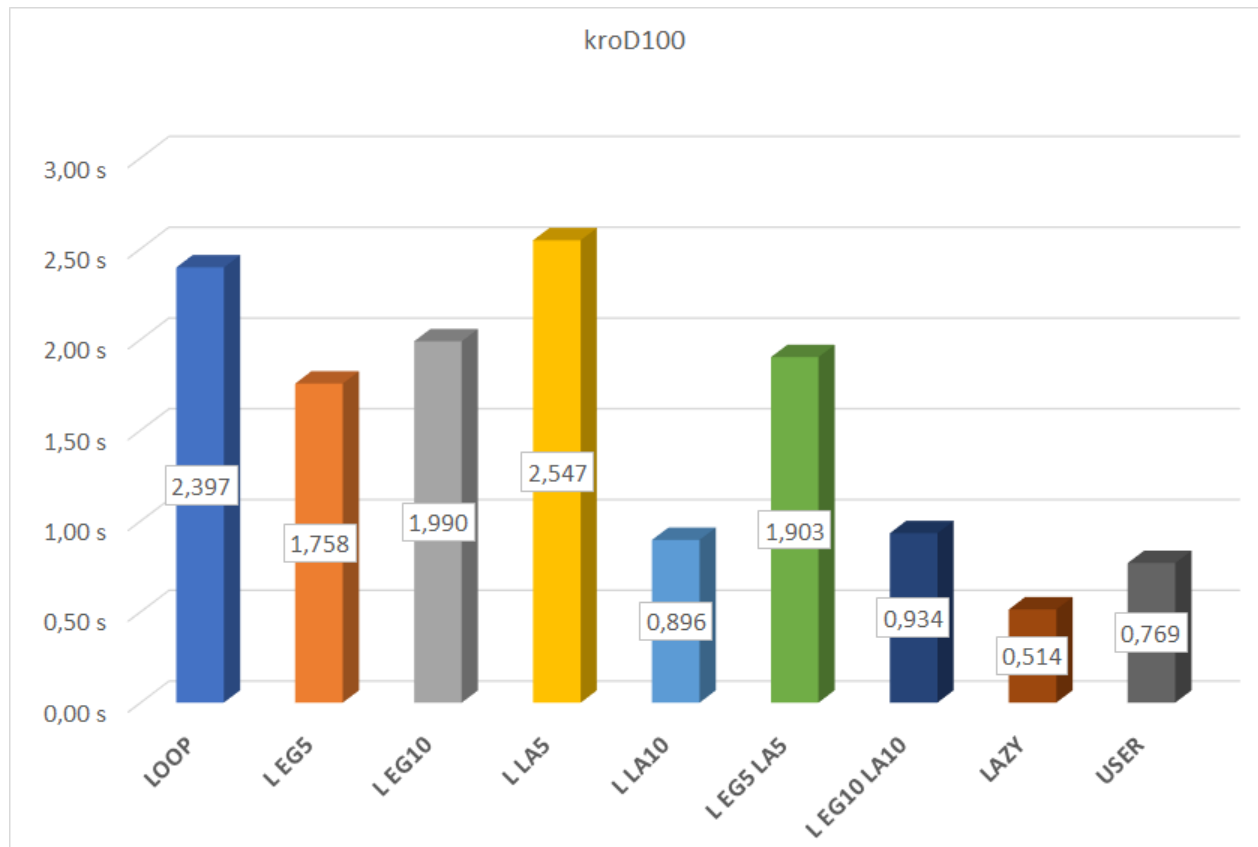


Fig. 29: kroD100

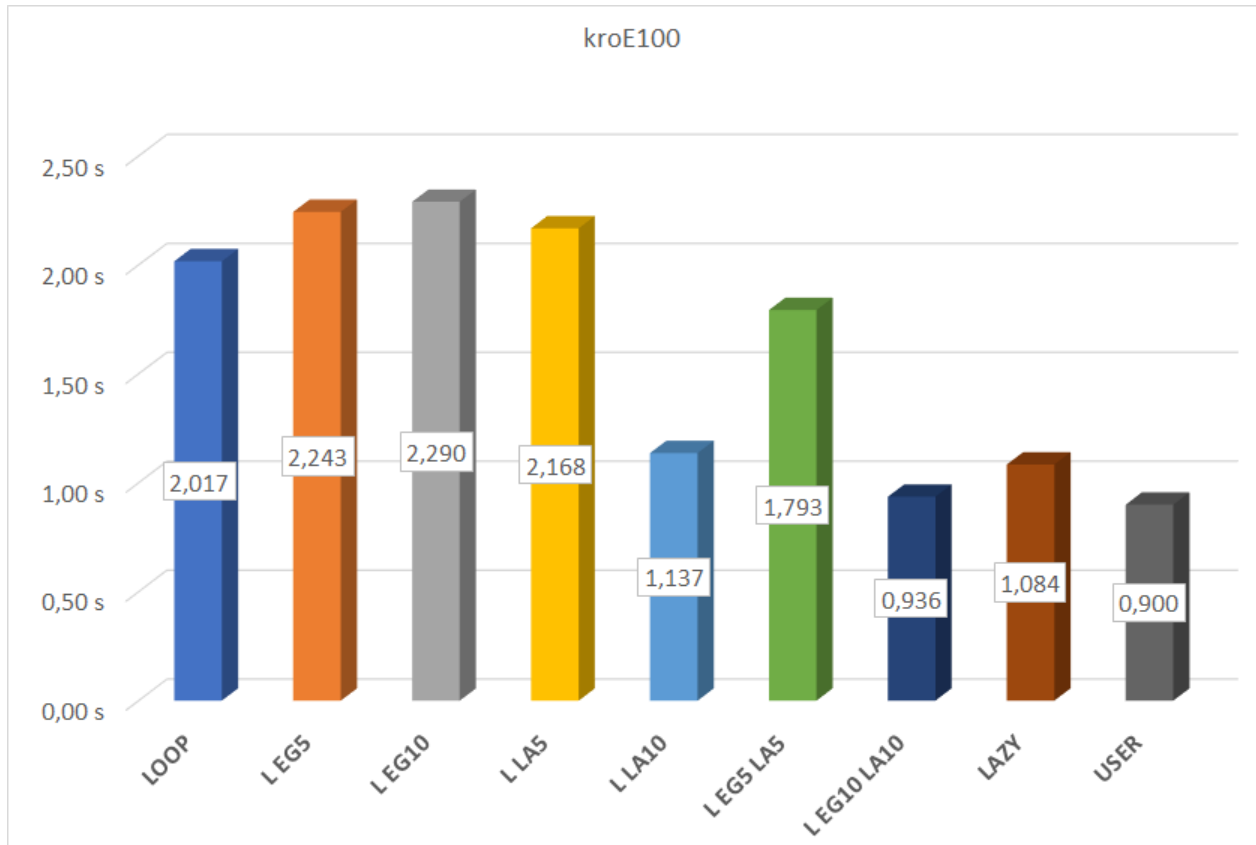


Fig. 30: kroE100

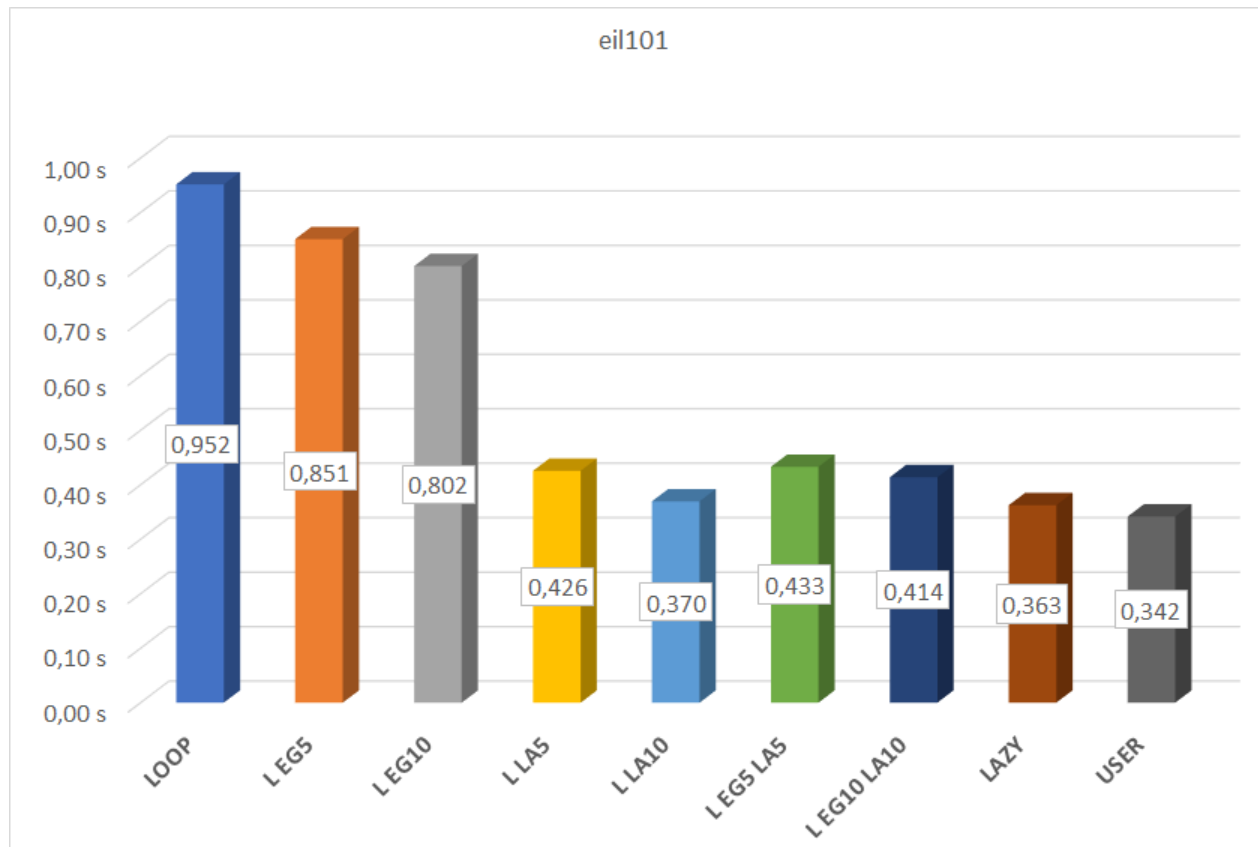


Fig. 31: eil101

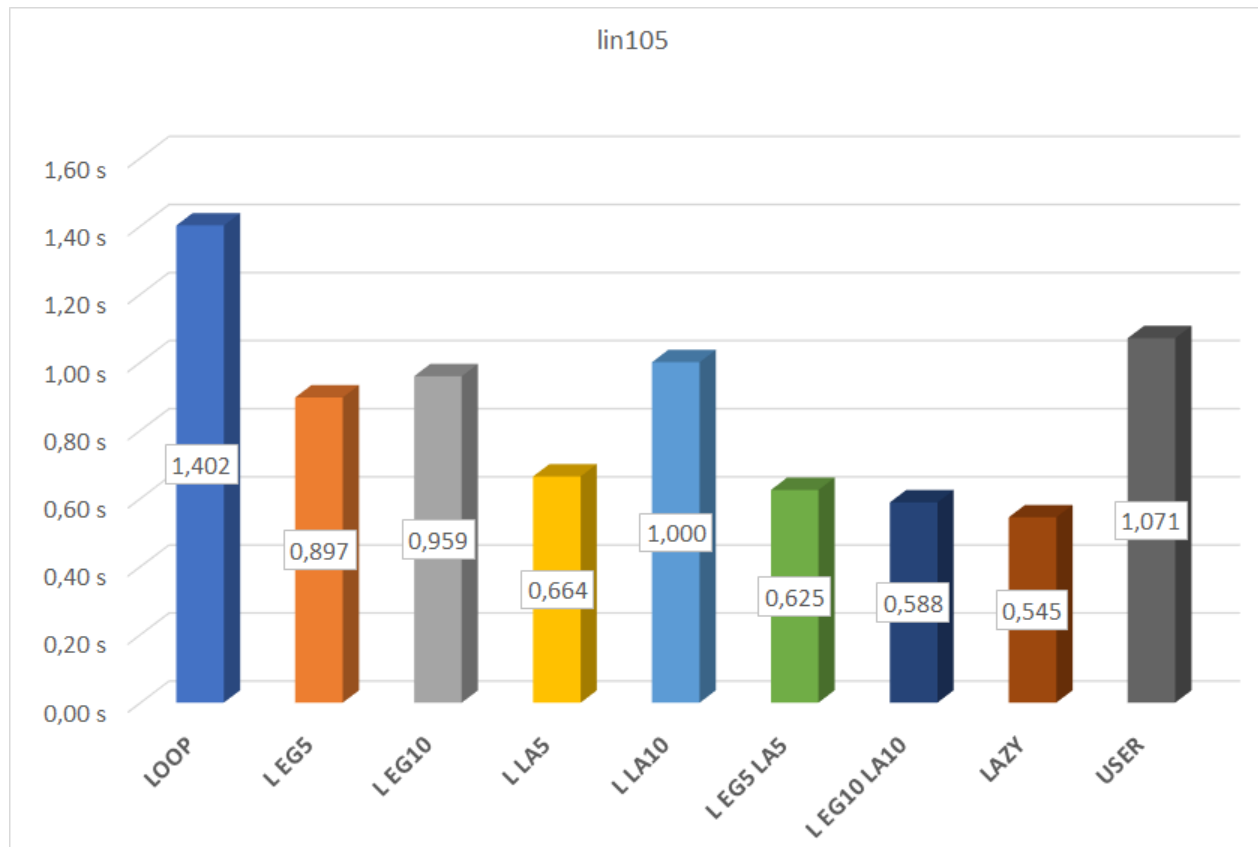


Fig. 32: lin105

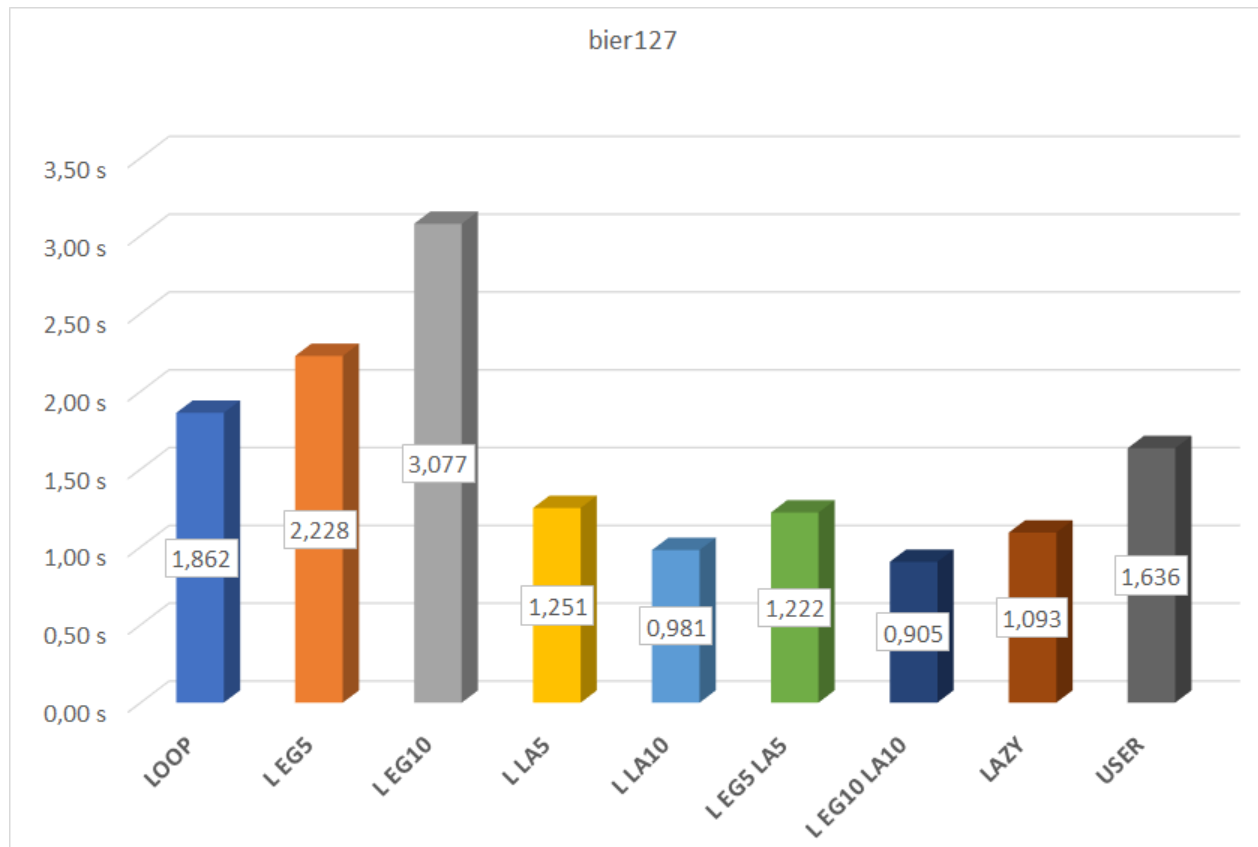


Fig. 33: bier127

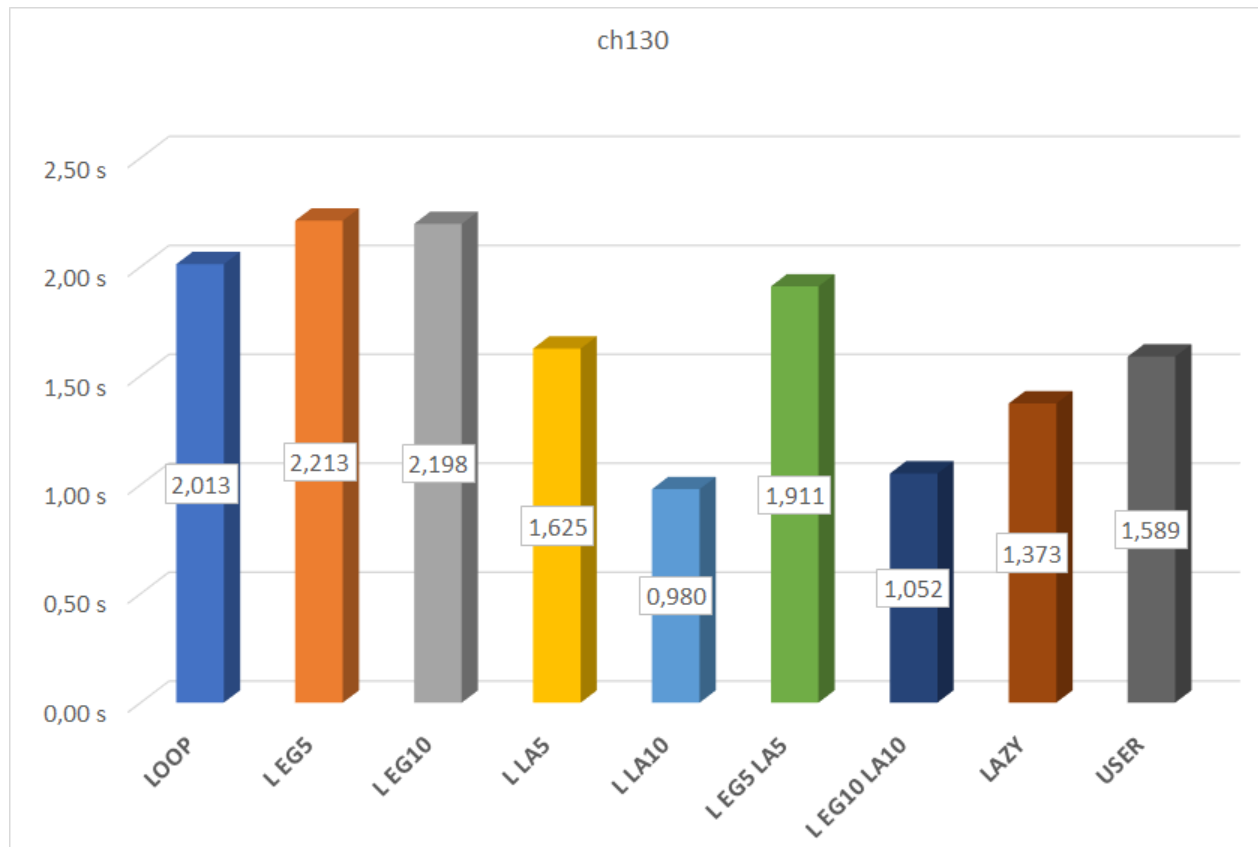


Fig. 34: ch130

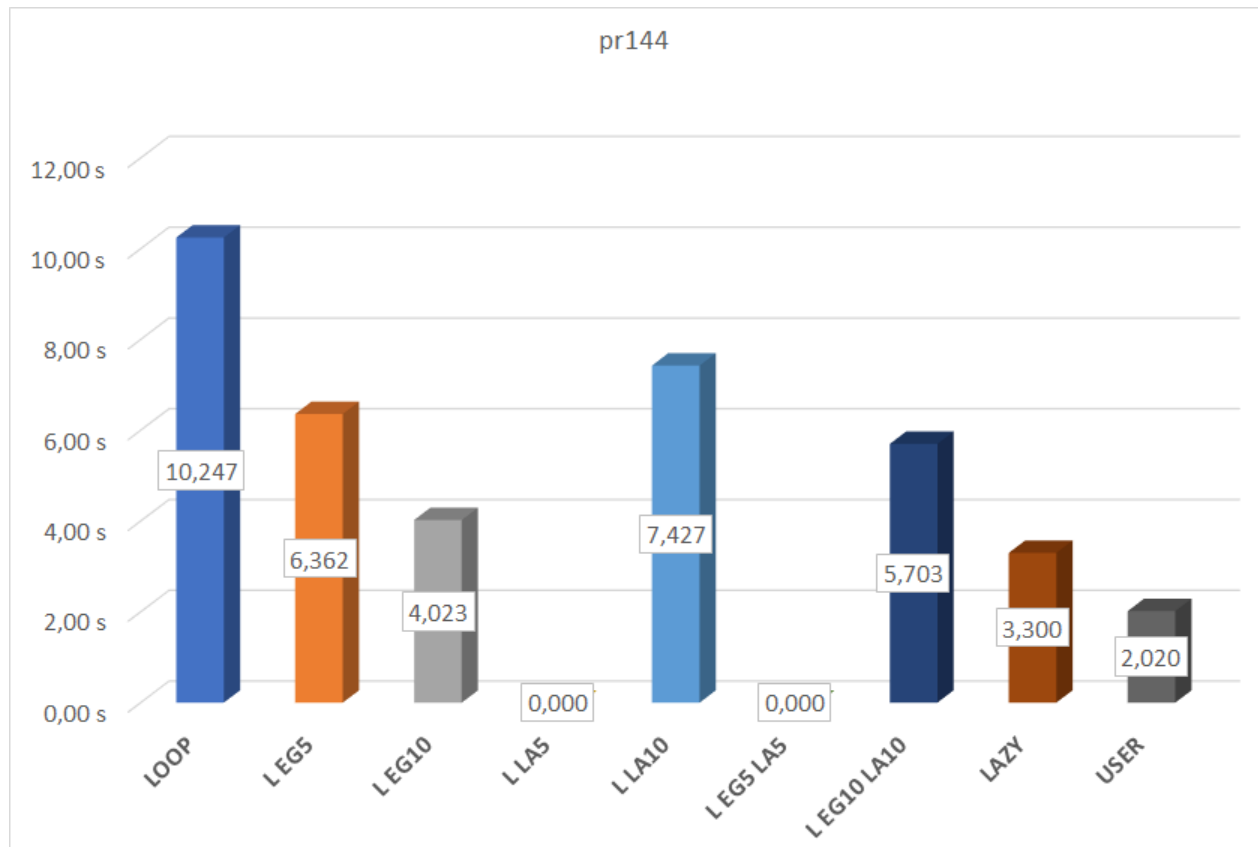


Fig. 35: pr144

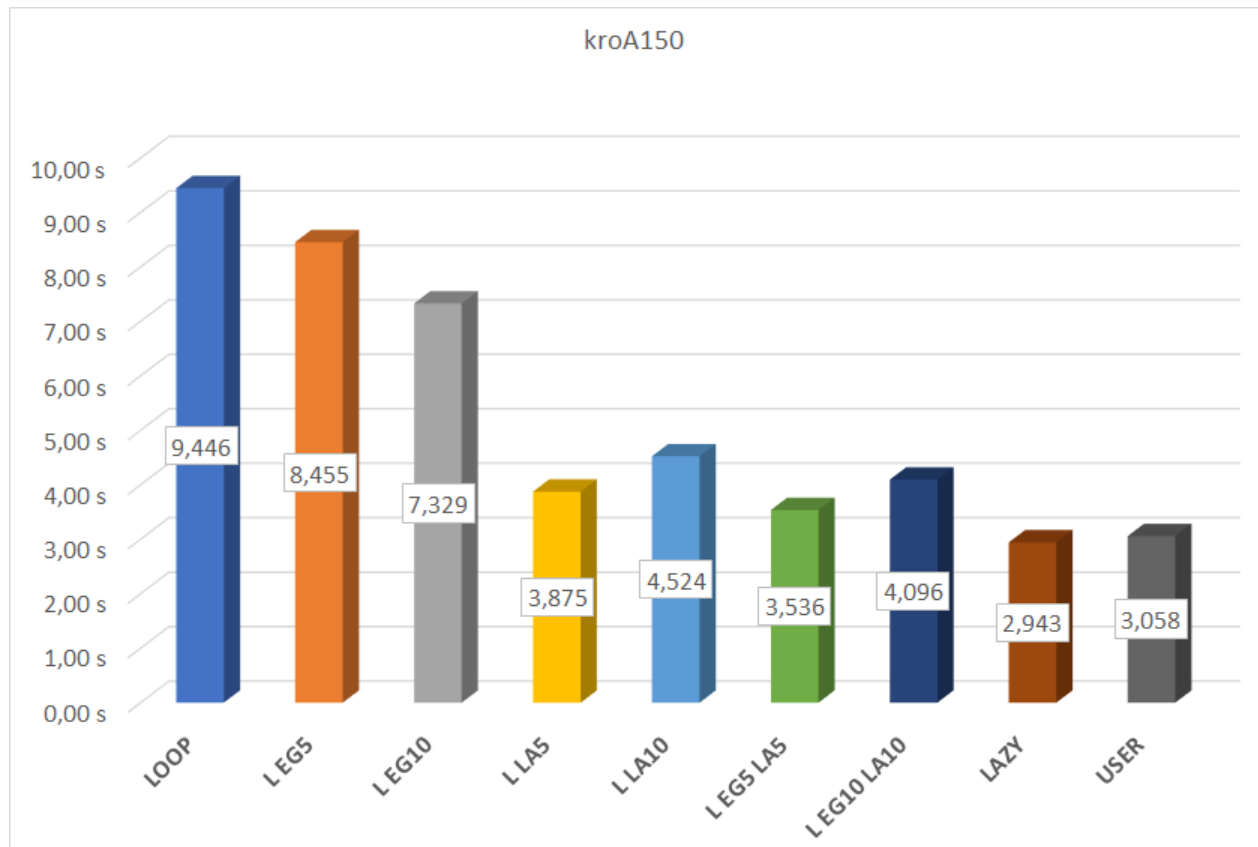


Fig. 36: kroA150



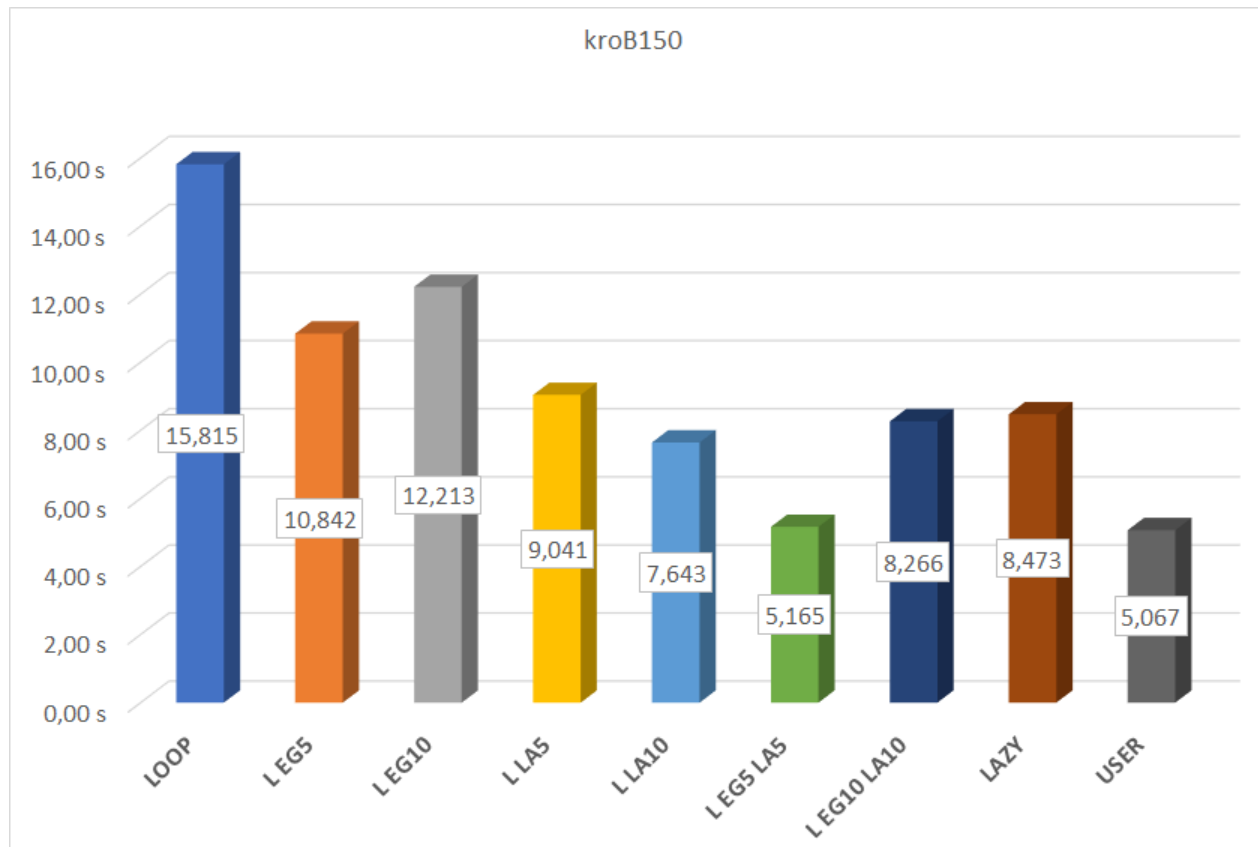


Fig. 37: kroB150

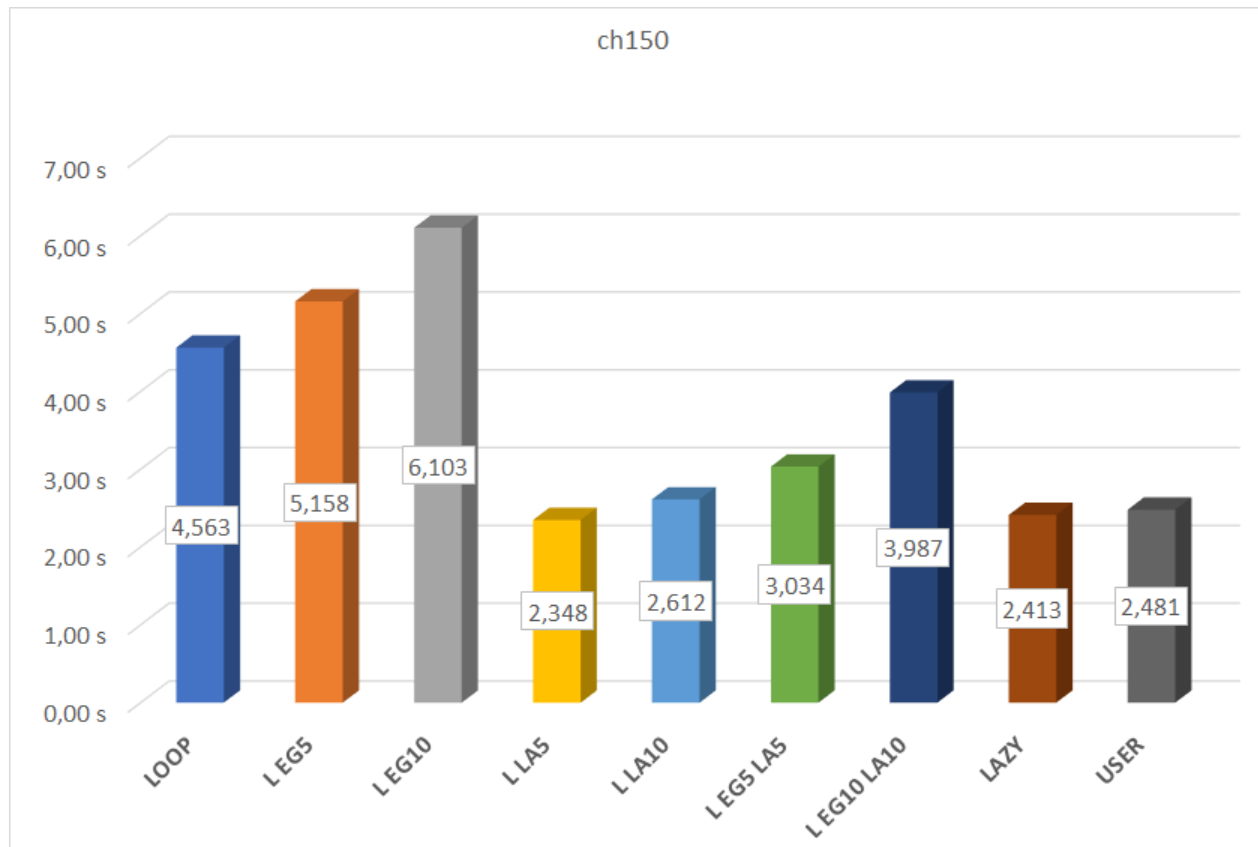


Fig. 38: ch150

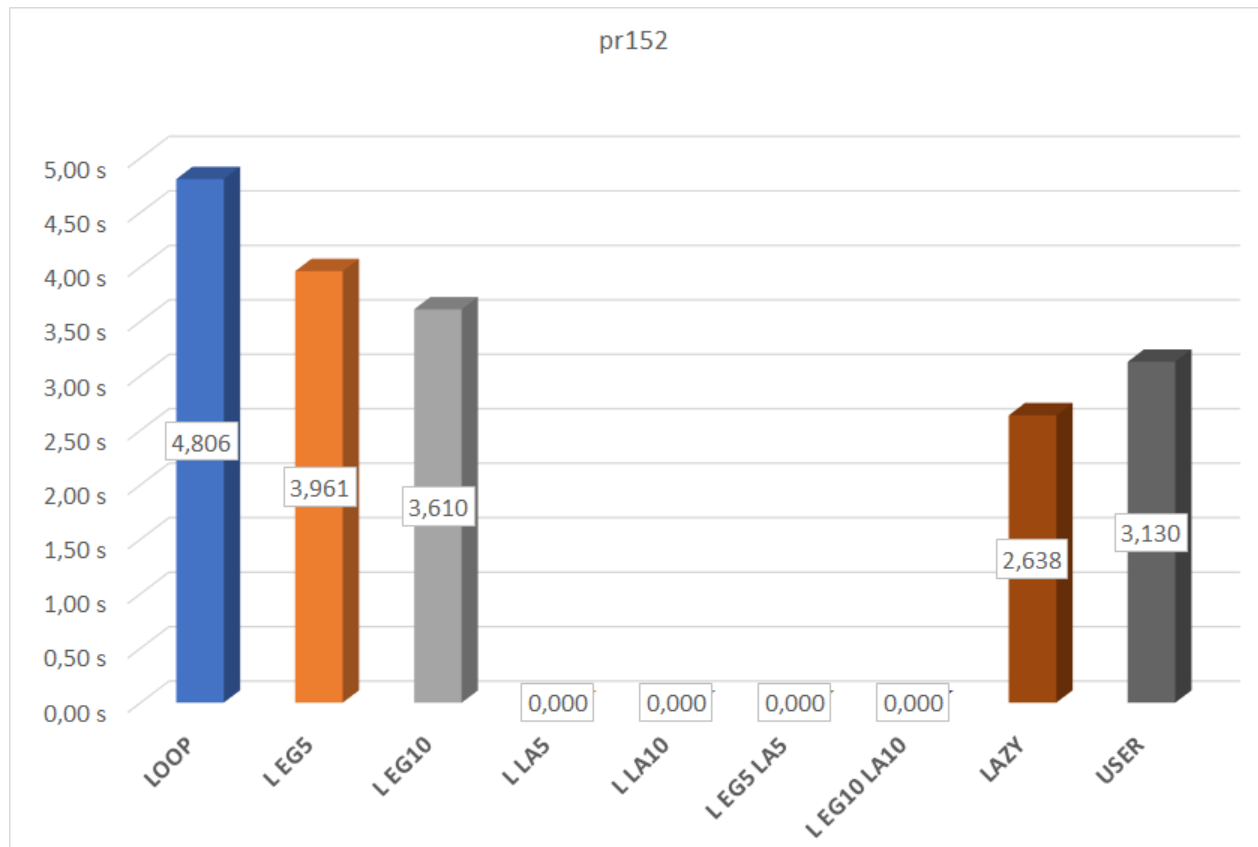


Fig. 39: pr152

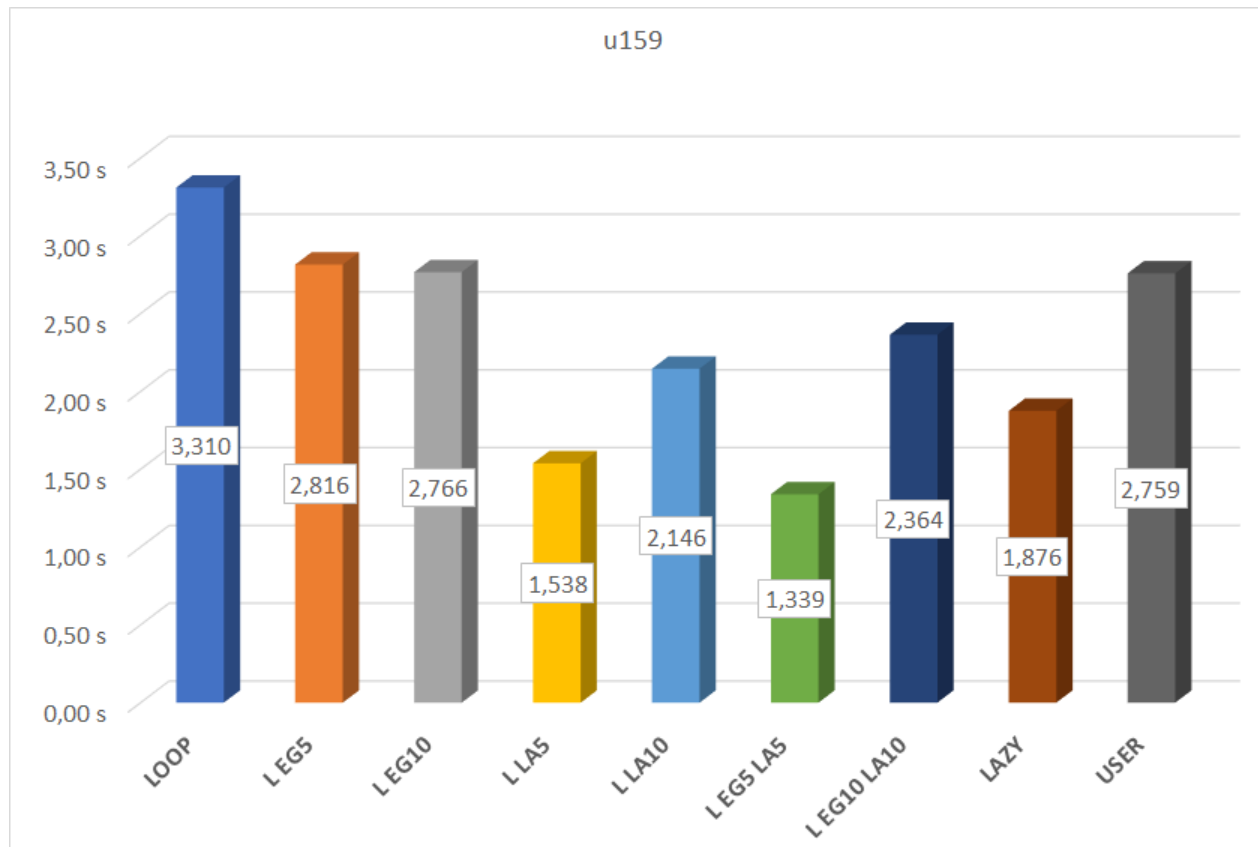


Fig. 40: u159

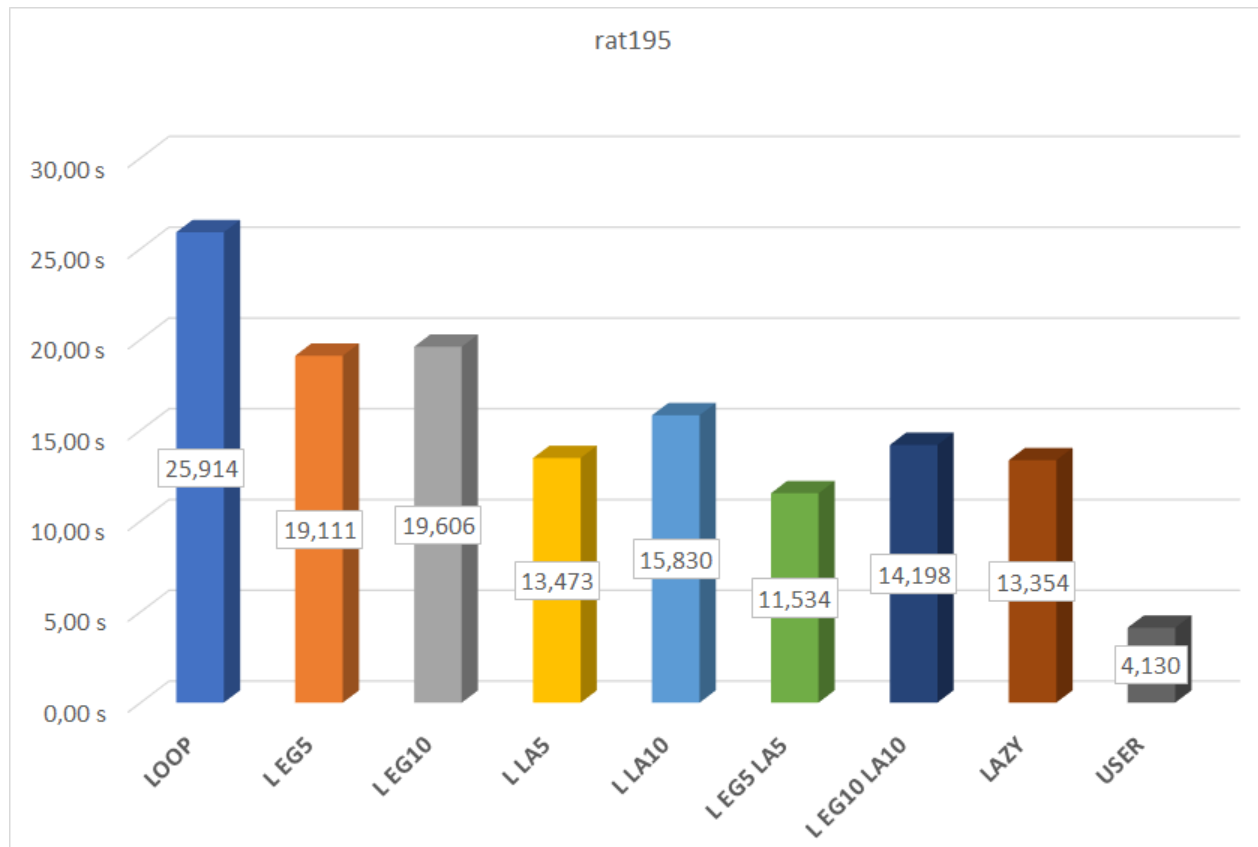


Fig. 41: rat195

### Instanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

Questo set di risultati ha le stesse identiche premesse indicate per il precedente se non che il numero di run eseguito per ogni coppia istanza/algoritmo è pari a 2 e non 5. Il motivo di tale scelta è puramente per una questione temporale.

Come più volte ripetuto è stato posto un tempo limite pari a 30 minuti il quale è stato raggiunto dall'istanza **pr99** se risolta attraverso l'utilizzo della sola *LazyConstraint Callback*: il valore medio comunicato da CPLEX per ottenere il risultato ottimo era in questo di circa lo 0,36%.

	LOOP	L EG5	L EG10	L LA5	L LA10	L EG5 LA5	L EG10 LA10	LAZY	USER
<b>kroA200</b>	26,12	35,36	35,02	23,71	25,25	20,73	23,20	61,62	65,22
<b>kroB200</b>	10,92	13,53	13,93	4,86	6,44	5,67	5,11	6,05	5,21
<b>tsp225</b>	28,10	26,72	28,15	11,61	15,46	14,85	15,20	23,49	6,70
<b>pr226</b>	326,40	32,52	16,64	0,00	0,00	0,00	0,00	7,97	4,90
<b>gil262</b>	26,86	40,91	27,55	8,95	14,02	18,27	20,29	79,30	37,50
<b>a280</b>	19,00	29,20	16,74	5,03	6,48	4,75	10,06	6,46	5,56
<b>pr299</b>	102,89	142,75	128,17	67,00	85,60	97,19	101,65	1800,00	27,52

Tabella 2: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

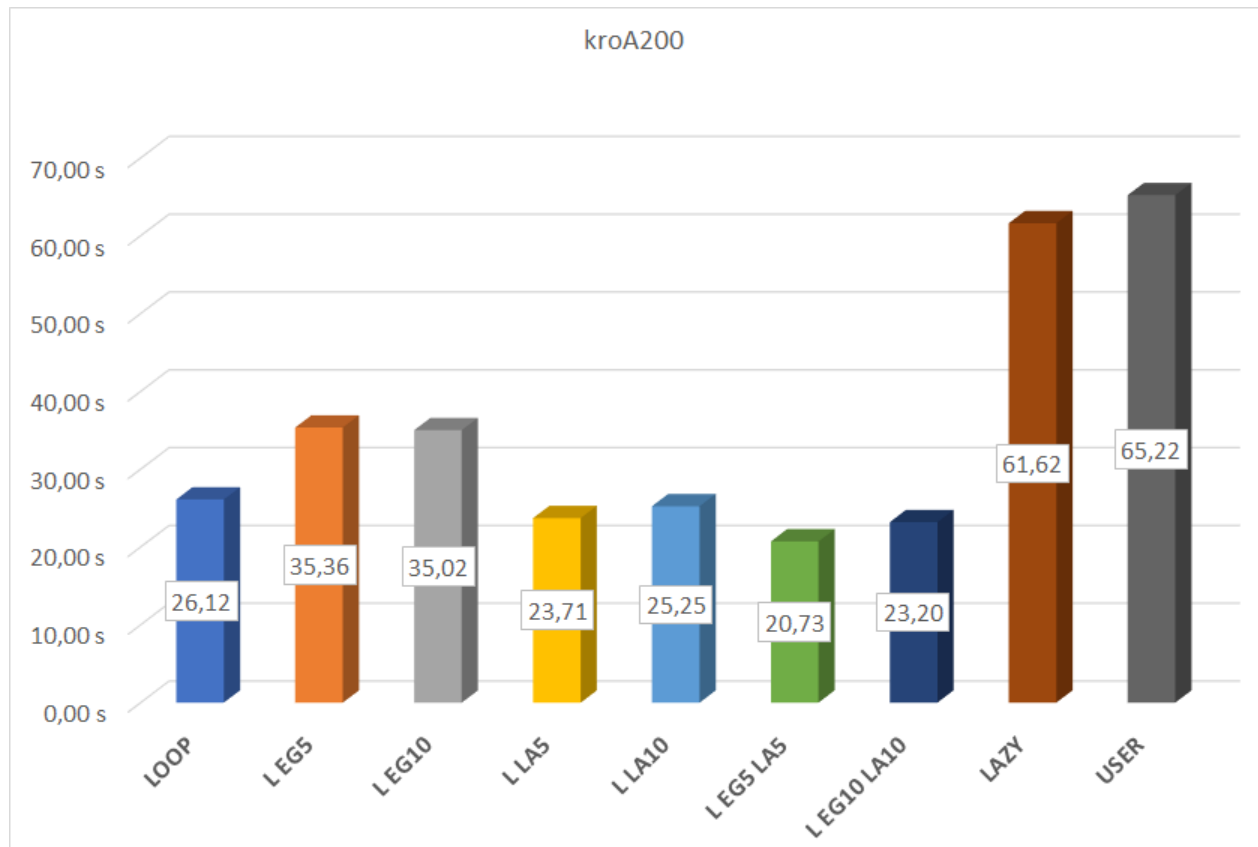


Fig. 42: kroA200

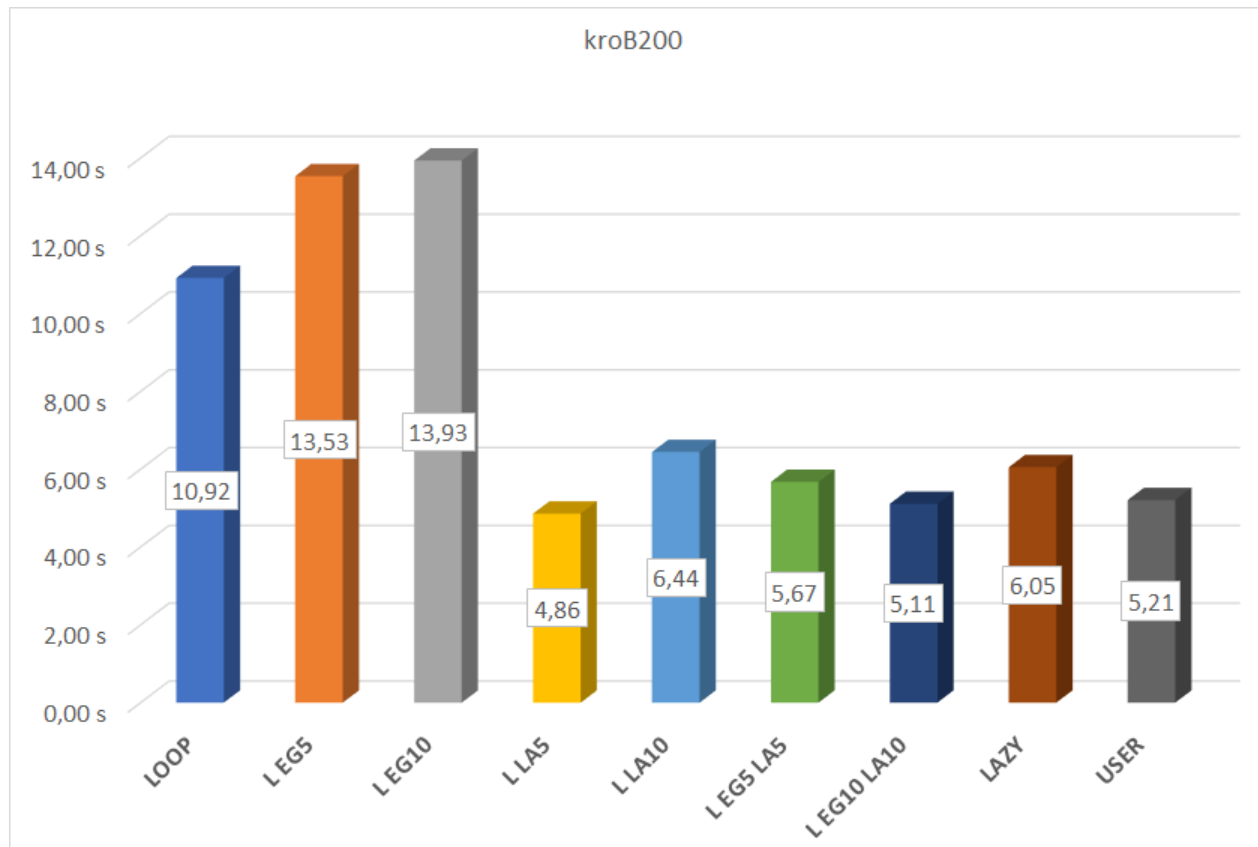


Fig. 43: kroB200



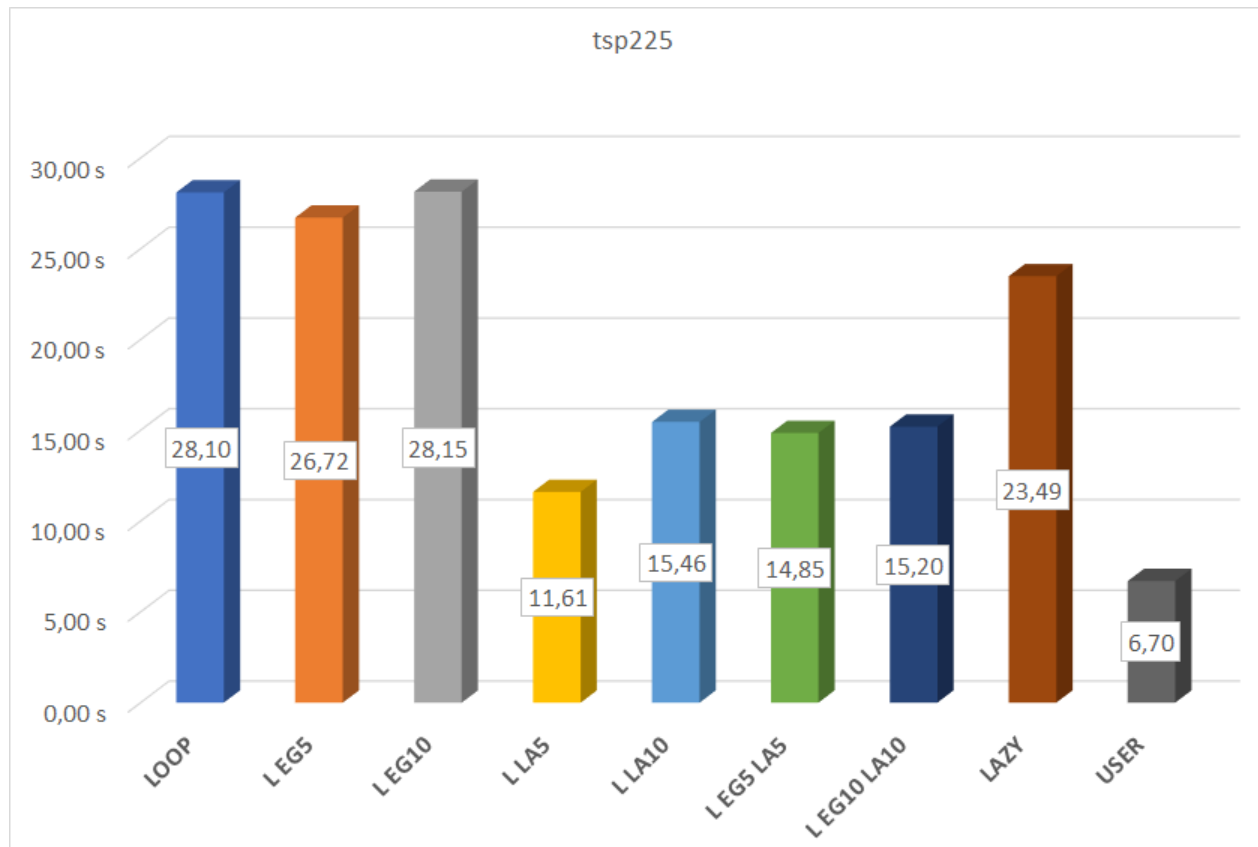


Fig. 44: tsp225

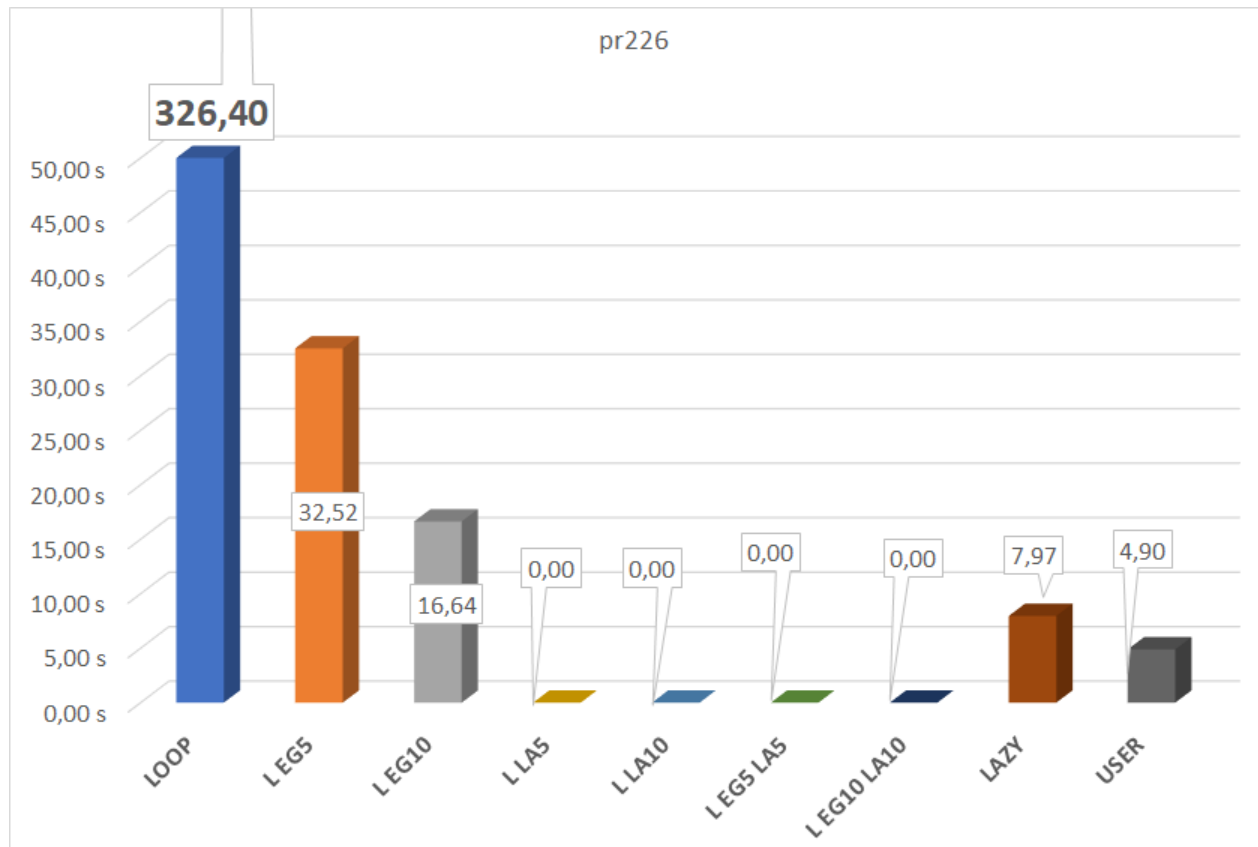


Fig. 45: pr226

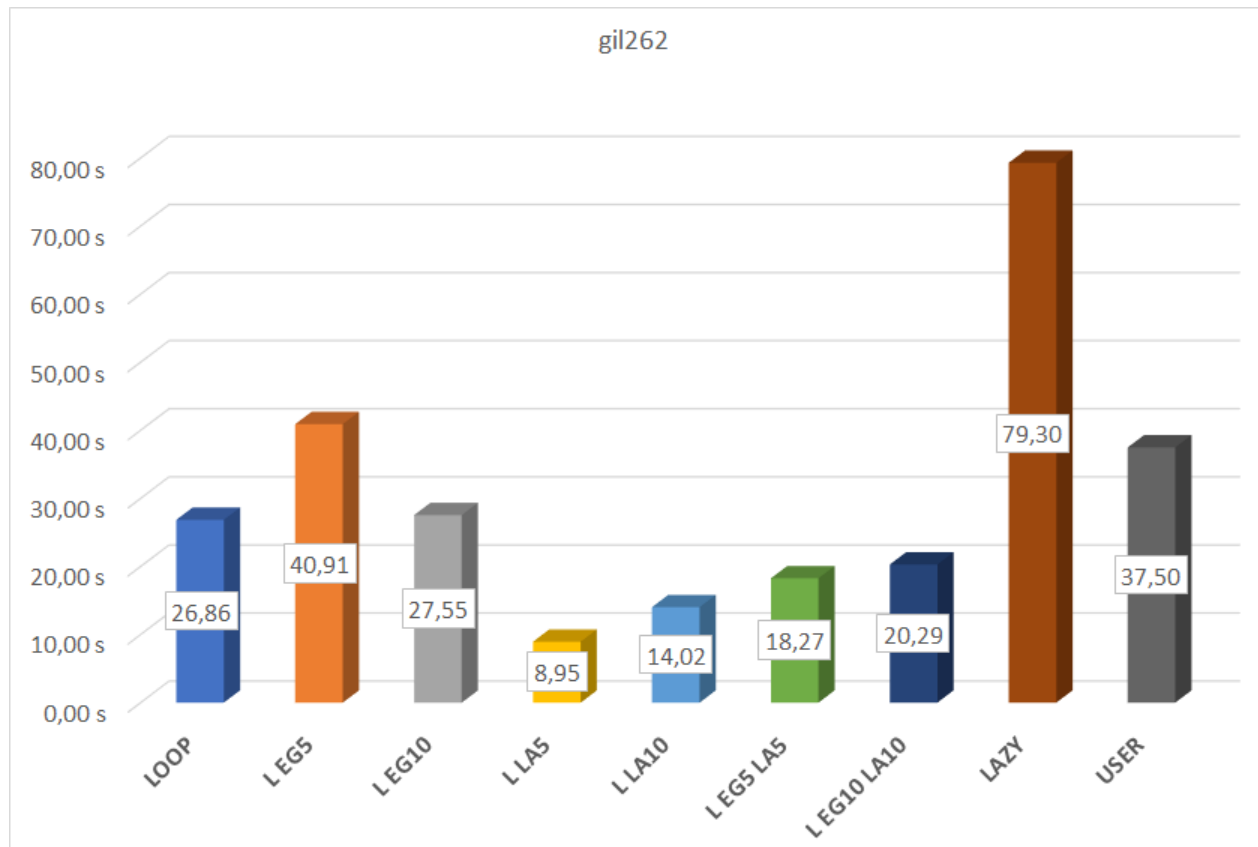


Fig. 46: gil262

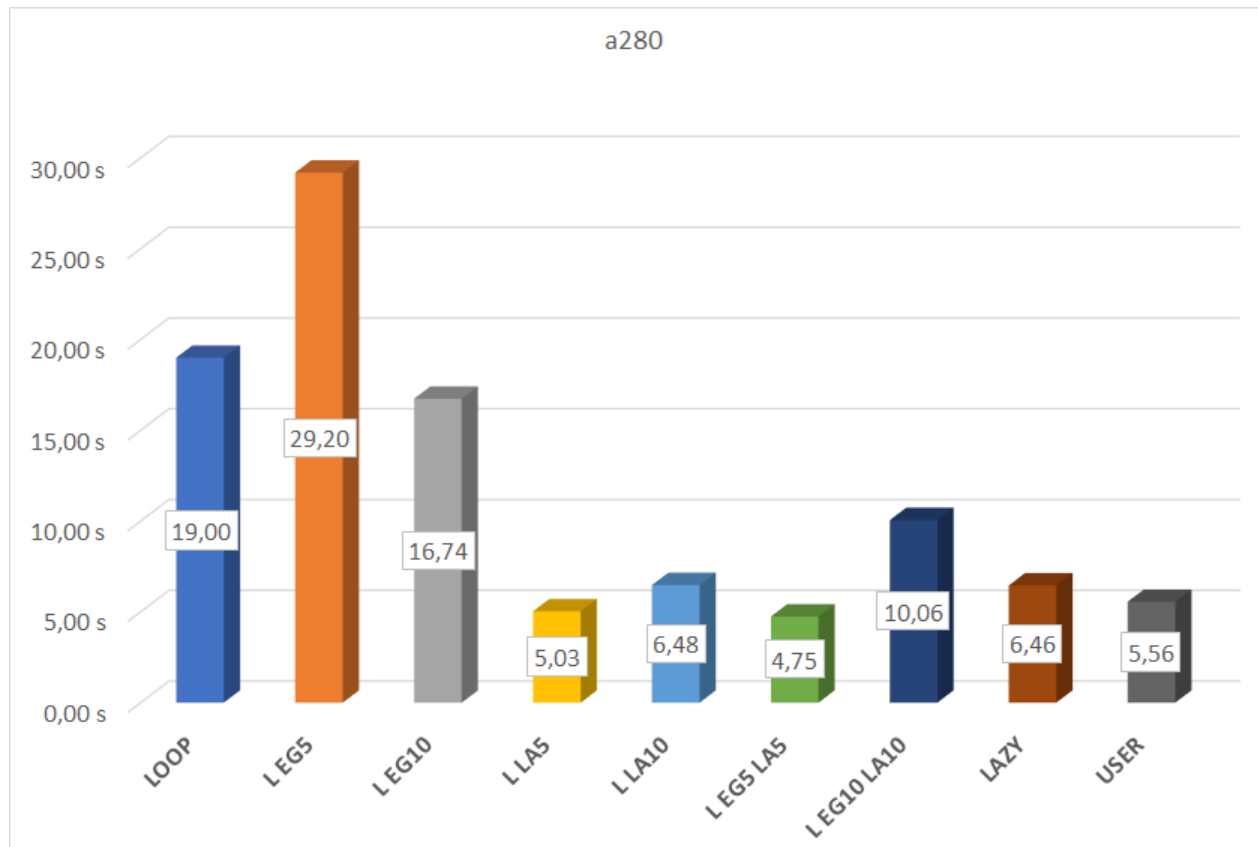


Fig. 47: a280

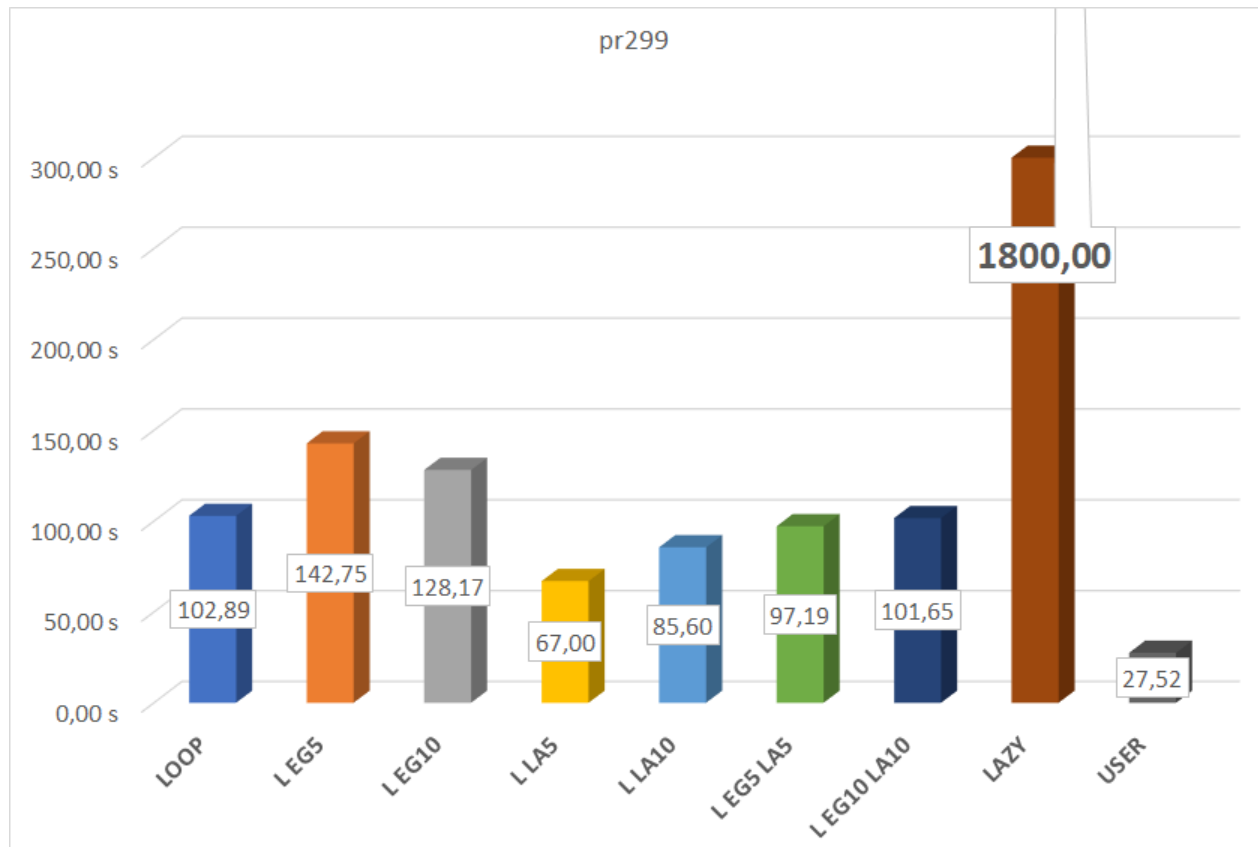


Fig. 48: pr299

**Istanze con numero di nodi compreso tra 300 e 999 + algoritmi euristici**  
**MULTI START**

lin318 MULTI START (costo ottimo 42029)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
44098	0,6	45374	0,6	45153	0,6	45442	0,6
43987	3,6	43904	1,7	44188	1,1	44054	1,5
43962	6,6	43663	4,1	43976	2,3	43695	1,9
43783	10,9	43620	14,9	43960	6,5	43139	40,6
43641	11,9	43549	20,7	43688	15,7	43094	684,7
43575	112,5	43231	319,1	43358	41,6	43094	1800,0
43286	145,3	43231	1800,0	43315	639,4		
42935	872,0			43230	833,4		
42935	1800,0			43020	1196,2		
				43020	1800,0		

Tabella 3: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

pr439 MULTI START (costo ottimo 107217)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)
116111	1,2	112354	1,4	116794	0,9	116014	1,5
114306	2,4	112228	8,2	114316	1,7	111574	3,2
112609	6,7	111791	13,9	112781	7,9	109631	6,2
112481	11,7	110439	21,9	111045	18,0	109539	198,4
110785	19,8	110324	32,9	110899	34,3	109395	901,7
110182	146,5	109949	104,7	110584	78,9	109240	1158,3
110122	168,2	109641	294,3	110424	172,5	109240	1800,0
109928	366,4	109519	544,3	109705	213,4		
109823	416,9	109270	1119,0	109659	526,0		
109352	1066,4	109220	1431,2	109508	706,3		
109335	1253,0	109220	1800,0	109508	1800,0		
109335	1800,0						

Tabella 4: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

d493 MULTI START (costo ottimo 35002)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)
36948	1,8	36912	2,4	36904	1,9	36459	1,6
36857	4,8	36895	4,9	36889	5,9	36409	37,6
36673	13,9	36821	12,1	36672	10,0	36256	91,0
36304	30,6	36771	20,6	36607	11,0	36175	1008,1
36257	191,1	36767	22,0	36393	15,8	36069	1059,5
36179	431,2	36633	31,5	36207	17,7	36069	1800,0
36157	1019,7	36588	32,3	36021	27,7		
36157	1800,0	36534	68,8	36021	1800,0		
		36441	90,3				
		36284	111,9				
		36253	278,9				
		36026	556,5				
		36026	1800,0				

Tabella 5: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

rat575 MULTI START (costo ottimo 6773)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)
7198	2,665	7284	2,226	7280	2,828	7230	2,307
7077	33,559	7208	6,366	7193	7,738	7203	4,532
7077	1800	7192	8,302	7166	9,767	7201	13,739
		7186	10,762	7161	26,771	7149	15,732
		7164	12,977	7141	46,01	7141	47,113
		7153	18,606	7100	74,421	7118	356,143
		7152	26,139	7081	293,085	7060	613,947
		7122	83,84	7079	860,752	7055	870,445
		7084	295,355	7079	1800	7055	1800
		7084	1800				

Tabella 6: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

d657 MULTI START (costo ottimo 48912)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)	Costo	Tempo(s)
53830	3,2	52927	3,7	53398	3,2	52952	3,9
52167	6,4	52766	6,1	53368	6,3	52463	13,1
51703	12,8	52504	13,5	51720	10,5	51797	21,5
51680	27,4	52179	20,4	51550	58,2	51246	54,7
51561	47,3	51656	56,0	51512	317,0	50974	1030,6
51509	236,7	51430	85,7	51419	635,9	50974	1800,0
51406	329,1	51362	752,1	51230	895,8		
51187	679,9	51109	941,9	51230	1800,0		
51032	861,0	51109	1800,0				
51032	1800,0						

Tabella 7: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

u724 MULTI START (costo ottimo 41910)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
45295	6,1	45000	6,9	45065	6,4	44836	6,5
44761	10,6	44795	11,9	44891	17,9	44189	17,4
44535	19,3	44530	28,7	44528	22,9	44182	84,6
44079	48,2	44163	41,4	44460	149,9	44161	259,9
43864	1290,6	44151	72,2	44451	244,1	44054	386,8
43864	1800,0	44136	372,2	44439	337,9	43749	990,8
		43979	603,3	43977	371,0	43711	1135,6
		43856	760,0	43971	1245,1	43711	1800,0
		43852	1420,2	43953	1303,0		
		43852	1800,0	43915	1380,6		
				43915	1800,0		

Tabella 8: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat783 MULTI START (costo ottimo 8806)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
9485	6,2	9460	7,6	9356	4,9	9401	5,0
9315	11,2	9317	19,9	9314	8,6	9399	31,7
9284	434,1	9265	395,6	9310	128,0	9376	88,8
9284	1800,0	9265	1800,0	9284	323,8	9369	127,7
				9264	1071,2	9368	136,2
				9264	1800,0	9358	175,5
						9320	262,4
						9312	377,0
						9304	559,9
						9264	785,0
						9242	1207,1
						9242	1800,0

Tabella 9: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

dsj1000 MULTI START (costo ottimo 18659688 )							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
20068535	19,4	20270696	13,6	20334609	14,3	19965496	21,9
19852900	239,8	20138131	26,8	19978177	34,8	19960654	66,8
19828957	279,7	19865783	79,5	19975814	213,2	19914229	165,1
19828957	1800,0	19852720	1579,5	19946138	469,5	19818264	306,9
		19741229	1720,5	19930787	607,3	19750658	838,6
		19741229	1800,0	19841513	718,0	19750658	1800,0
				19802820	1729,8		
				19802820	1800,0		

Tabella 10: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

## TABU SEARCH

lin318 TABU SEARCH (costo ottimo 42029)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
42935	0	43231	0	43020	0	43094	0
42935	1800	43221	1240,856	43020	1800	43094	1800
		43194	1241,153				
		43175	1241,67				
		43156	1241,962				
		43156	1800				

Tabella 11: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

pr439 TABU SEARCH (costo ottimo 107217)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
109335	0,0	109220	0,0	109508	0,0	109240	0,0
109300	2,2	109158	1,2	109499	2,7	109086	4,1
109245	3,7	108910	2,5	109302	4,2	108967	7,3
109242	4,0	108906	252,5	109070	5,0	108955	7,8
109188	4,4	108904	253,1	109070	1800,0	108887	8,3
109185	293,8	108896	254,4			108862	1016,3
109175	294,7	108890	255,0			108862	1800,0
109173	295,6	108791	256,2				
109135	297,3	108766	256,8				
109110	298,1	108745	257,4				
109097	299,0	108742	257,9				
109051	300,6	108720	312,3				
109018	301,4	108715	312,8				
109001	302,9	108713	313,5				
108981	303,7	108654	353,6				
108977	304,4	108582	354,2				
108974	305,2	108571	354,8				
108927	306,5	108569	355,4				
108888	307,2	108530	356,6				
108884	307,9	108505	357,1				
108882	308,6	108480	357,7				
108837	310,5	108459	358,2				
108816	311,0	108449	452,2				
108814	311,6	108438	452,8				
108762	656,2	108432	453,5				
108737	656,8	108413	454,7				
108716	657,3	108392	455,2				
108696	657,9	108382	455,8				
108691	658,5	108379	456,4				
108691	1800,0	108379	1800,0				

Tabella 12: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



d493 TABU SEARCH (costo ottimo 35002)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
36157	0,0	36026	0,0	36021	0,0	36069	0,0
36129	0,6	35954	0,8	36019	2,0	36049	1,0
36128	1,5	35953	1,2	36003	2,7	36033	1,7
36120	2,7	35911	3,0	35988	6,5	36021	208,5
36113	12,6	35904	5,5	35987	1409,0	36019	209,2
36091	14,7	35900	8,1	35985	1409,8	36018	401,4
36071	17,8	35897	200,7	35985	1800,0	36016	402,1
36063	18,7	35894	201,4			36006	477,1
36026	23,8	35891	202,3			36004	598,8
36024	24,7	35890	203,1			36002	599,6
36022	25,6	35885	204,5			36001	600,3
36021	320,8	35882	205,2			35998	666,1
36009	322,6	35881	256,0			35996	666,8
36002	324,5	35876	257,6			35994	667,6
35999	325,3	35873	258,4			35993	862,3
35998	326,2	35868	259,9			35986	863,1
35992	327,8	35865	260,6			35967	863,8
35991	328,6	35862	261,4			35962	864,7
35983	330,1	35859	262,1			35959	865,4
35976	330,8	35845	262,8			35956	866,2
35969	331,5	35842	325,6			35954	867,0
35966	332,2	35837	327,0			35953	867,7
35964	1082,4	35830	327,7			35953	1800,0
35955	1468,0	35827	328,5				
35955	1800,0	35826	908,5				
		35823	1016,8				
		35820	1017,6				
		35817	1018,4				
		35812	1021,4				
		35810	1080,7				
		35807	1081,3				
		35805	1082,0				
		35803	1082,7				
		35801	1209,3				
		35798	1210,0				
		35795	1210,7				
		35778	1392,0				
		35775	1392,8				
		35772	1393,6				
		35771	1394,4				
		35769	1395,8				
		35766	1396,6				
		35765	1397,3				
		35755	1588,0				
		35746	1588,8				
		35743	1589,5				
		35742	1590,2				
		35742	1800,0				

Tabella 13: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat575 TABU SEARCH (costo ottimo 6773)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
7077	0,0	7084	0	7079	0	7055	0
7075	0,8	7084	1800	7078	455,9	7049	0,8
7062	2,1			7077	546,6	7034	1,2
7055	3,2			7075	547,7	7033	4,6
7046	5,2			7074	548,9	7032	5,2
7045	729,4			7073	647,4	7021	7,0
7043	730,5			7072	648,4	7019	9,0
7042	731,6			7071	1369,5	7018	561,0
7040	733,6			7070	1370,5	7014	562,3
7038	734,5			7069	1372,5	7013	563,5
7036	735,5			7067	1373,4	7012	565,8
7034	736,5			7066	1374,4	7010	566,9
7033	737,4			7066	1800	7009	568,0
7032	917,4					7008	569,1
7031	918,5					7008	1800
7030	920,4						
7028	921,3						
7027	922,3						
7026	923,4						
7025	1006,1						
7023	1007,2						
7022	1008,3						
7021	1187,9						
7020	1188,9						
7019	1189,9						
7018	1364,6						
7017	1365,5						
7016	1366,5						
7013	1537,6						
7012	1538,9						
7011	1541,5						
7010	1542,7						
7006	1545,0						
7004	1546,0						
7003	1547,1						
7002	1548,2						
7001	1550,3						
6999	1551,2						
6997	1552,2						
6996	1553,1						
6995	1554,0						
6992	1555,0						
6992	1800,0						



Tabella 14: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



d657 TABU SEARCH (costo ottimo 48912)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
51032	0,0	51109	0,0	51230	0,0	50974	0
51020	2,3	51097	2,3	51221	2,2	50974	1800
50953	12,5	51095	7,2	51210	5,6		
50945	16,7	51092	8,8	51159	7,1		
50927	34,0	51065	10,5	51150	9,4		
50915	39,8	51046	11,4	51142	19,1		
50904	41,3	50999	15,9	51135	112,9		
50889	102,8	50995	17,8	51124	114,5		
50882	107,0	50990	22,9	51116	116,1		
50879	109,0	50990	1800,0	51111	117,6		
50876	112,9			51105	120,7		
50872	114,7			51099	122,1		
50870	116,7			51095	123,6		
50869	118,6			51054	126,2		
50868	120,5			51048	127,6		
50864	124,2			51045	128,9		
50860	125,9			51044	576,2		
50859	127,7			51037	676,6		
50794	130,9			51035	678,1		
50786	132,5			51030	679,6		
50778	134,0			51029	681,1		
50772	135,7			51028	682,6		
50769	137,4			51024	685,4		
50761	141,6			51020	686,8		
50750	142,8			51017	688,1		
50741	144,1			51015	689,4		
50734	145,3			51014	690,8		
50728	251,8			51007	693,4		
50726	253,1			51001	694,5		
50725	254,3			50995	695,8		
50716	356,6			50990	697,1		
50702	358,2			50986	698,3		
50701	359,8			50986	1800,0		
50700	361,3						
50696	364,2						
50690	365,5						
50686	366,9						
50674	369,5						
50666	370,8						
50663	371,9						
50659	814,1						
50657	815,6						
50651	818,2						
50648	819,5						
50645	1152,0						
50641	1153,2						
50639	1154,5						
50634	1155,7						
50626	1157,1						

Tabella 15: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

u724 TABU SEARCH (costo ottimo 41910)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43864	0	43915	0	43852	0	43711	0
43845	1,1	43910	10,2	43833	1,2	43692	1,3
43842	4,2	43868	12,4	43824	37,6	43680	7,8
43779	13,9	43867	274,9	43819	1458,8	43677	163,1
43745	15,1	43861	277,0	43818	1461,0	43670	296,0
43732	22,5	43860	279,1	43811	1465,2	43661	297,8
43723	138,5	43836	283,1	43803	1467,5	43659	299,6
43715	140,2	43819	285,0	43796	1469,6	43657	303,0
43711	141,9	43815	286,9	43794	1471,8	43653	304,6
43708	145,3	43813	288,8	43793	1473,8	43650	306,2
43707	277,7	43812	290,6	43789	1475,9	43649	307,7
43703	279,3	43804	294,2	43783	1479,9	43648	309,2
43700	281,0	43797	295,9	43778	1481,8	43647	584,7
43698	282,7	43795	297,6	43774	1483,7	43643	586,2
43696	284,3	43793	299,2	43768	1485,6	43639	587,8
43694	287,5	43786	305,5	43765	1487,4	43635	589,3
43691	289,1	43786	1800	43763	1489,2	43632	591,0
43682	413,5			43761	1491,2	43632	1800
43668	415,0			43760	1494,8		
43664	416,7			43759	1496,4		
43661	418,4			43752	1499,6		
43661	1800			43748	1501,1		
				43745	1502,7		
				43740	1504,1		
				43740	1800		

Tabella 16: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

rat783 TABU SEARCH (costo ottimo 8806)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
9284	0	9264	0	9265	0	9242	0
9283	59,138	9263	1,598	9257	1,352	9234	10,865
9280	66,021	9244	36,956	9256	4,652	9227	12,11
9278	73,23	9243	46,365	9255	312,909	9226	319,17
9277	80,592	9237	75,046	9254	315,584	9225	324,048
9276	187,071	9236	77,554	9249	318,202	9223	326,39
9274	188,977	9234	180,455	9248	323,082	9222	328,707
9273	190,821	9233	182,178	9246	325,655	9220	333,319
9267	336,05	9232	817,879	9245	328,17	9218	335,499
9266	337,778	9230	961,186	9244	330,618	9217	337,708
9265	339,488	9226	963,541	9242	335,162	9216	339,933
9264	341,182	9223	965,863	9241	337,401	9215	342,214
9263	342,961	9221	968,187	9240	339,715	9214	346,54
9262	807,17	9220	972,935	9239	341,989	9213	348,609
9261	809,36	9219	975,038	9238	348,087	9212	350,655
9260	811,598	9218	977,245	9237	350,126	9211	352,693
9259	813,923	9208	981,403	9236	352,109	9210	354,751
9258	818,163	9207	983,363	9235	355,929	9209	360,401
9257	820,26	9206	985,374	9234	357,908	9208	362,363
9256	822,233	9205	987,334	9233	359,712	9208	1800
9245	826,103	9204	991,107	9232	361,496		
9244	827,873	9203	993,077	9231	363,321		
9243	829,747	9202	994,903	9230	365,161		
9242	831,591	9201	996,771	9229	530,5		
9241	833,401	9199	1122,157	9228	532,25		
9240	835,158	9196	1124,374	9227	677,756		
9239	836,958	9195	1126,53	9226	679,984		
9239	1800	9194	1128,633	9225	682,174		
		9192	1132,679	9223	686,373		
		9191	1134,628	9221	688,342		
		9190	1136,655	9220	690,422		
		9189	1142,332	9219	692,373		
		9188	1144,31	9218	694,368		
		9187	1146,111	9217	698,042		
		9186	1147,922	9216	699,793		
		9186	1800	9215	701,541		
				9214	703,321		
				9214	1800		

Tabella 17: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



dsj1000 TABU SEARCH (costo ottimo 18659688)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
19828957	0,0	19802820	0,0	19741229	0,0	19750658	0,0
19815585	10,0	19799454	3,8	19737657	3,7	19747109	5,2
19815108	28,3	19792630	11,3	19734229	44,4	19744713	14,8
19814543	37,8	19784187	29,1	19721723	47,1	19741026	31,7
19806185	45,7	19781812	33,8	19721431	117,4	19731537	65,7
19802768	57,2	19779585	52,1	19717868	121,6	19726726	69,1
19793270	60,2	19779059	517,1	19716480	543,5	19723170	72,5
19763782	204,9	19777739	521,6	19714929	547,6	19703317	82,7
19735109	209,9	19775888	530,1	19714904	551,7	19703270	86,1
19731456	215,0	19774017	534,2	19713214	559,4	19700268	100,9
19731152	224,7	19772302	538,3	19711789	563,2	19700268	1800,0
19730940	229,5	19771185	542,4	19711668	566,9		
19730819	234,2	19769005	546,5	19711574	570,4		
19730725	238,9	19768426	550,5	19710059	577,4		
19730702	243,7	19766651	554,6	19709646	580,8		
19730312	252,8	19766604	558,6	19709256	584,2		
19729922	257,1	19764803	566,3	19708959	587,7		
19729558	265,7	19763166	570,1	19708912	591,2		
19729161	269,7	19762155	574,0	19708322	597,8		
19728767	273,9	19761116	577,6	19707759	601,0		
19728650	277,9	19760999	581,2	19707258	604,2		
19728023	285,7	19760914	584,8	19706871	607,4		
19727374	289,5	19759230	591,8	19650420	924,4		
19726873	293,2	19758836	595,2	19649525	1107,9		
19726194	300,5	19758499	598,6	19648888	1112,5		
19725501	303,8	19758269	601,9	19647427	1121,0		
19725050	307,2	19758172	605,3	19645696	1125,2		
19724943	310,6	19757525	611,7	19644087	1129,4		
19724021	314,0	19756962	614,6	19642735	1133,5		
19723280	320,3	19756504	617,5	19642341	1137,6		
19722482	323,4	19756107	620,5	19641226	1145,5		
19721767	326,4	19755810	623,5	19640768	1149,4		
19720766	329,6	19755465	1165,3	19640647	1153,3		
19720177	332,8	19754597	1168,3	19640553	1157,3		
19717160	464,4	19754261	1171,4	19638988	1164,6		
19716116	469,6	19754261	1800,0	19638598	1168,0		
19711421	474,7			19638301	1171,4		
19711406	480,0			19638103	1174,9		
19710425	516,8			19637628	1181,5		
19709515	521,8			19637065	1184,5		
19708784	526,9			19636652	1187,6		
19708328	536,9			19636409	1190,6		
19707525	541,7			19636292	1193,6		
19706950	551,1			19636035	1712,2		
19705597	555,8			19635837	1715,8		
19704368	560,3			19635743	1719,4		
19703208	565,0			19634106	1726,5		
19702524	573,5			19633693	1730,0		
19701524	577,6			19633306	1733,5		

Tabella 18: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

**GENETICO**



lin318 GENETICO GEN#20 (costo ottimo 42029)	
Costo	Tempo(s)
67669	0,1
46016	0,9
45290	2,1
44701	8,7
44481	18,2
44027	27,5
43660	38,3
43570	65,2
43488	65,7
43457	455,4
43347	457,1
43157	458,2
43120	470,4
43053	476,9
43044	534,1
42960	1674,0
42960	1800

Tabella 19: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

pr439 GENETICO GEN#20 (costo ottimo 107217)	
Costo	Tempo(s)
178844	0,2
172081	0,2
111632	5,5
111364	150,2
111301	203,1
111265	207,7
111134	235,9
110839	259,0
110667	309,8
110623	875,3
110508	1319,7
110508	1800

Tabella 20: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

d493 GENETICO GEN#20 (costo ottimo 35002)	
Costo	Tempo(s)
50521	0,3
37341	5,2
37107	8,0
36996	17,9
36936	32,9
36855	54,3
36810	115,7
36780	116,8
36771	117,6
36751	130,3
36683	527,4
36525	1176,5
36525	1800

Tabella 21: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

rat575 GENETICO GEN#20 (costo ottimo 6773)	
Costo	Tempo(s)
10225	0,3
9927	0,3
9858	0,3
9613	0,3
7411	8,1
7332	14,6
7317	112,0
7268	122,3
7263	277,5
7256	278,9
7238	283,3
7228	283,9
7206	338,7
7124	1175,1
7124	1800

Tabella 22: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

d657 GENETICO GEN#20 (costo ottimo 48912)	
Costo	Tempo(s)
67669	0,1
46016	0,9
45290	2,1
44701	8,7
44481	18,2
44027	27,5
43660	38,3
43570	65,2
43488	65,7
43457	455,4
43347	457,1
43157	458,2
43120	470,4
43053	476,9
43044	534,1
42960	1674,0
42960	1800

Tabella 23: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

u724 GENETICO GEN#20 (costo ottimo 41910)	
Costo	Tempo(s)
64140	0,5
45937	18,8
44988	58,6
44351	116,3
44279	1114,6
44279	1800,0

Tabella 24: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

rat783 GENETICO GEN#20 (costo ottimo 8806)	
Costo	Tempo(s)
13519	0,4
13180	0,5
9672	12,9
9562	30,1
9552	57,6
9536	76,5
9516	79,8
9485	89,2
9482	101,4
9426	125,2
9403	656,9
9386	800,9
9380	806,2
9380	1800

Tabella 25: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

dsj1000 GENETICO GEN#20 (costo ottimo 18659688)	
Costo	Tempo(s)
28530708	0,7
28339022	0,9
27940942	1,5
27704229	1,7
20180860	55,9
19914677	113,8
19899518	215,1
19886374	950,6
19885127	1241,2
19885127	1800,0

Tabella 26: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

# VNS

lin318 VNS (costo ottimo 42029)							
Thread1				Thread2			
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
44328	0,0	45631	0,0	45189	0,0	44805	0,0
44293	0,1	44369	0,2	44674	0,2	44459	1,7
44242	0,1	44163	0,3	44375	0,5	44411	1,9
44128	3,0	43344	0,7	44365	3,1	44251	2,1
43691	3,3	43320	286,4	43945	7,2	44128	2,7
43675	3,4	43310	637,1	43901	7,3	44034	2,9
43639	4,8	43233	637,2	43807	7,7	43968	3,0
43591	6,1	43146	914,6	43685	27,2	43655	3,1
43583	25,0	43142	914,6	43574	27,3	43579	3,2
43369	25,0	43142	1800	43447	81,3	43578	38,1
43282	95,7			43392	81,7	43524	61,8
43280	937,2			43374	124,6	43316	289,0
43163	955,1			43230	137,8	43211	790,5
43052	1483,7			43168	771,2	42984	1560,1
43052	1800			43129	1385,1	42984	1800
				43129	1800		

Tabella 27: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

pr439 VNS (costo ottimo 107217)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
115162	0,0	115118	0,0	118329	0,0	116833	0,0
114445	0,1	113644	0,6	117757	0,3	116375	1,9
113959	0,9	113100	6,5	117749	0,5	114772	2,9
113010	1,3	112843	7,2	117469	2,8	114236	6,1
112977	3,1	112083	37,8	116925	4,2	114226	8,2
112431	8,9	111462	66,1	113697	5,2	112902	11,1
112400	9,4	111095	67,5	113573	5,3	112328	18,4
111864	19,0	111013	67,9	113007	15,0	111610	18,8
111378	19,2	110754	203,2	112911	15,5	111182	79,3
111219	19,3	110613	379,3	112512	17,0	110818	139,9
111153	26,1	110219	490,3	110844	18,4	110367	293,0
110824	26,5	109775	568,4	110598	46,9	110297	391,1
110700	26,7	109764	1219,9	110580	84,4	110229	976,1
109214	28,0	109167	1220,5	110249	192,0	110138	976,3
108691	1343,0	109167	1800	110132	368,4	109942	1468,1
108691	1800			109930	994,4	109942	1800
				109563	1086,7		
				109287	1173,8		
				109287	1800		

Tabella 28: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

d493 VNS (costo ottimo 35002)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
37367	0,0	37017	0,0	37712	0,0	37356	0,0
37302	4,0	36771	14,6	37561	0,8	37256	0,3
37210	8,3	36736	66,1	37519	1,5	37062	1,0
37114	8,8	36593	82,2	37328	2,3	37022	105,9
37065	10,1	36495	522,8	37285	2,6	36960	106,5
36888	12,1	36344	1170,5	37263	5,4	36902	107,2
36833	20,5	36344	1800	37177	5,9	36784	107,5
36546	27,8			36894	10,7	36579	108,0
36532	28,2			36813	14,2	36530	108,8
36452	31,1			36738	94,7	36519	113,5
36421	537,5			36550	97,0	36388	136,3
36276	537,9			36517	124,4	36296	136,8
36276	1800			36484	125,1	36296	1800
				36431	144,0		
				36398	144,6		
				36260	273,5		
				36260	1800		

Tabella 29: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

rat575 VNS (costo ottimo 6773)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
7239	0,0	7342	0,0	7318	0,0	7369	0,0
7233	34,4	7312	0,8	7294	0,4	7366	0,4
7232	57,6	7287	15,7	7283	5,8	7316	2,0
7229	58,2	7276	16,5	7277	6,1	7270	3,0
7226	71,9	7250	18,6	7274	8,4	7269	4,0
7203	130,5	7248	19,1	7265	20,9	7240	6,5
7192	132,2	7188	127,2	7231	26,4	7229	114,3
7171	300,1	7186	313,7	7200	80,6	7207	115,6
7171	1800,0	7173	381,3	7189	511,7	7181	520,9
		7173	1800,0	7180	833,6	7157	992,6
				7176	869,9	7157	1800,0
				7174	871,4		
				7161	872,1		
				7129	873,5		
				7129	1800,0		

Tabella 30: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

d657 VNS (costo ottimo 48912)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
52642	0,0	52724	0,0	52852	0,0	53319	0,0
52583	3,1	52629	1,6	52484	1,1	53304	0,8
52443	17,5	52283	29,2	52157	3,1	53274	1,7
52271	19,7	52199	88,0	52053	100,1	53141	2,4
52145	42,1	52166	89,0	52035	100,6	52650	3,0
52072	46,4	52154	120,9	51824	145,5	52037	3,4
52058	51,0	51923	126,3	51789	147,6	51922	136,8
51944	63,9	51874	152,2	51785	148,4	51667	147,5
51767	70,8	51409	153,1	51733	420,5	51352	149,8
51669	214,3	51109	177,2	51703	421,5	51352	1800,0
51648	319,5	50997	179,8	51620	422,3		
51580	325,7	50872	180,4	51602	425,0		
51462	326,8	50768	181,6	51550	429,8		
51051	388,2	50768	1800,0	51405	431,6		
51051	1800,0			51235	433,1		
				51224	885,6		
				51194	889,4		
				50809	1194,6		
				50809	1800,0		

Tabella 31: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

u724 VNS (costo ottimo 41910)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
45228	0,001	44729	0,001	44683	0,001	45146	0,001
45054	41,522	44669	3,564	44505	54,886	45022	0,69
44987	47,077	44636	28,622	44441	56,136	45001	25,236
44957	183,277	44496	112,255	44390	834,913	44965	26,903
44788	211,512	44459	1083,77	44316	1118,98	44929	27,838
44786	415,122	44331	1284,459	44316	1800,0	44846	79,558
44500	438,924	44196	1708,761			44763	117,202
44381	479,237	44196	1800,0			44681	124,146
44246	483,172					44567	345,671
44201	744,046					44556	458,118
44201	1800,0					44416	459,21
						44270	1151,915
						44202	1152,22
						44202	1800,0

Tabella 32: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat783 VNS (costo ottimo 8806)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
9399	0,0	9432	0,0	9606	0,0	9459	0,0
9399	1800	9418	24,7	9596	0,7	9453	53,0
		9393	26,2	9578	2,5	9452	639,3
		9371	117,1	9563	43,9	9448	911,3
		9366	118,0	9562	44,7	9439	912,4
		9356	120,2	9528	47,6	9426	1000,8
		9356	1800	9523	63,5	9402	1660,8
				9513	78,4	9394	1677,7
				9504	84,9	9384	1683,0
				9495	95,3	9384	1800
				9488	139,0		
				9477	140,1		
				9466	163,8		
				9445	164,0		
				9443	168,9		
				9421	169,6		
				9420	390,7		
				9419	391,7		
				9415	392,4		
				9406	976,6		
				9406	1800		

Tabella 33: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

dsj1000 VNS (costo ottimo 18659688)							
Thread1		Thread2		Thread3		Thread4	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
20388124	0,0	19933316	0,0	20179476	0,0	20192173	0,0
20350385	0,9	19901388	3,9	20041915	8,3	20190252	17,1
20315711	29,2	19819155	23,8	20037603	822,0	20161855	136,0
20273258	82,7	19818600	849,1	20010375	889,4	20068673	149,9
20207005	90,0	19811979	850,2	19968630	890,5	20063392	158,9
20202303	92,6	19811979	1800,0	19930678	1750,2	20054192	159,4
20196943	95,1			19918893	1755,2	19926458	160,8
20059067	113,1			19918893	1800,0	19856223	638,3
20054941	119,3					19834387	1649,3
20023983	150,4					19834387	1800,0
19976516	314,6						
19892262	324,5						
19861128	1082,7						
19861128	1800,0						

Tabella 34: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

## HARD FIXING & LOCAL BRANCH

lin318 - HF & LB - PT1 - (costo ottimo 42029)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43231	0,0	43231	0,0	42935	0,0	42935	0,0
43198	1,7	43132	8,1	42916	1,0	42813	7,2
43009	2,1	43040	15,7	42820	2,8	42721	16,2
42976	2,6	42957	24,3	42803	6,0	42678	25,8
42951	3,6	42875	30,6	42749	6,5	42638	33,7
42939	4,8	42832	39,7	42746	8,7	42601	46,8
42756	11,3	42792	48,1	42724	13,3	42568	55,8
42743	19,2	42743	59,2	42591	14,9	42540	64,4
42724	24,2	42710	70,5	42575	22,1	42524	76,4
42672	25,7	42677	81,9	42483	23,0	42512	89,5
42658	39,6	42644	94,1	42436	27,8	42506	102,5
42592	51,1	42628	107,4	42397	39,3	42425	147,1
42529	163,7	42530	144,7	42357	62,7	42371	189,0
42525	171,9	42455	179,9	42351	72,7	42317	225,9
42480	180,2	42401	234,4	42329	138,9	42293	282,4
42457	197,5	42347	269,1	42294	143,8	42270	327,9
42395	203,5	42314	322,3	42248	187,2	42248	705,7
42381	222,3	42302	374,4	42237	1415,6	42248	1800
42380	231,7	42273	415,9	42237	1800		
42377	239,7	42265	470,7				
42277	249,8	42178	512,7				
42218	254,6	42165	559,1				
42183	298,6	42143	963,8				
42143	327,5	42143	1800				
42143	1800						

Tabella 35: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

lin318 - HF & LB - PT2 - (costo ottimo 42029)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43094	0,0	43094	0,0	43020	0,0	43020	0,0
43075	0,2	43034	9,8	42704	1,9	42839	7,9
43029	9,8	42987	21,1	42655	6,6	42728	15,0
42964	11,1	42939	30,3	42536	10,1	42691	26,0
42924	19,5	42899	40,1	42512	11,8	42651	37,1
42864	25,0	42850	48,4	42511	18,5	42602	47,7
42815	27,8	42813	58,1	42452	28,6	42573	58,7
42801	33,3	42776	72,2	42340	36,5	42545	68,5
42759	33,9	42743	83,2	42297	44,0	42529	79,3
42707	37,5	42715	94,6	42271	46,3	42518	90,2
42667	44,5	42687	107,5	42234	50,6	42516	102,4
42650	53,3	42671	118,8	42207	68,8	42413	139,9
42640	63,3	42655	130,3	42193	294,9	42311	167,0
42551	68,4	42645	143,8	42160	304,4	42237	195,4
42551	1800	42564	208,6	42128	336,8	42186	219,4
		42507	259,3	42107	436,4	42135	240,3
		42461	314,1	42107	1800	42112	277,3
		42451	395,8			42104	309,7
		42451	1800			42091	341,5
						42091	1800

Tabella 36: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

pr439 - HF & LB - PT1 - (costo ottimo 107217)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
109508	0,0	109508	0,0	109240	0,0	109240	0,0
109354	1,0	109276	32,0	109206	0,3	108998	54,1
109309	4,3	109061	59,0	109038	0,6	108815	93,7
109259	6,4	108867	90,3	108747	3,8	108647	134,1
109236	9,6	108718	124,4	108722	25,8	108540	172,5
109195	10,2	108611	159,9	108496	29,1	108495	220,6
109057	11,0	108510	196,9	108455	52,2	108452	265,3
108693	11,7	108419	234,4	108400	59,1	108427	308,6
108673	12,9	108334	269,1	108361	71,2	108404	356,0
108664	22,6	108283	306,3	108314	105,1	108386	403,0
108573	24,2	108251	345,6	108312	122,6	108370	454,8
108555	37,1	108204	386,5	108214	152,4	108360	505,9
108462	48,2	108178	429,3	108173	179,4	108334	557,1
108411	61,2	108152	476,5	108150	188,4	108202	774,1
108120	67,7	108129	523,7	108147	206,6	108076	945,7
108085	77,7	108120	571,0	108045	230,2	108009	1087,1
108055	83,4	107935	731,8	108019	300,2	107950	1229,6
108024	92,2	107794	818,5	107967	342,9	107903	1357,0
107985	97,3	107764	958,1	107961	864,3	107866	1529,1
107884	106,6	107759	1108,2	107960	932,5	107753	1712,5
107807	128,4	107757	1231,1	107887	990,1	107753	1800,0
107807	1800,0	107673	1327,4	107867	1052,7		
		107587	1459,4	107862	1241,9		
		107587	1800,0	107471	1250,0		
				107400	1291,8		
				107400	1800,0		

Tabella 37: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

pr439 - HF & LB - PT2 - (costo ottimo 107217)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43231	0,0	43231	0,0	42935	0,0	42935	0,0
43198	1,7	43132	8,1	42916	1,0	42813	7,2
43009	2,1	43040	15,7	42820	2,8	42721	16,2
42976	2,6	42957	24,3	42803	6,0	42678	25,8
42951	3,6	42875	30,6	42749	6,5	42638	33,7
42939	4,8	42832	39,7	42746	8,7	42601	46,8
42756	11,3	42792	48,1	42724	13,3	42568	55,8
42743	19,2	42743	59,2	42591	14,9	42540	64,4
42724	24,2	42710	70,5	42575	22,1	42524	76,4
42672	25,7	42677	81,9	42483	23,0	42512	89,5
42658	39,6	42644	94,1	42436	27,8	42506	102,5
42592	51,1	42628	107,4	42397	39,3	42425	147,1
42529	163,7	42530	144,7	42357	62,7	42371	189,0
42525	171,9	42455	179,9	42351	72,7	42317	225,9
42480	180,2	42401	234,4	42329	138,9	42293	282,4
42457	197,5	42347	269,1	42294	143,8	42270	327,9
42395	203,5	42314	322,3	42248	187,2	42248	705,7
42381	222,3	42302	374,4	42237	1415,6	42248	1800
42380	231,7	42273	415,9	42237	1800		
42377	239,7	42265	470,7				
42277	249,8	42178	512,7				
42218	254,6	42165	559,1				
42183	298,6	42143	963,8				
42143	327,5	42143	1800				
42143	1800						

Tabella 38: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



d493 - HF & LB - PT1 - (costo ottimo 35002)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
36157	0,0	36157	0,0	36026	0,0	36026	0,0
36092	3,5	36107	34,4	36017	1,7	35946	18,3
36085	6,8	36070	75,7	36009	3,5	35873	36,2
35951	8,2	36026	111,1	36003	6,7	35828	66,5
35867	10,6	35997	148,0	36002	7,8	35788	104,6
35853	12,0	35969	194,2	35983	9,3	35763	142,2
35817	13,1	35938	236,3	35976	11,9	35702	171,2
35788	16,8	35919	288,7	35961	13,4	35681	208,7
35785	18,9	35900	335,0	35916	14,6	35662	256,7
35780	27,5	35886	384,6	35895	15,7	35644	308,5
35768	28,6	35873	429,1	35809	19,3	35629	358,2
35764	31,3	35861	479,0	35769	23,2	35617	407,3
35704	32,6	35849	531,2	35742	27,0	35609	455,9
35701	34,3	35838	573,9	35711	27,6	35601	508,1
35682	38,5	35828	625,5	35708	30,7	35594	561,6
35669	40,8	35819	679,7	35702	33,3	35588	622,3
35654	48,0	35805	734,0	35687	37,6	35583	675,3
35628	58,2	35796	796,9	35668	40,9	35579	733,7
35607	93,5	35787	849,7	35656	50,4	35560	786,4
35586	106,6	35780	914,9	35609	51,8	35557	838,9
35563	137,9	35775	972,3	35591	52,9	35554	891,5
35559	153,7	35771	1042,4	35589	60,9	35552	946,8
35557	162,2	35768	1112,0	35568	75,2	35488	1078,6
35479	175,0	35765	1180,5	35564	85,5	35411	1142,1
35440	190,2	35752	1247,0	35555	94,1	35385	1314,6
35437	198,2	35749	1314,5	35548	102,0	35364	1471,6
35361	208,2	35747	1384,9	35531	136,0	35344	1631,3
35358	219,1	35645	1525,9	35518	148,1	35299	1714,8
35312	241,5	35605	1717,0	35446	165,4	35282	1800
35272	252,4	35605	1800	35434	171,3		
35260	336,8			35423	184,6		
35248	354,0			35373	191,3		
35244	369,7			35369	232,0		
35149	389,5			35357	246,1		
35090	416,0			35356	258,6		
35084	429,7			35346	284,5		
35053	539,0			35343	305,7		
35047	592,6			35338	321,4		
35046	605,7			35335	338,7		
35043	624,5			35329	361,1		
35030	1128,2			35274	374,2		
35028	1189,2			35191	381,8		
35028	1800			35189	395,1		
				35185	413,1		
				35152	448,8		
				35145	454,1		
				35133	474,8		
				35133	1800		

Tabella 39: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti





d493 - HF & LB - PT2 - (costo ottimo 35002)

Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
36021	0,0	36021	0,0	36069	0,0	36069	0,0
35991	1,0	35928	14,9	36059	3,8	36000	25,9
35948	6,7	35857	47,2	36054	9,6	35960	63,9
35935	8,8	35808	76,5	36030	14,6	35934	98,9
35914	10,0	35765	105,8	36002	17,4	35906	133,2
35898	19,9	35736	142,3	35962	19,8	35885	178,1
35869	20,8	35712	178,4	35934	22,3	35866	225,2
35868	22,3	35691	215,1	35931	26,5	35847	270,1
35725	33,7	35671	260,2	35902	30,3	35828	314,8
35690	36,3	35652	303,5	35880	34,7	35809	358,5
35680	44,0	35633	347,5	35850	36,5	35791	401,8
35653	58,7	35617	393,7	35824	40,9	35775	443,2
35630	65,8	35601	439,5	35776	43,8	35764	495,3
35584	72,9	35587	484,2	35774	46,1	35757	537,6
35567	77,7	35574	528,2	35772	51,0	35752	614,4
35541	86,1	35562	570,4	35718	51,7	35748	692,6
35535	94,3	35551	614,9	35697	56,4	35745	764,9
35491	104,0	35541	657,9	35682	58,6	35744	836,7
35488	110,7	35476	686,0	35621	65,4	35706	1060,6
35449	113,1	35468	726,2	35618	72,7	35677	1262,0
35417	118,3	35460	770,3	35614	76,4	35658	1451,5
35416	125,5	35452	813,0	35609	92,0	35645	1686,9
35397	130,7	35447	852,6	35606	99,6	35645	1800
35379	133,2	35413	876,5	35600	104,6		
35328	141,0	35409	913,9	35597	112,7		
35287	146,0	35406	956,2	35596	123,1		
35283	147,3	35404	1011,7	35562	129,0		
35276	155,7	35402	1052,8	35537	137,6		
35269	158,7	35401	1102,6	35481	151,6		
35261	170,7	35392	1356,3	35473	160,4		
35231	175,7	35384	1690,5	35468	165,9		
35230	204,2	35378	1800	35456	173,5		
35219	215,1			35451	181,3		
35213	246,5			35442	215,7		
35212	277,9			35433	249,3		
35195	292,4			35430	259,5		
35194	305,5			35400	271,3		
35155	312,7			35378	289,9		
35149	329,1			35365	330,7		
35131	341,7			35345	335,5		
35128	351,0			35340	380,9		
35127	386,1			35338	412,9		
35121	396,7			35331	427,5		
35121	1800			35315	452,8		
				35305	473,6		
				35276	498,6		
				35255	520,6		
				35224	565,2		
				35219	592,1		
				35218	618,5		

Tabella 40: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat575 - HF & LB - PT1 - (costo ottimo 6773)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
7077	0,0	7077	0,0	7084	0,0	7084	0,0
7072	2,7	7063	25,2	7080	1,7	7070	21,4
7063	3,8	7050	52,5	7077	7,5	7057	45,9
7061	8,4	7039	76,1	7075	8,9	7046	67,1
7055	9,7	7028	105,9	7068	10,5	7035	93,6
7050	11,9	7019	134,0	7047	13,6	7025	122,4
7040	15,2	7011	169,3	7044	17,6	7016	150,7
7031	21,3	7003	199,4	7027	19,1	7009	184,9
7030	22,9	6995	233,0	7016	20,6	7003	216,7
7025	23,8	6988	267,7	7006	25,8	6998	250,9
7011	25,6	6981	305,8	7000	29,3	6993	279,1
7004	27,0	6975	342,2	6988	32,6	6988	307,8
7003	28,9	6969	372,5	6984	37,9	6983	331,0
6978	33,7	6963	403,1	6959	39,1	6978	367,1
6974	36,0	6958	430,1	6955	42,5	6974	399,4
6973	39,2	6954	458,5	6951	43,4	6970	428,5
6971	41,7	6950	489,1	6946	55,9	6967	458,9
6970	42,5	6946	515,9	6938	63,6	6965	488,4
6966	43,9	6939	544,3	6933	67,5	6963	523,4
6965	45,5	6935	572,8	6931	73,8	6961	555,9
6958	59,3	6932	605,5	6925	77,4	6959	586,1
6940	70,8	6929	635,2	6923	88,1	6957	616,9
6931	76,2	6926	662,9	6909	96,7	6956	651,7
6923	82,9	6924	694,4	6908	102,8	6955	679,6
6914	85,9	6922	732,9	6904	104,8	6954	716,7
6906	87,9	6920	772,7	6894	116,0	6953	743,8
6902	91,8	6916	806,2	6893	121,8	6934	785,3
6899	96,1	6913	836,9	6891	131,9	6924	811,8
6891	126,1	6912	872,0	6889	140,1	6915	841,5
6890	141,8	6911	908,5	6886	142,8	6907	891,8
6889	153,2	6910	942,9	6884	144,6	6901	941,9
6880	164,6	6909	980,1	6873	176,3	6894	966,5
6876	177,6	6908	1021,8	6872	182,9	6889	1024,8
6875	278,6	6903	1102,9	6858	204,2	6885	1073,3
6872	307,2	6900	1155,0	6853	209,3	6882	1112,6
6869	320,3	6894	1198,4	6851	212,9	6879	1181,9
6852	334,1	6891	1274,8	6849	245,3	6877	1250,6
6839	370,4	6888	1335,1	6842	250,1	6872	1488,3
6837	378,1	6885	1423,7	6841	261,0	6868	1675,2
6828	382,5	6881	1481,2	6835	267,0	6860	1748,3
6826	391,0	6879	1579,7	6834	272,1	6858	1800
6817	395,1	6878	1670,1	6830	289,7		
6814	557,7	6876	1738,7	6828	301,6		
6813	641,4	6876	1800,189	6822	350,8		
6811	676,1			6819	385,2		
6810	731,6			6818	460,0		
6809	771,5			6814	486,1		
6806	795,7			6813	550,8		
6804	1066,0			6808	582,0		

Tabella 41: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat575 - HF & LB - PT2 - (costo ottimo 6773)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
7079	0,0	7079	0,0	7055	0,0	7055	0,0
7069	1,5	7059	5,9	7048	0,8	7041	28,4
7056	5,5	7043	25,2	7043	2,0	7029	56,8
7048	6,9	7034	52,1	7042	5,9	7017	81,2
7044	9,5	7026	73,7	7021	12,2	7009	122,1
7043	12,0	7018	98,2	7008	15,9	7001	163,2
7041	14,1	7010	126,0	7002	19,5	6994	192,2
7040	16,8	7003	158,2	6971	27,9	6988	228,2
7034	19,4	6997	190,2	6965	30,6	6982	263,5
7019	22,7	6993	223,3	6956	34,0	6977	302,0
7013	26,2	6989	253,1	6948	37,2	6972	345,4
7012	28,4	6985	282,9	6941	44,5	6967	381,8
7008	31,3	6981	315,5	6939	49,3	6962	413,7
7000	34,6	6977	348,9	6938	52,2	6958	450,3
6989	45,6	6975	381,9	6937	63,7	6954	491,9
6981	52,0	6973	414,4	6925	68,4	6950	521,8
6946	80,2	6971	452,4	6917	70,5	6946	554,3
6925	91,4	6967	491,3	6907	72,2	6943	584,1
6920	115,7	6965	526,4	6893	82,0	6934	625,1
6914	130,7	6963	564,5	6891	98,9	6922	660,7
6911	141,8	6961	602,2	6889	100,2	6919	695,5
6910	158,1	6959	647,5	6878	103,3	6916	727,8
6907	162,5	6958	684,7	6876	107,1	6912	768,2
6895	178,2	6957	727,4	6873	111,3	6910	802,7
6893	188,9	6956	768,0	6871	116,9	6908	837,4
6889	194,6	6955	807,1	6865	124,1	6903	866,2
6883	201,2	6945	884,6	6852	127,7	6901	903,5
6870	210,4	6937	925,3	6851	135,0	6899	937,9
6868	217,3	6926	970,3	6848	164,3	6898	974,5
6863	220,1	6918	1040,9	6845	177,1	6897	1010,9
6862	225,5	6910	1096,0	6826	181,2	6896	1044,9
6853	233,5	6902	1130,3	6820	190,9	6895	1086,0
6844	248,1	6897	1191,2	6815	200,4	6894	1122,1
6843	278,0	6892	1250,1	6813	235,0	6883	1186,1
6837	295,1	6887	1305,3	6809	249,0	6872	1212,0
6836	379,5	6884	1368,8	6804	259,2	6861	1238,1
6831	410,1	6876	1409,0	6802	273,2	6851	1260,2
6830	524,8	6873	1472,0	6801	595,5	6834	1275,9
6826	841,5	6870	1552,8	6800	622,8	6826	1307,8
6819	859,2	6864	1601,0	6796	679,5	6824	1369,5
6815	900,7	6863	1694,6	6794	715,4	6820	1570,6
6799	931,3	6863	1800	6792	738,4	6816	1659,3
6797	944,6	192		6791	765,5	6814	1765,1
6796	1353,9			6788	971,4	6814	1800
6792	1430,2			6787	1012,6		
6791	1762,1			6786	1054,2		
6791	1800			6784	1078,4		
				6783	1104,9		
				6782	1135,6		



Tabella 42: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



lin318 - HF & LB - PT2 - (costo ottimo 42029)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
51032	0,0	51032	0,0	51230	0,0	51230	0,0
51008	10,8	50911	40,6	51218	24,6	51101	39,4
51002	13,4	50804	76,9	51131	43,1	50983	99,1
50979	16,7	50702	103,2	51040	56,8	50904	161,7
50916	18,5	50635	154,4	51028	59,2	50832	227,6
50902	26,3	50575	195,3	50952	61,7	50763	313,9
50889	32,6	50515	230,5	50914	65,1	50696	380,4
50884	38,3	50463	284,2	50913	76,2	50631	455,6
50766	40,2	50416	332,2	50848	83,1	50580	531,5
50761	42,9	50371	380,6	50785	88,7	50538	606,7
50723	46,3	50340	442,5	50778	90,3	50460	663,3
50610	49,2	50312	514,6	50742	95,5	50418	739,6
50601	58,3	50290	593,8	50722	101,2	50381	804,6
50574	61,2	50269	663,7	50701	105,5	50345	870,4
50475	72,4	50250	754,7	50680	109,5	50314	943,4
50452	81,7	50231	843,5	50592	111,4	50284	1005,3
50426	87,6	50201	908,6	50491	115,9	50254	1074,6
50418	88,7	50183	993,0	50469	121,4	50228	1158,1
50387	93,7	50165	1061,1	50354	131,7	50203	1255,8
50325	98,2	50138	1151,0	50323	134,3	50179	1330,3
50307	105,3	50121	1238,1	50297	150,2	50158	1417,5
50300	108,9	50107	1325,2	50276	153,3	50123	1494,7
50268	113,0	50093	1421,7	50275	155,1	50102	1572,0
50222	118,2	50081	1507,1	50193	160,0	50084	1683,1
50194	123,2	50056	1578,7	50192	167,4	50067	1766,2
50133	132,7	50004	1649,6	50191	174,3	50051	1800
50063	146,9	49977	1724,2	50189	177,7		
50054	162,1	49969	1800	50184	181,1		
50037	167,5			50181	183,1		
49954	189,3			50177	190,3		
49934	204,8			50173	201,2		
49930	208,8			50149	215,3		
49914	212,3			50142	217,9		
49897	243,9			50137	236,2		
49869	269,2			50132	327,3		
49863	299,4			50111	345,2		
49791	336,5			50107	404,6		
49766	343,5			50094	433,0		
49753	362,9			49953	453,6		
49697	380,6			49877	470,3		
49656	391,3			49804	484,7		
49635	430,2			49752	504,2		
49630	438,4			49720	550,3		
49606	462,9			49648	567,4		
49595	498,8			49640	591,0		
49435	524,3			49542	624,4		
49410	557,8			49501	641,6		
49406	621,9			49465	662,8		

Tabella 43: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



lin318 - HARD FIXING E LOCAL BRANCH - PT1 - (costo ottimo 42029)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
51109	0,0	51109	0,0	50974	0,0	50974	0,0
51095	1,2	51033	66,1	50953	8,6	50889	45,9
51088	2,9	50981	132,7	50821	12,3	50807	100,3
51082	4,1	50934	212,1	50781	15,7	50708	143,6
51050	5,3	50891	293,2	50733	21,5	50627	201,4
51046	8,2	50853	353,6	50538	26,6	50572	254,0
51003	9,7	50815	411,7	50464	31,5	50521	340,5
50997	10,8	50777	490,0	50394	36,3	50482	402,1
50983	15,9	50744	564,9	50355	38,3	50448	476,1
50936	20,6	50706	641,6	50346	47,9	50416	535,6
50848	27,9	50674	715,7	50327	82,9	50388	598,2
50816	32,8	50646	806,4	50303	93,4	50365	715,1
50713	35,7	50619	873,9	50276	116,3	50344	834,1
50604	49,9	50594	942,7	50037	127,6	50323	934,3
50577	52,3	50571	1017,2	50027	132,8	50302	1036,1
50534	54,9	50549	1088,8	49900	145,5	50281	1134,6
50509	61,8	50528	1164,4	49853	154,9	50261	1240,5
50376	65,3	50509	1246,2	49806	159,6	50230	1318,5
50338	69,9	50492	1320,4	49759	166,3	50210	1403,9
50299	83,8	50451	1392,0	49689	171,8	50191	1493,4
50287	86,5	50432	1477,2	49679	215,9	50177	1586,8
50286	87,7	50415	1550,7	49672	244,2	50171	1712,8
50270	90,4	50398	1615,3	49619	251,5	50170	1800
50224	93,6	50383	1686,7	49600	259,1		
50202	96,5	50365	1746,9	49533	292,7		
50189	102,7	50352	1800	49478	302,7		
50151	106,8			49433	323,0		
50142	110,9			49432	328,2		
50137	113,0			49423	360,8		
50122	122,9			49412	374,3		
50121	129,3			49372	422,6		
50116	133,2			49332	438,0		
49992	140,0			49298	469,3		
49971	145,5			49283	501,4		
49952	149,7			49268	541,4		
49926	169,4			49261	805,8		
49881	189,0			49256	824,5		
49762	201,5			49194	853,8		
49736	218,4			49185	878,7		
49717	234,9			49178	927,9		
49708	241,5			49159	1013,8		
49691	272,6			49140	1052,2		
49684	275,3			49120	1271,2		
49648	281,1			49077	1665,4		
49641	283,4			49075	1755,6		
49613	302,1			49075	1800		
49601	313,6						
49578	317,6						
49546	332,6						

Tabella 44: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti





u724 - HF & LB - PT1 - (costo ottimo 41910)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43864	0,0	43864	0,0	43915	0,0	43915	0,0
43793	9,0	43796	66,7	43897	4,0	43848	53,2
43736	19,3	43756	129,2	43881	8,4	43781	105,9
43699	23,2	43719	198,2	43878	10,0	43733	163,5
43678	32,1	43684	270,6	43862	12,0	43691	243,8
43611	34,8	43648	342,9	43814	26,6	43652	304,8
43519	37,9	43613	423,6	43784	30,3	43613	360,7
43447	44,9	43583	510,6	43759	47,3	43578	426,6
43441	47,0	43556	612,3	43756	55,4	43544	502,3
43429	49,3	43516	671,6	43699	60,1	43491	562,3
43328	57,6	43469	725,9	43689	64,8	43459	625,2
43307	61,3	43405	760,3	43675	67,6	43427	696,5
43286	66,5	43357	805,4	43582	100,6	43387	768,2
43214	72,6	43316	853,7	43579	210,8	43314	790,0
43151	75,6	43287	919,5	43502	224,6	43289	898,2
43105	77,2	43261	985,9	43457	231,9	43272	1016,0
43045	79,4	43235	1061,2	43293	259,4	43245	1072,4
43021	86,6	43209	1128,0	43248	272,4	43228	1162,3
43016	88,7	43185	1208,4	43218	290,6	43212	1255,7
42976	93,0	43122	1243,0	43140	301,2	43200	1361,4
42971	97,4	43102	1315,1	43132	304,5	43189	1466,2
42948	101,6	43083	1380,3	43086	312,8	43181	1577,7
42930	104,4	43064	1444,7	43025	321,6	43171	1730,6
42907	119,0	43047	1514,2	42987	331,0	43166	1800,0
42747	130,2	43030	1585,4	42942	349,3		
42745	140,9	43014	1657,0	42939	363,2		
42723	149,5	43000	1744,6	42872	373,8		
42707	159,1	42993	1800,0	42860	405,7		
42698	171,1			42821	459,9		
42682	189,1			42813	493,5		
42664	198,5			42760	522,5		
42648	251,2			42744	547,6		
42618	276,0			42726	729,0		
42614	292,6			42707	773,8		
42586	322,1			42686	857,5		
42571	345,4			42652	912,3		
42496	384,8			42617	951,7		
42482	403,7			42598	1000,9		
42478	456,5			42566	1047,8		
42448	474,5			42566	1800,0		
42428	500,1						
42397	517,3						
42396	531,1						
42356	609,3						
42353	703,2						
42352	840,5						
42340	1085,0						
42325	1137,8						
42325	1137,8						

Tabella 45: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



u724 - HF & LB - PT2 - (costo ottimo 41910)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
43852	0,0	43852	0,0	43711	0,0	43711	0,0
43833	5,7	43746	23,7	43690	0,8	43633	47,0
43790	8,1	43668	60,0	43686	3,6	43563	102,5
43726	10,6	43595	102,3	43616	10,4	43511	151,9
43717	12,2	43525	147,9	43570	13,3	43459	202,5
43537	20,6	43474	204,8	43561	18,0	43409	257,4
43515	27,0	43426	272,8	43543	19,8	43362	294,9
43496	40,5	43383	327,5	43531	21,8	43322	342,9
43394	45,0	43342	399,5	43484	27,9	43288	406,1
43363	47,9	43311	465,0	43470	29,3	43253	460,2
43256	52,1	43281	535,2	43445	38,3	43219	514,1
43175	59,1	43256	633,5	43440	40,8	43187	568,3
43159	63,4	43237	709,7	43365	45,5	43156	623,5
43136	67,2	43218	804,4	43355	52,9	43129	693,9
43129	69,4	43201	885,6	43336	57,9	43103	738,5
43125	76,3	43188	990,6	43271	99,2	43080	796,0
43120	84,6	43154	1058,4	43234	109,9	43059	858,1
43066	91,1	43133	1152,4	43220	122,2	43039	916,8
43052	97,4	43121	1225,5	43098	142,2	43020	987,6
43051	112,7	43109	1300,7	43026	149,6	43001	1051,4
43021	130,2	43073	1388,0	42964	163,2	42984	1127,6
42982	137,7	43059	1492,8	42931	167,6	42970	1206,0
42967	144,7	43047	1603,8	42912	191,8	42958	1280,5
42917	162,4	43041	1709,6	42903	204,3	42946	1361,2
42911	169,2	43036	1800,0	42800	225,2	42934	1441,5
42880	201,9			42794	246,4	42924	1529,3
42876	216,7			42725	280,9	42914	1625,5
42867	225,3			42675	293,1	42906	1706,7
42836	238,3			42643	309,7	42896	1786,7
42827	254,1			42628	314,1	42896	1800,0
42816	273,4			42600	325,2		
42800	285,0			42586	334,5		
42751	292,4			42564	347,5		
42731	308,8			42554	368,6		
42722	327,3			42549	402,0		
42680	349,0			42537	434,7		
42632	369,4			42518	467,6		
42621	391,6			42468	493,3		
42604	460,7			42440	513,8		
42536	483,7			42421	568,8		
42520	511,7			42417	592,8		
42510	526,1			42396	631,6		
42457	546,4			42379	664,0		
42446	570,0			42319	673,6		
42437	606,7			42296	681,4		
42392	644,8			42254	706,4		
42389	658,9			42253	737,5		
42367	689,8			42230	755,7		
42325	722,5			42211	766,2		
42324	722,5			42211	766,2		

Tabella 46: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti



rat783 - HF & LB - PT1 - (costo ottimo 8806)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
9284	0,0	9284	0,0	9264	0,0	9264	0,0
9283	4,2	9260	21,9	9260	1,4	9237	14,2
9276	8,2	9238	54,4	9252	3,4	9222	51,0
9269	11,4	9220	81,6	9227	6,4	9210	97,4
9247	15,8	9205	113,9	9222	8,2	9200	153,8
9242	18,5	9193	165,2	9217	11,5	9190	198,7
9237	27,2	9181	201,7	9216	12,9	9181	259,5
9236	30,8	9171	252,6	9206	15,4	9172	327,2
9216	36,5	9161	307,7	9203	20,9	9166	385,8
9201	41,7	9152	349,7	9200	22,2	9161	457,8
9189	46,7	9143	400,5	9193	26,7	9156	533,5
9184	50,3	9134	449,3	9182	37,2	9151	601,9
9183	52,4	9126	500,5	9181	40,1	9147	669,8
9180	55,6	9119	545,5	9177	47,9	9143	761,1
9164	66,0	9112	598,4	9156	52,9	9137	827,5
9156	68,7	9105	669,4	9145	56,9	9134	909,3
9148	74,0	9099	740,4	9139	66,0	9129	963,8
9146	83,9	9093	780,8	9138	69,2	9126	1033,5
9141	93,0	9088	836,1	9137	71,9	9123	1088,8
9138	96,9	9083	899,9	9125	77,6	9120	1160,8
9129	99,7	9078	971,1	9120	81,4	9107	1204,5
9128	105,1	9073	1032,8	9112	89,7	9104	1273,9
9111	115,5	9069	1087,4	9108	94,8	9101	1357,3
9110	117,1	9065	1145,5	9101	101,2	9098	1454,6
9098	120,1	9061	1215,7	9099	107,5	9095	1559,2
9092	122,8	9057	1265,1	9096	149,3	9088	1608,6
9090	130,3	9054	1344,3	9092	165,9	9085	1693,6
9089	133,2	9051	1407,2	9075	200,4	9083	1793,5
9076	142,3	9048	1480,7	9071	220,5	9083	1800,0
9053	147,5	9045	1545,1	9067	230,5		
9045	151,8	9042	1620,0	9064	260,6		
9040	156,8	9040	1691,9	9063	274,7		
9028	159,6	9038	1792,9	9056	303,9		
9025	164,0	9038	1800,0	9016	315,9		
9019	174,9			9015	326,2		
9011	205,4			9013	360,2		
9006	219,6			9004	366,7		
8995	237,2			9002	379,8		
8993	258,5			9000	399,7		
8986	280,6			8998	425,7		
8985	285,6			8994	439,0		
8976	302,8			8993	470,4		
8972	312,5			2078988	485,4		
8963	354,1			8983	490,6		
8956	382,8			8978	502,7		
8955	396,1			8973	526,2		
8954	415,1			8959	536,6		
8948	437,0			8947	546,8		

Tabella 47: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti





rat783 - HF & LB - PT2 - (costo ottimo 8806)							
Thread1 HF		Thread1 LB		Thread2 HF		Thread2 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
9265	0,0	9265	0,0	9242	0,0	9242	0,0
9254	5,0	9250	41,7	9228	8,6	9227	47,4
9248	23,2	9238	85,8	9221	12,0	9216	94,5
9231	27,4	9226	144,9	9218	15,3	9206	174,4
9228	32,1	9215	200,6	9203	20,4	9199	232,2
9227	35,4	9199	243,0	9202	26,3	9192	295,9
9226	40,3	9189	321,3	9198	31,7	9184	347,9
9222	47,4	9180	389,8	9197	34,3	9178	418,2
9220	50,2	9171	458,1	9184	47,0	9172	478,6
9214	55,0	9163	517,6	9160	51,6	9166	557,8
9203	65,2	9155	591,7	9159	59,4	9160	620,4
9198	68,9	9147	652,4	9153	68,9	9154	691,0
9196	75,4	9140	718,9	9152	75,5	9149	774,1
9187	86,9	9133	782,1	9149	78,4	9145	888,5
9185	91,3	9127	855,2	9142	81,2	9141	972,9
9182	100,3	9122	931,2	9138	82,7	9137	1062,1
9162	111,8	9117	1002,5	9134	85,7	9133	1169,8
9158	115,9	9108	1062,3	9127	89,4	9130	1240,0
9148	118,7	9104	1131,1	9115	105,9	9127	1342,9
9128	128,9	9100	1253,4	9111	109,4	9122	1438,8
9122	133,3	9097	1350,6	9100	112,0	9119	1525,5
9113	137,8	9094	1433,2	9095	114,4	9117	1633,4
9112	139,9	9091	1509,5	9094	116,3	9115	1736,7
9108	147,7	9088	1593,0	9089	121,1	9114	1800,0
9106	153,3	9084	1711,6	9081	128,0		
9099	159,3	9081	1800,0	9079	135,6		
9090	175,0			9068	151,6		
9089	183,0			9065	159,8		
9087	188,9			9063	169,2		
9082	203,4			9058	178,4		
9080	222,9			9057	193,6		
9075	245,8			9055	199,0		
9064	266,4			9054	203,6		
9041	273,6			9047	207,4		
9026	278,7			9043	232,0		
9023	296,3			9037	244,1		
9017	314,8			9036	259,8		
9008	320,3			9032	276,7		
9002	328,4			9025	287,9		
8992	343,6			9019	305,7		
8991	359,8			9013	328,0		
8988	369,6			9010	354,9		
8986	419,5			9006	370,6		
8982	423,6			9002	389,6		
8981	444,1			9001	414,5		
8975	452,1			9000	421,9		
8970	462,9			8993	442,5		
8969	484,7			8988	468,3		
8967	503,2			8985	475,2		

Tabella 48: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

dsj1000 - HF & LB - PT1 - (costo ottimo 18659688)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
19828957	0,0	19828957	0,0	19802820	0,0	19802820	0,0
19827606	35,2	19785306	334,9	19802624	12,7	19766778	1800,0
19800821	42,4	19757645	574,7	19797273	21,1		
19800407	46,1	19730261	861,5	19786996	250,5		
19796390	55,4	19705557	1148,6	19764462	264,9		
19790196	61,5	19686452	1642,8	19762089	278,0		
19760026	80,4	19686452	1800,0	19762089	1800,0		
19745769	84,3						
19744871	106,9						
19717210	129,6						
19711613	151,9						
19698196	178,0						
19688308	192,7						
19679234	214,6						
19647964	221,3						
19644534	228,3						
19642590	265,8						
19637587	272,1						
19623073	275,4						
19608692	336,3						
19605498	354,5						
19588944	380,5						
19586950	384,0						
19578103	395,5						
19565140	429,3						
19545338	453,7						
19542075	462,2						
19537854	465,7						
19529469	471,1						
19529469	1800,0						

Tabella 49: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

dsj1000 - HF & LB - PT2 - (costo ottimo 18659688)							
Thread3 HF		Thread3 LB		Thread4 HF		Thread4 LB	
Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)	Costo	Tempo (s)
19741229	0,0	19741229	0,0	19750658	0,0	19750658	0,0
19734873	3,4	19686323	209,5	19707751	59,8	19728370	1800,0
19724110	19,2	19664867	496,3	19707751	1800,0		
19707498	28,4	19646947	856,9				
19697656	67,3	19631054	1232,5				
19697656	1800,0	19619359	1800,0				

Tabella 50: Tabella risultati istanze con numero di nodi compreso tra 200 e 299 + algoritmi esatti

POLISHING OLD

lin318 POLISHING GEN#20 (costo ottimo 42029)	
Costo	Tempo(s)
53491	0,0
43973	139,5
43850	218,8
43108	266,1
42747	280,5
42575	312,0
42446	317,2
42366	320,7
42311	322,7
42246	508,5
42216	716,3
42199	893,1
42149	923,9
42136	1089,7
42125	1166,8
42040	1183,8
42029	1228,0

Tabella 51: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

pr439 POLISHING GEN#20 (costo ottimo 107217)	
Thread1	
Costo	Tempo (s)
134016	0,0
114211	111,6
112134	140,6
110751	147,6
110626	153,5
109026	164,0
108898	174,6
108876	183,3
108472	288,2
108294	311,4
108267	376,3
108022	391,2
108002	469,7
107785	565,0
107783	1036,6
107762	1147,4
107734	1166,4
107715	1172,5
107692	1175,2
107674	1244,1
107671	1432,1
107653	1732,3
107653	1800

Tabella 52: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

d493 POLISHING GEN#20 (costo ottimo 35002)	
Costo	Tempo(s)
35919	357,9
35864	375,5
35798	418,2
35760	440,9
35592	506,4
35591	538,8
35535	565,3
35503	721,3
35477	766,2
35475	773,0
35471	787,8
35349	1007,6
35335	1079,2
35332	1128,4
35291	1504,2
35279	1712,8
35279	1800,0

Tabella 53: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

rat575 POLISHING GEN#20 (costo ottimo 6773)

Costo	Tempo(s)
8445	0,0
7572	12,1
7555	38,7
7499	47,2
7412	58,0
7294	196,9
7270	205,1
7264	236,1
7251	241,4
7247	245,0
7245	251,8
7244	254,2
7238	256,6
7235	264,4
7179	315,4
7157	345,1
7122	393,7
7109	408,4
7092	436,2
7061	478,4
7054	485,2
7044	489,1
7037	499,8
7034	533,3
7033	538,4
7032	542,9
7024	573,5
7021	588,2
7013	662,4
7006	724,1
6991	781,5
6979	808,0
6934	838,7
6931	858,7
6923	938,6
6920	978,1
6913	1028,3
6906	1036,3
6898	1069,7
6895	1086,2
6894	1093,6
6893	1167,7
6879	1261,6
6875	1268,7
6872	1274,7
6868	1282,3
6866	1307,5
6858	1370,4
6849	1470,9
6846	1478,2



d657 POLISHING GEN#20 (costo ottimo 48912)	
Costo	Tempo(s)
58910	0,1
54332	23,1
54028	48,9
53136	69,7
53115	91,0
52659	99,3
52409	120,3
52359	136,2
51692	208,5
51666	224,4
51637	293,0
51618	323,7
51478	388,9
51427	412,7
51300	416,2
51143	483,8
51097	505,6
51060	584,6
51006	594,5
50969	600,9
50667	747,0
50533	844,0
50490	913,7
50489	926,6
50472	956,8
50449	1015,2
50421	1082,7
50363	1255,6
50360	1359,2
50203	1463,2
50197	1741,5
50174	1764,8
50137	1784,3
50137	1800,0

Tabella 55: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

u724 POLISHING GEN#20 (costo ottimo 41910)	
Costo	Tempo(s)
52713	0,1
46497	28,4
46449	49,4
45327	143,6
45144	236,9
44385	257,1
44335	282,4
44114	448,6
44102	457,5
43965	560,8
43835	675,4
43802	707,7
43765	723,1
43753	747,5
43734	1144,8
43707	1455,2
43695	1477,9
43642	1687,1
43642	1800,0

Tabella 56: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

rat783 POLISHING GEN#20 (costo ottimo 8806)	
Costo	Tempo(s)
10569	0,1
9845	452,9
9766	487,3
9660	598,2
9592	620,3
9503	645,5
9484	675,4
9362	695,1
9360	720,3
9321	753,3
9312	783,8
9177	927,7
9118	967,6
9077	1049,7
9058	1108,4
9052	1134,1
9037	1218,8
9034	1232,5
9023	1261,6
9018	1340,7
9011	1530,9
9010	1554,4
9003	1638,3
8999	1733,4
8980	1745,3
8980	1800,0

Tabella 57: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

dsj1000 POLISHING GEN#20 (costo ottimo 18659688)	
Costo	Tempo(s)
24020026	0,2
21338705	36,2
21285117	177,8
21118513	207,0
20915692	257,8
20544991	309,2
20527792	415,3
20391806	454,1
19949773	1965,0
19949773	1800,0

Tabella 58: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

**POLISHING NEW**

lin318 POLISHING GEN#20 (costo ottimo 42029)	
Costo	Tempo(s)
54981	0,1
51105	6,2
49247	13,5
47680	20,1
46976	26,5
46911	32,4
46388	37,5
46111	42,1
45970	46,7
45650	51,7
45049	57,4
44750	63,3
44551	69,0
44090	79,1
43998	83,4
43927	87,8
43835	91,9
43596	96,3
43143	109,2
43089	117,4
42921	137,7
42878	181,9
42855	225,8
42832	271,1
42820	275,0
42797	286,0
42778	293,5
42769	335,2
42752	380,8
42711	729,0
42692	777,4
42667	874,4
42642	958,9
42629	1024,6
42605	1272,4
42575	1604,5
42550	1652,1
42550	1800

Tabella 59: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

pr439 POLISHING GEN#20 (costo ottimo 107217)	
Thread1	
Costo	Tempo (s)
134289	0,1
124339	9,8
122260	18,1
120612	33,9
118650	41,1
118050	47,9
116941	55,1
116030	62,0
114880	75,5
113888	88,1
113629	101,2
112983	107,3
112872	113,4
112828	119,9
112785	137,8
112782	161,5
112733	184,9
112461	249,7
112458	256,0
112409	262,2
112401	268,8
112260	286,8
112173	299,0
112165	317,2
112129	380,4
111704	440,0
111668	450,7
111640	472,3
111631	550,8
111484	663,4
111456	723,5
111309	729,4
111289	762,8
111252	882,4
111180	949,2
111040	1073,3
110852	1263,8
110781	1342,3
110376	1537,1
110376	1800

Tabella 60: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

d493 POLISHING GEN#20 (costo ottimo 35002)	
Costo	Tempo(s)
41715	0,1
39768	20,8
38964	40,2
38593	53,0
38148	66,0
37575	77,2
37232	87,0
37203	96,8
36875	107,5
36768	116,7
36574	125,2
36469	133,6
36428	142,1
36329	158,9
36277	167,4
36227	175,9
36096	184,2
36064	192,4
36053	215,4
36045	223,3
36042	239,1
36039	245,9
36032	253,5
36006	261,0
35947	268,1
35882	289,3
35864	443,1
35862	450,7
35861	540,7
35775	614,4
35726	974,8
35720	1443,6
35714	1464,8
35695	1544,0
35690	1565,0
35690	1800,0

Tabella 61: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti



rat575 POLISHING GEN#20 (costo ottimo 6773)	
Costo	Tempo(s)
8540	0,2
7800	36,7
7767	60,9
7449	79,8
7405	99,8
7369	116,3
7332	129,9
7276	144,1
7245	173,7
7163	187,7
7133	201,4
7128	214,2
7095	228,0
7070	240,7
7014	267,5
7005	280,8
7002	293,5
6948	307,2
6938	320,5
6922	333,8
6917	346,6
6896	359,1
6895	369,8
6887	382,3
6884	394,6
6883	407,5
6879	420,1
6877	432,5
6875	454,9
6873	480,8
6871	612,6
6870	643,2
6867	759,4
6866	881,4
6864	893,1
6860	1006,8
6855	1645,6
6855	1800,0

Tabella 62: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

d657 POLISHING GEN#20 (costo ottimo 48912)	
Costo	Tempo(s)
61347	0,2
56669	21,9
55895	41,1
55054	78,1
54228	95,5
52536	109,2
52506	123,2
52452	149,4
52097	161,7
51585	174,7
51433	186,9
51114	199,8
50643	224,1
50485	235,9
50401	259,4
50367	304,2
50283	326,4
50261	337,4
50252	409,5
50221	429,2
50177	532,3
50176	542,7
50132	589,7
50131	617,9
50090	721,2
50069	759,6
50063	769,1
50021	815,2
50001	919,2
49993	928,9
49973	949,5
49945	967,5
49942	1004,3
49883	1106,3
49865	1135,3
49822	1144,8
49777	1257,1
49766	1367,3
49721	1377,2
49715	1480,8
49677	1679,7
49664	1784,9
49664	1800

Tabella 63: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

u724 POLISHING GEN#20 (costo ottimo 41910)	
Costo	Tempo(s)
51626	0,2
49285	21,4
48591	39,7
47940	56,6
46975	73,8
46558	89,4
46129	105,9
45741	121,6
45662	137,3
45308	153,1
45163	168,4
44871	184,2
44321	198,7
44181	213,9
44006	243,1
43871	273,1
43801	302,7
43767	317,5
43647	331,9
43560	346,1
43450	359,8
43440	387,3
43432	415,1
43408	457,0
43314	600,7
43305	664,4
43271	677,6
43256	718,5
43200	865,4
43170	913,9
43127	1046,7
43095	1059,2
43064	1083,9
43033	1130,7
43002	1176,0
42988	1198,6
42978	1327,2
42917	1586,8
42916	1599,0
42855	1611,4
42845	1748,9
42826	1800

Tabella 64: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

rat783 POLISHING GEN#20 (costo ottimo 8806)	
Costo	Tempo(s)
11112	0,3
10506	47,1
10299	81,3
10120	116,8
10021	145,6
9894	174,6
9769	208,7
9544	233,3
9521	261,9
9494	287,6
9400	310,7
9381	334,0
9350	354,2
9295	375,8
9273	399,0
9238	435,1
9220	454,7
9212	473,1
9209	536,1
9204	554,2
9189	572,2
9185	593,1
9178	624,8
9175	666,3
9170	685,6
9168	726,6
9167	923,6
9165	957,9
9160	1128,5
9157	1145,2
9152	1240,2
9150	1271,4
9147	1436,0
9146	1634,5
9146	1800,0

Tabella 65: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti

dsj1000 POLISHING GEN#20 (costo ottimo 18659688)	
Costo	Tempo(s)
24494420	0,4
22674664	76,3
21691770	158,5
21155611	256,3
20987474	325,0
20616148	356,1
20422249	388,6
20213051	418,5
20178946	472,5
20143854	496,5
20011270	523,3
20007438	549,7
19845787	577,3
19787330	601,7
19784020	627,0
19738981	674,3
19703485	721,7
19701881	746,1
19697274	865,4
19693950	907,0
19642970	1138,3
19629331	1159,7
19626007	1180,0
19614747	1198,8
19605271	1237,8
19600250	1297,9
19592437	1341,1
19559338	1555,4
19553706	1595,0
19551042	1634,7
19535350	1654,9
19525508	1674,1
19502251	1732,7
19502251	1800,0

Tabella 66: Tabella risultati istanze con numero di nodi inferiore a 200 + algoritmi esatti