Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Tesina di Ricerca Operativa 2

Travelling Salesman Problem

Autori

Raffaele Di Nardo Di Maio 1204879

Cristina Fabris 1205722

Anno Accademico 2019-2020

Indice

Capitolo 1

Introduzione

La seguente trattazione analizza il Problema del Commesso Viaggiatore (Travelling Salesman Problem, TSP), che consiste nell'individuare un circuito hamiltoniano di costo minimo in un assegnato grafo orientato G=(V,A)[10]. La formulazione matematica di tale problema è la seguente:

$$x_{ij} = \begin{cases} 1 & \textit{se l'arco } (i,j) \in A \textit{ viene scelto nel circuito ottimo} \\ 0 & \textit{altrimenti} \end{cases}$$

$$min \sum_{(i,j)\in A} c_{ij}\, x_{ij} \tag{1.1}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1 \qquad \forall\, j \in V \tag{1.2}$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} = 1 \qquad \forall\, i \in V \tag{1.3}$$

$$\sum_{(i,j)\in\delta^+(S)} x_{ij} \geq 1 \qquad S \subset V \;:\; 1 \in S \tag{1.4}$$

$$x_{ij} \geq 0 \;\; intero \qquad (i,j) \in A. \tag{1.5}$$

Tuttavia le soluzioni algoritmiche presentante risolvono una sua variante, detta simmetrica, che viene applicata ad un grafo completo non orientato G=(V,E).
Di seguito viene riportata la formulazione matematica di tale versione:

$$min \sum_{e\in E} c_e\, x_e \tag{1.6}$$

$$\sum_{e\in\delta(v)} x_e = 2 \qquad \forall\, v \in V \tag{1.7}$$

$$\sum_{e\in E(S)} x_e \leq |S| - 1 \qquad \forall\, S \underset{\neq}{\subset} V : |S| \geq 3. \tag{1.8}$$

A livello commerciale esistono diverse tipologie di risolutori di problemi di programmazione lineare intera, basati sul Branch & Bound. I più conosciuti in circolazione sono i seguenti:

- **IBM ILOG CPLEX Optimization Studio**[1]
  è un soluzione analitica, sviluppata dall'IBM e gratuita a livello accademico.

- **FICO® Xpress Optimization**[2]
  è stato prodotto dalla Fair Isaac Corporation(FICO) ed è costituito da 4 componenti principali: FICO® Xpress Insight, FICO® Xpress Executor, FICO® Xpress Solver e FICO® Xpress Workbench. Questa soluzione è disponibile gratuitamente solo nella versione Community, in cui però vengono applicate restrizioni sul numero di righe e colonne del tableau, di token non lineari e di funzioni dell'utente.

- **Gurobi**[3]
  è una soluzione, sviluppata dalla Gurobi Optimization, che viene rilasciata anche con una versione accademica.

- **COIN Branch and Cut solver (CBC)**[4]
  è un risolutore MIP(mixed-integer program) open-source scritto in C++ e sviluppato dalla Computational Infrastructure for Operations Researc (COIN).

Nel Capitolo 2 vengono riportate diverse soluzioni math-euristiche e non per il problema del Commesso Viaggiatore, che fanno uso di ILOG CPLEX.
In commercio, il più noto ed efficiente software per la risoluzione del TSP è Concorde, sviluppato in ANSI C e disponibile per l'uso in ambito accademico[5].
Nel Capitolo 3 vengono analizzati gli algoritmi euristici, sviluppati senza far uso di ILOG CPLEX.
Nel Capitolo 4 vengono invece riportati i confronti, a livello temporale e di costo, delle soluzioni ottenute con i differenti algoritmi enunciati.
Nell'Appendice A, B, C, D vengono descritte rispettivamente la tipologia di istanze utilizzate, la documentazione utilizzata ed il funzionamento di CPLEX, il programma GNUPLOT utilizzato nella stampa delle soluzioni e il programma perfprof.py usato per creare i performance profile del Capitolo 4.
Tutti i tempi di esecuzione e i costi delle soluzioni, ottenuti mediante la fase di testing, sono consultabili invece nelle tabelle riportate nell'Appendice E. Tutte le soluzioni descritte sono state implementate in linguaggio C e testate sul sistema operativo Windows 10 con Visual Studio, ed i sorgenti sono disponibili online[1].

[1] `https://github.com/RaffaDNDM/Operational-Research-2`

Capitolo 2

Risoluzione del problema tramite CPLEX

2.1 Algoritmi Esatti

2.1.1 Modelli compatti

I modelli compatti del Travelling Salesman Problem, sono formulazioni matematiche in cui il numero di variabili e di vincoli è polinomiale nella taglia dell'istanza. In particolare, in quelle analizzate in seguito,esse sono entrambe $O(n^2)$, con n = *numero di nodi*.
I modelli compatti sono però applicabili solo a grafi orientati. Quindi ciascuna variabile x_{ij} del modello simmetrico deve essere convertita nelle due variabili x_{ij} e x_{ji} del corrispondente modello orientato (vedi Capitolo 1). Questo comporta un significativo rallentamento nella computazione della soluzione, in quanto il metodo di risoluzione, nel momento in cui si trovi a scartare un ramo (i, j) dalla possibile soluzione, deve verificare comunque se il corrispondente lato (j, i) potrebbe invece farne parte. Tale controllo risulta però inutile, poichè i due rami vengono associati alla stessa variabile nel modello simmetrico iniziale.

2.1.1.1 Formulazione di Miller, Tucker e Zemlin

Nella formulazione del modello di Miller, Tucker e Zemlin, detta anche formulazione sequenziale, per ogni nodo i del grafo viene introdotta una nuova variabile u_i, che rappresenta la posizione di tale vertice all'interno del circuito restituito. Vengono introdotti inoltre nuovi vincoli basati sulle variabili u_i che garantiscono la formazione di un unico tour e l'eliminazione di tutti i possibili subtour. Nello specifico il modello utilizzato da Miller, Tucker e Zemlin è il seguente[7]:

$$min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{2.1}$$

$$\sum_{i \in V} x_{ih} \;=\; 1 \qquad \forall\, h \in V \tag{2.2}$$

$$\sum_{j \in V} x_{hj} \;=\; 1 \qquad \forall\, h \in V \tag{2.3}$$

$$u_i - u_j + n\, x_{ij} \;\leq\; n - 1 \qquad \forall\, i, j \in V - \{1\}, i \neq j \tag{2.4}$$

$$0 \leq u_i \;\leq\; n - 2 \qquad \forall\, i \in V - \{1\} \tag{2.5}$$

Esistono però due diversi modi per inserire i vincoli aggiuntivi, legati alle variabili u_i, sfruttando le funzioni di CPLEX.
Nel primo questi vengono aggiunti direttamente al modello, durante la costruzione di quest'ultimo. In tal modo, durante la fase di preprocessamento, il programma è già a conoscenza di tutti i vincoli che dovrà rispettare la soluzione ottima e ciò gli permette di apportare ulteriori semplificazioni al modello, rilevandone alcune proprietà utili.
Il secondo metodo, invece, prevede l'inserimento nel modello di questi vincoli tramite "lazy constraint". In tal modo i vincoli non sono noti al programma dall'inizio, ma vengono inseriti all'interno di un pool. Nel momento in cui viene calcolata una soluzione, CPLEX ne verificherà la correttezza analizzando l'insieme di vincoli precedentemente definito e se dovesse trovarne uno violato, lo aggiungerà al modello e ripeterà la computazione. Questo secondo approccio implementativo permette di eseguire calcoli su un modello di dimensioni inferiori rispetto a quello ottenuto utilizzando il primo metodo. Tuttavia i tempi di calcolo possono aumentare significativamente in quanto CPLEX non può sfruttare la conoscenza dell'intero modello sin dalla sua costruzione. Nell'implementazione sviluppata, sono state impiegate entrambe le soluzioni descritte.

2.1.1.2 Formulazione di Gavish e Graves

Nella formulazione di Gavish e Graves, per impedire la formazione di sub-tour all'interno della soluzione, viene associata a ciascun ramo una nuova variabile y_{ij}, che rappresenta il flusso tra i nodi i e j. Il modello, legato al problema del commesso viaggiatore, assume la seguente forma[7]:

$$min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{2.6}$$

$$\sum_{i \in V} x_{ih} \;=\; 1 \qquad \forall\, h \in V \tag{2.7}$$

$$\sum_{j \in V} x_{hj} \;=\; 1 \qquad \forall\, h \in V \tag{2.8}$$

$$\sum_{j \in V} y_{1j} \;=\; n - 1 \tag{2.9}$$

$$\sum_{j \in V} y_{hj} \;=\; \sum_{i \in V} y_{ih} \; - 1 \qquad \forall\, h \in V - \{1\} \tag{2.10}$$

$$y_{ij} \leq \; (n-1)\, x_{ij} \qquad \forall\, i, j \in V, i \neq j \tag{2.11}$$

$$y_{ii} = 0 \qquad \forall\, i \in V \tag{2.12}$$

$$y_{i1} = 0 \qquad \forall\, i \in V \tag{2.13}$$

In questa formulazione matematica, i vincoli di subtour elimination vengono sostituiti dai vincoli (2.9), (2.10), (2.11), (2.12), (2.13) che rappresentano rispettivamente: il flusso di uscita dal primo nodo, il bilanciamento dei flussi in ogni singolo nodo, i vincoli di accoppiamento del flusso con un ramo selezionato nella soluzione, il valore del flusso su auto-anelli e il valore del flusso entrante nel primo nodo.
La soluzione di questo modello risulta essere però molto lontana dalla convex hull e per migliorarla è possibile sostituire il vincolo (2.11) con:

$$y_{ij} \leq (n-2)\, x_{ij} \quad \forall\, i \neq j.$$

Inoltre per evitare che la soluzione ottima imposti ad 1 entrambi gli archi orientati x_{ij} e x_{ji}, che nell'istanza iniziale sono legati allo stesso arco, viene aggiunto anche il seguente vincolo:

$$x_{ij} + x_{ji} \leq 1 \quad \forall i, j \in V \text{ con } i < j$$

L'implementazione svolta all'interno del programma utilizza i vincoli (2.11) nella loro prima forma e li inserisce nel modello in fase di costruzione.

2.1.2 Loop

Negli anni '60, Jacques F. Benders sviluppò un approccio generale, applicabile a qualsiasi problema di programmazione lineare, per ridurre il numero esponenziale di vincoli presenti in un modello. Per ovviare tale problema, il metodo Loop costruisce inizialmente il modello senza quei vincoli e li aggiunge solo in seguito durante la risoluzione. L'algoritmo di Benders calcola una soluzione e valuta se questa rispetti tutti i vincoli non inseriti nel modello. Nel caso in cui dovesse trovarne uno che non viene rispettato, lo inserisce nel modello.
Nel caso del TSP, i vincoli in numero esponenziale sono i Subtour Elimination (SEC), che hanno la seguente forma:

$$\sum_{e \in E(S)} x_e \leq \mid S \mid -1 \; \forall\, S \underset{\neq}{\subset} V \; : \mid S \mid \geq 2 \tag{2.14}$$

o equivalentemente:

$$\sum_{e \in \delta(S)} x_e \geq 2 \; \forall\, S \underset{\neq}{\subset} V \; : \mid S \mid \geq 2 \tag{2.15}$$

Definendo il modello per il problema del commesso viaggiatore, vengono rimossi i SEC. In seguito viene risolto il problema e nel caso in cui la soluzione abbia più di una componente connessa, viene aggiunto al modello un vincolo di Subtour Elimination per ogni ciclo generato. Il processo viene ripetuto iterativamente, come mostrato nel seguente pseudocodice, fino a che la soluzione non sia costituita da un unico tour.

Algoritmo 1: LOOP

Input: `model`= Modello TSP simmetrico senza vincoli di Subtour Elimination

Output: `x`= soluzione intera senza subtour

```
1 x ← solve(model)
2 ncomps ← comps(x)
3 while ncomps > 1 do
4 |   Aggiungi sum_{e∈δ(S_k)} x_e ≤| S_k | −1 ∀ componente connessa S_k
5 |   if ncomps ≥ 2 then
6 |   |   x ← solve(model)
7 |   |   ncomps ← comps(x)
```

All'aumentare del numero di vincoli inseriti nel modello, il costo della soluzione peggiora o resta invariato rispetto al costo di quella elaborata all'iterazione precedente del metodo Loop.
Il numero di iterazioni che vengono effettuate dall'algoritmo non è calcolabile a priori e potrebbe essere anche molto elevato. Nel caso peggiore verranno inseriti tutti i vincoli di Subtour Elimination, ovvero un numero esponenziale di disequazioni, soprattutto nel caso di istanze clusterizzate. L'introduzione di nuovi SEC, solo nel momento in cui si presenta una loro violazione, permette di ridurre la dimensione del modello ma diminuisce l'attività di pre-processamento svolta da CPLEX

prima di risolvere il problema. Inoltre il problema principale del metodo Loop è la generazione di un nuovo albero completo di branching ad ogni nuova iterazione.
In passato, con le versioni del MIP solver di CPLEX, questa operazione era molto onerosa mentre attualmente il metodo loop garantisce la risoluzione, anche di istanze molto grandi, in tempi ragionevoli. Questo non accade invece per il Branch & Bound in quanto vengono aggiunte nuove ramificazioni all'albero già esistente.
Il metodo Loop può essere modificato svolgendo prima l'algoritmo Loop con l'aggiunta di parametri differenti da quelli utilizzati di default del risolutore CPLEX e solo in seguito effettuando l'algoritmo di Benders, questa volta utilizzando le impostazioni di default di CPLEX.
Quest'ottimizzazione è basata sul fatto che CPLEX salvi alcune soluzioni, ottenute in precedenza dal risolutore sullo stesso modello, e le sfrutti come bound nel nuovo modello. Per questo motivo, la soluzione euristica ottenuta nella prima fase viene sfruttata come bound nella seconda.

2.1.3 Branch & Cut

Come precedentemente anticipato, CPLEX effettua inizialmente una fase di pre-processamento in cui semplifica il modello, e terminata questa operazione inizia ad eseguire la fase di Branch & Cut. Ogni volta che CPLEX calcola una nuova soluzione x^*, prima di accettare la soluzione o scartarla e proseguire nello sviluppo dei successivi rami dell'albero decisionale, applica dei tagli e degli algoritmi euristici per aggiornarla (vedi Figura 2.1).

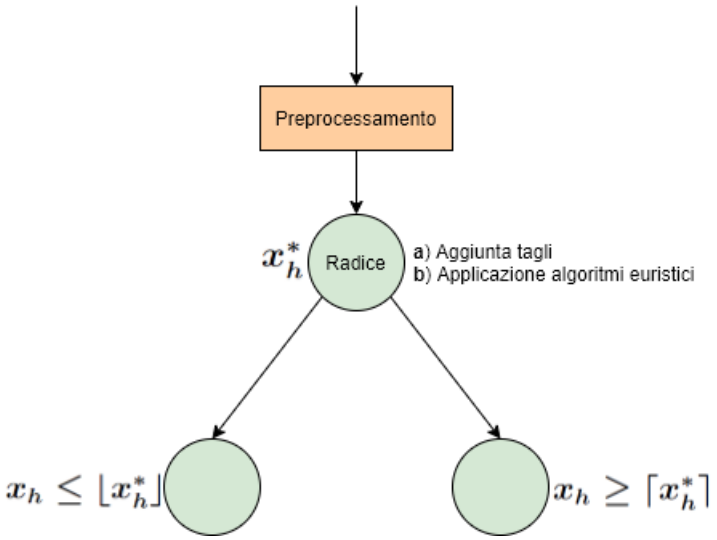

Figura 2.1: Albero decisionale del Branch and Cut

Nello sviluppo del Branch & Cut, per ciascun nodo dell'albero decisionale, vengono considerati due bound:

- **Upper Bound**
  viene definito dagli algoritmi euristici utilizzati.

- **Lower Bound**
  viene definito dal rilassamento del problema e dall'azione di diversi tagli.

CPLEX permette di personalizzare i tagli da applicare nel Branch & Cut e di inserire iterativamente i Subtour Elimination relativi alle componenti connesse nella soluzione. Per fare ciò l'utente può implementare questi vincoli mediante delle callback, dette *lazy constraints callback*, utilizzate per aggiungere lazy constraint al modello.
Le lazy callback implementate vengono chiamate solo nel momento in cui deve essere aggiornato

l'incumbent e se necessario CPLEX aggiungerà al modello i vincoli violati. Queste callback verranno quindi invocate più frequentemente all'inizio del calcolo della soluzione del problema, e meno nelle iterazioni successive. Questo poichè essendoci in partenza meno vincoli, sarà più facile per la soluzione soddisfarli tutti.
A differenza dei *lazy constraint*, con l'utilizzo delle *lazy callback* i vincoli non sono costantemente presenti in un pool, ma vengono generati "al volo" quando necessario. Quest'operazione velocizza notevolmente il calcolo della soluzione ottima, in quanto permette a CPLEX di non dover calcolare nuovamente l'albero decisionale dalla radice, ma di procedere nel suo sviluppo aggiungendo nuovi rami.
In particolari casi, però, CPLEX può ritenere più conveniente distruggere l'intero albero decisionale finora calcolato e ricominciare la computazione dalla radice. Questo può avvenire in qualunque punto dell'elaborazione della soluzione ottima e, utilizzando istanze abbastanza grandi, queste interruzioni aumentano e sono evidenti mediante i log di CPLEX.
Attraverso l'utilizzo delle callback è possibile accedere a numerose informazioni relative alle elaborazioni fatte da CPLEX. Per questo motivo, alcune procedure vengono automaticamente disattivate, affinché l'utente non possa venire a conoscenza di particolari dettagli implementativi. Un esempio è la dynamic search. Tale blocco può causare però un drastico rallentamento nel calcolo della risoluzione.
Nelle ultime versioni di CPLEX sono state introdotte le *generic callback* che permettono di mantenere attivi tutti i meccanismi presenti all'interno del programma per la computazione della soluzione e di ovviare quindi tale problema.

2.1.4 Patching

Negli algoritmi analizzati nelle precedenti sezioni può succedere che CPLEX, prima di trovare il tour ottimo, computi soluzioni con più componenti connesse. Per evitare che vengano scartate senza essere sfruttate, per via dei vincoli di subtour elimination, è possibile utilizzare questo l'algoritmo Patching per costruire un nuovo tour ammissibile dalla soluzione attuale.
Due componenti connesse all'interno della soluzione calcolata da CPLEX, grazie all'eliminazione di un ramo da ciascuna di esse ($\{a, a'\}$ e $\{b, b'\}$ rispettivamente), vengono unite in un unico tour. Per fare ciò vengono analizzate le due possibili configurazioni.

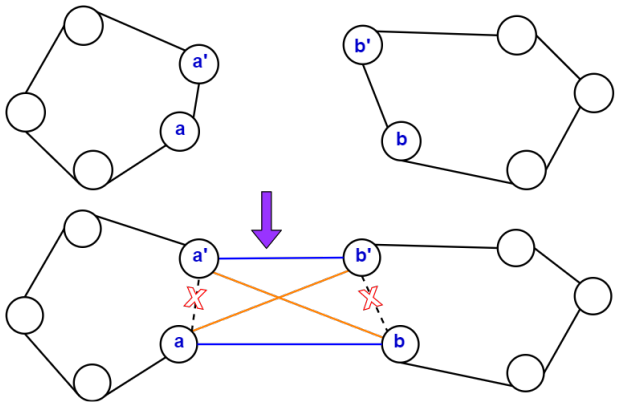

Figura 2.2: Esempio di patching

I nuovi collegamenti possibili tra le due componenti prevedono l'inserimento degli coppia di rami $\{a, b\}$ e $\{a', b'\}$ o dei due lati $\{a, b'\}$ e $\{a', b\}$ (vedi Figura 2.2). In fase di unione viene selezionata la coppia che comporterebbe nella soluzione la minima variazione di costo. Tale scelta garantisce di unire le due componenti senza creare alcun incrocio.
Per scegliere la coppia di rami (a, a') e (b, b') da rimuovere e di conseguenza le componenti coinvolte nell'unione, è necessario minimizzare la variazione del costo complessivo dovuta alla creazione del

nuovo tour:

$$\min_{\forall\, a,b \in V:\ [comp(a)=comp(a')] \wedge [comp(b)=comp(b')] \wedge [comp(a) \neq comp(b)]} (\Delta\ (a,a',b,b'), \Delta'\ (a,a',b,b'))$$

dove:

$$\begin{aligned} \Delta\ (a,a',b,b') &= c_{ab'} + c_{ba'} - c_{aa'} - c_{bb'} \\ \Delta'\ (a,a',b,b') &= c_{ab} + c_{b'a'} - c_{aa'} - c_{bb'} \end{aligned}$$

L'operazione di fusione di due componenti connesse dev'essere essere ripetuta finché la soluzione non diventi ammissibile.
Nella nostra implementazione è stato scelto di espandere sempre la stessa componente, unendola a quelle più vicine. Grazie a questa scelta viene minimizzato il numero di rami per cui è necessario modificare la struttura dati, che memorizza l'ordine di percorrenza dei nodi.
Per poter implementare l'algoritmo è necessario utilizzare due diversi tipi di callback messe a disposizione da CPLEX. La prima appartiene alla tipologia delle *lazy constraint callback* ed è necessaria per ricevere la soluzione trovata dal programma e rielaborarla. A questa soluzione viene applicato l'algoritmo di patching ed il risultato viene memorizzato all'interno di una struttura dati accessibile anche dalla seconda callback creata dell'utente.
La seconda callback necessaria è una *heuristic callback* e permette all'utente di passare a CPLEX una soluzione da cui proseguire la computazione. Questa soluzione sarà selezionata dalla struttura dati, popolata dalla prima callback, e verrà utilizzata da CPLEX solo nel caso in cui sia migliore dell'incumbent attuale.
Utilizzando, invece, le *generic callback* non è necessario implementare due diverse user-callback. E' sufficiente invocare la callback in due differenti contesti:

- **CPX_CALLBACKCONTEXT_CANDIDATE**
  per leggere la soluzione calcolata da CPLEX, aggiungere i subtour elimination e applicare l'algoritmo di patching;

- **CPX_CALLBACKCONTEXT_LOCAL_PROGRESS**
  per suggerire a CPLEX una soluzione, calcolata precedentemente con l'algoritmo Patching;

Per garantire che le callback siano thread-safe, la struttura dati ausiliaria, utilizzata per memorizzare le soluzioni calcolate con l'algoritmo Patching, è stata organizzata in modo tale che ogni thread acceda e modifichi esclusivamente determinate informazioni.

Algoritmo 2: Patching

Input: x= soluzione di un problema di TSP con più componenti connesse

Output: y= soluzione intera formata da un'unica componente connessa

```
1 n_comps ←numerocomponenticonnessedellasoluzione
2 c_1 ←{0, ..., 0}
3 while n_comps > 1 do
4     c_1 ← first_subtour(x)
5     c_2 ← nearest_subtour(c_1)
6     merge_component(c_1, c_2)
7     update(n_comps)
8 y ←c_1
```

2.2 Algoritmi Math-Euristici

Gli algoritmi euristici sono progettati per risolvere istanze del problema in tempi significativamente più brevi rispetto agli algoritmi esatti. Di conseguenza, però, al termine della computazione non garantiscono di ottenere una soluzione ottima, ma solo una sua buona approssimazione ammissibile. Gli algoritmi Math-euristici sfruttano l'approccio utilizzato dai metodi euristici unito alla programmazione matematica, introducendo nuovi vincoli al modello. L'algoritmo che maggiormente rappresenta questo metodo è il Soft Fixing (vedi Sezione 2.2.2).
Durante la computazione della soluzione, CPLEX utilizza diversi algoritmi euristici e math-euristici e, variando alcuni parametri, è possibile cambiare la frequenza con cui vengono applicati o il tempo a loro dedicato.

2.2.1 Hard Fixing

Un primo algoritmo math-euristico di semplice implementazione è l'Hard Fixing, composto dalle seguenti fasi:

1. Impostazione di un time limit per la computazione di una soluzione;
2. Calcolo della soluzione;
3. Selezione, in maniera randomica, di un sottoinsieme di rami appartenenti alla soluzione ottima (Figura 2.3). Il numero di questi è definito da una percentuale fissata sul totale dei lati scelti. Le variabili dei rami selezionati, sono impostate al valore 1.

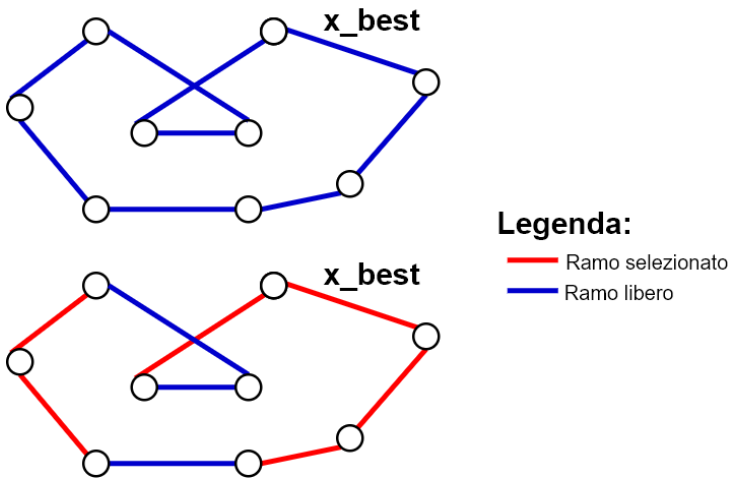

Figura 2.3: Selezione rami

Questi passaggi vengono eseguiti in maniera ciclica per un numero fissato di iterazioni. In questo modo, ad ogni computazione della soluzione, CPLEX dovrà risolvere un problema più semplice di quello originale, essendo molte variabili del modello già selezionate nella **Fase 3** dell'iterazione precedente.
All'interno del programma sviluppato, il time limit nominato nella **Fase 1** è dato da una frazione della deadline complessiva inserita dall'utente e dipende dal numero di diverse percentuali che si desidera utilizzare. All'ultima iterazione il time limit viene ricalcolato in base al tempo ancora a disposizione per il raggiungimento della deadline complessiva inserita dall'utente. Ad ogni computazione il costo della soluzione potrà solo migliorare o restare invariato rispetto alla precedente soluzione calcolata.
La percentuale relativa al numero di rami da fissare può variare ad ogni iterazione. Generalmente viene utilizzata una percentuale alta nelle prime iterazioni, in cui la soluzione non è ancora stata raffinata, e viene ridotta nelle successive, in modo da aumentare i gradi di libertà per CPLEX nella

computazione della soluzione.
La selezione in maniera randomica degli archi nella **Fase 3**, garantisce che l'algoritmo termini poiché non vengono fissate sempre le stesse variabili. Particolare attenzione deve essere posta al fatto di lasciare, nell'insieme dei rami scelti randomicamente, solo quelli selezionati all'iterazione immediatamente precedente. Nell'implementazione sviluppata, le percentuali utilizzate sono state memorizzate in un vettore in questo ordine { 90%, 75%, 50%, 25%, 10%, 0%}. Di seguito viene inoltre riportato lo pseudocodice dell'implementazione sviluppata:

Algoritmo 3: Hard Fixing

Input: `model`= modello TSP simmetrico senza vincoli di Subtour Elimination
`deadline`= time limit complessivo dell'algoritmo
`percentage`= array con i valori delle percentuali di fissagio degli archi
`num_nodi`= numero di nodi dell'istanza TSP

Output: `x`= soluzione intera senza subtour

```
1  n ←0
2  while expired_time < deadline do
3      setTimeLimit()
4      x ← solve(model)
5      for j ←0 to num_nodi − 1 do
6          k ← random(0, 1)
7          if 100 ∗ k ≤ percentage[n mod lenght(percentage)] then
8              Aggiungi x_best[j] to S where S = {edges to fix}
9      forall x_ij ∈ S do
10         x_ij ←1
11     n ←n + 1
```

2.2.2 Soft Fixing

Il metodo seguente fa utilizzo di vincoli aggiuntivi, detti di **Local Branching**[11]. L'approccio utilizzato è simile a quello dell'Hard Fixing, in questo caso però la scelta delle variabili da imporre a 1 non avviene in maniera randomica ma viene lasciata a CPLEX.
Partendo da una soluzione intera ammissibile del TSP x^H, viene aggiunto un vincolo sulle sue variabili con valore 1:

$$\sum_{e \in E \,:\, {x_e}^H = 1} x_e \;\geq\; 0.9\, n$$

dove la sommatoria indica il numero di variabili, uguali a 1 in x^H, che non cambieranno il loro valore e **n** indica il numero di archi selezionati in x^H, pari al numero di nodi.
In questo caso, il vincolo permetterà a CPLEX di fissare il 90% dei rami scelti in x^H e avere il 10% di libertà. Un modo alternativo di scrivere lo stesso vincolo è il seguente:

$$\sum_{e \in E \,:\, {x_e}^H = 1} x_e \;\geq\; n - k$$

dove $k = 2, ..., 20$ rappresenta i gradi di libertà di CPLEX nel calcolare la nuova soluzione.
Ad ogni iterazione dell'algoritmo viene aggiunto un nuovo vincolo di *Local Branching*, basato sull'attuale soluzione restituita da CPLEX, e viene rimosso il vincolo inserito nel modello all'iterazione

precedente.
Non scegliendo in maniera randomica i lati da selezionare, come invece accade nell'Hard Fixing, se non dovesse esserci alcun miglioramento del costo e quindi cambiamento della soluzione, i lati selezionati da CPLEX con il nuovo vincolo sarebbero gli stessi dell'iterazione precedente. Per ovviare tale problema, il valore di k viene inizializzato a 2 e, nel caso in cui non dovesse migliorare la soluzione, viene incrementato.
Da dati sperimentali, si evince come questo metodo aiuti CPLEX a convergere più velocemente alla soluzione ottima e che valori di k superiori a 20 non aiutino ad ottenere risultati migliori.
Normalmente per poter analizzare lo spazio delle soluzioni è necessario enumerare tutti gli elementi al suo interno. Grazie all'aggiunta di un vincolo di *Local Branching* è, invece, possibile eseguire quest'operazione in maniera più semplice e veloce.
Definita una soluzione intera ammissibile x^H e utilizzando la distanza di Hamming, le soluzioni $k-opt$, rispetto ad x^H, sono quelle che hanno distanza k da essa (vedi Figura 2.4).

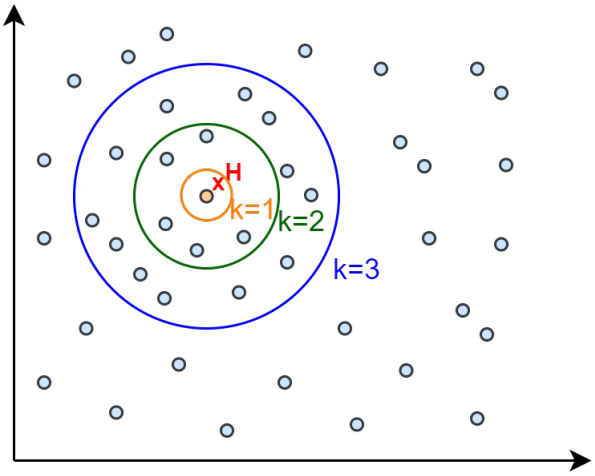

Figura 2.4: Spazio delle soluzioni e distanza di Hamming.

Utilizzando l'enumerazione delle soluzioni per **k** generico, dovrebbero essere generate circa n^k soluzioni a distanza k da x^H. In seguito dovrebbero essere analizzate tutte per individuare quella con costo minore di x^H.
L'utilizzo del *Local Branching* può essere adottato con anche problemi generici e non solo con il TSP. Data una soluzione approssimata (euristica) x_H di un problema di ottimizzazione, la formulazione matematica che permette di generare soluzioni a distanza minore o uguale di R, tramite Local Branching, è la seguente:

$$min\{c^T x: \quad Ax = b,\ x \in \{0,1\}^n\} \tag{2.16}$$

$$\sum_{j \in E: {x_j}^H = 0} x_j \ + \sum_{j \in E: {x_j}^H = 1} 1 - x_j = \leq R \tag{2.17}$$

dove (2.17) rappresenta la distanza di Hamming della nuova soluzione x ottenuta da x_H.
L'obiettivo del Soft Fixing è cercare di migliorare il costo della soluzione, cercando di analizzare le soluzioni più vicine nello spazio. Nella Figura 2.5 viene riportato un esempio di una possibile evoluzione dell'algoritmo nella ricerca dell'ottimo.

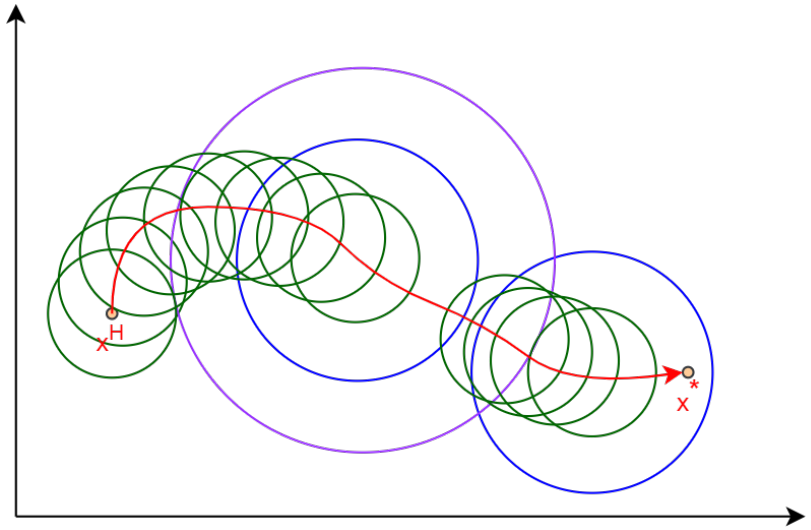

Figura 2.5: Esempio di esecuzione dell'algoritmo nello spazio delle soluzioni.

Capitolo 3

Algoritmi euristici

In questo capitolo verranno trattati algoritmi euristici che non fanno uso di CPLEX. La necessità di non utilizzare CPLEX si ha per istanze con un elevato numero di nodi per cui la risoluzione del tableau diventerebbe un'operazione molto onerosa per via dell'alto numero di variabili coinvolte.
Attraverso gli algoritmi euristici, viene computata un'approssimazione della soluzione ottima che spesso può essere anche sfruttata inizialmente dal risolutore CPLEX, per aiutarlo nello risoluzione di problemi anche più complessi.
Ad esempio può essere aggiunta prima della computazione, utilizzando la funzione *CPXaddmipstarts()*, o ,se già definita, può essere modificata tramite *CPXchgmipstarts()*.
Per progettare un efficiente algoritmo euristico, questo deve essere composto da due fasi che si alternino:

- **Intensificazione o raffinamento**
  In questa fase la soluzione corrente viene migliorata fino al raggiungimento di un ottimo (locale o globale) nello spazio delle soluzioni.

- **Diversificazione**
  Fase in cui la soluzione viene perturbata con una politica predefinita affinché si allontani da un ottimo locale nello spazio delle soluzioni.

3.1 Algoritmi di costruzione

Questa tipologia di algoritmi euristici è fondamentale per la computazione di una prima soluzione ammissibile del problema.

3.1.1 Nearest Neighborhood

L'algoritmo Nearest Neighborhood è basato su un approccio di tipo greedy. Inizialmente viene selezionato un nodo generico tra quelli che compongono il grafo ed in seguito vengono aggiunti iterativamente degli archi del grafo, secondo il criterio enunciato nella seguente sezione.
In ciascuna iterazione vengono analizzati gli archi uscenti dal nodo selezionato all'iterazione precedente e viene selezionato tra questi il ramo collegato all'estremo più vicino. Il nuovo nodo raggiunto verrà impostato come punto di partenza nell'analisi dei costi dell'iterazione successiva (vedi Figura 3.1). All'ultima iterazione viene selezionato invece l'arco che collega l'ultimo nodo visitato al nodo scelto randomicamente all'inizio dell'algoritmo.
Il problema di questo algoritmo è che in ogni iterazione viene selezionato esclusivamente il vertice più vicino a quello scelto precedentemente, senza però cercare di prevedere e migliorare la futura evoluzione del costo del tour, creato dall'algoritmo.
Come in Figura 3.1, la scelta dell'arco di costo minimo non implica che in seguito venga generata la soluzione ottima. La scelta del nodo iniziale è fondamentale in quanto una perturbazione della partenza genera un tour differente.

Definito **n** come il numero di nodi presenti nel grafo, si avranno quindi n soluzioni differenti, ottenute ciascuna attraverso l'applicazione dell'algoritmo partendo da una diversa scelta iniziale.

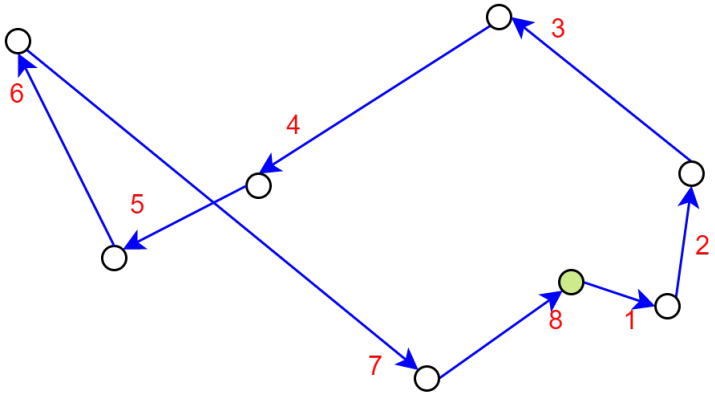

Figura 3.1: Esempio di esecuzione di Nearest Neighborhood.

3.1.2 Heuristic Insertion

L'algoritmo seguente usa un approccio simile al precedente ma prevede inizialmente la selezione di un ciclo, a cui verranno apportate modifiche per ottenere una soluzione iniziale ammissibile del problema. Per definire il ciclo di partenza vengono utilizzati diversi metodi. Di seguito sono riportati i due più utilizzati:

- **Selezione di due nodi**
  Vengono scelti i due nodi più lontani tra loro nel grafo, o due nodi casuali, e vengono collegati mediante i due archi orientati, che li connettono tra loro.
- **Inizializzazione geometrica**
  Nel caso in cui i nodi del grafo appartengano ad uno spazio bidimensionale, ne viene calcolata la convex-hull e questa viene utilizzata come ciclo di partenza.

Questa prima soluzione viene modificata iterativamente e per ogni coppia di nodi non appartenenti al ciclo **C**, restituito dall'iterazione precedente, viene calcolato l'extramileage Δ_h come segue:

$$\Delta_h = \min_{(a,b)\in C} (c_{ah} + c_{hb} - c_{ay})$$

con c_{ij} = costo dell'arco che collega i a j (vedi Figura 3.2).
Alla fine di ciascuna iterazione viene aggiunto nel grafo il nodo **k** che minimizza l'**extra-mileage** (vedi Figura 3.3):

$$k = \underset{h}{argmin}\ \Delta_h$$

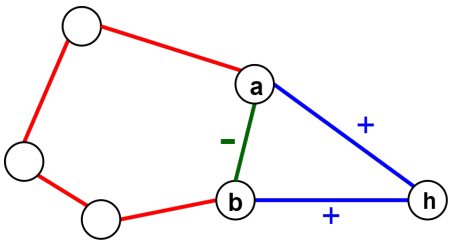

Figura 3.2: Parte del calcolo dell'extramileage del nodo **h**.

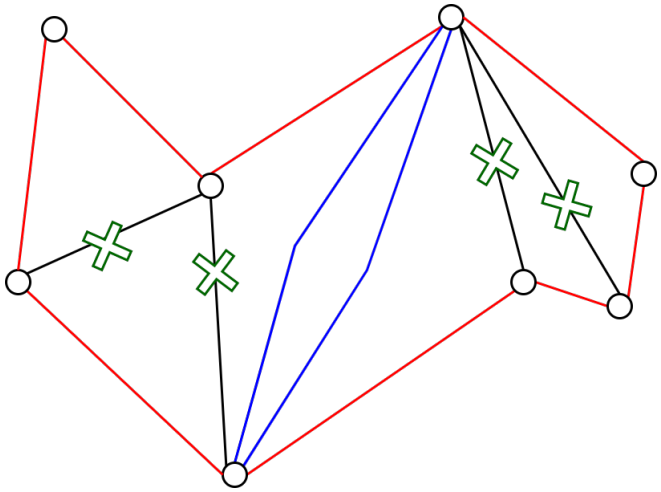

Figura 3.3: Esempio dell'applicatione di Heuristic insertion.

3.1.3 GRASP

Il metodo Greedy Randomized Adaptive Search Procedure (GRASP) è un approccio algoritmico che permette di aggiungere una componente aleatoria alla computazione deterministica del minimo di un insieme di valori.
Ad ogni iterazione dei due precedenti algoritmi di costruzione, invece di selezionare l'arco di costo minimo o l'extra-mileage minimo, vengono memorizzati i rami di costo minore e le scelte con extra-mileage minore.
Tra le possibili mosse salvate, ne viene scelta randomicamente una. Nel programma sviluppato, oltre ai precedenti algoritmi di costruzione, ne sono state implementate anche delle varianti che fanno uso del GRASP. In questo caso sono state memorizzate le tre scelte migliori in ogni iterazione.
Tali varianti permettono di modificare in maniera aleatoria l'evoluzione del tour, in modo da evitare che, nelle ultime iterazioni dell'algoritmo, le scelte possibili siano legate esclusivamente ad elevati incrementi della funzione obiettivo.
Ciò evita ad esempio che nel Nearest Neighborhood possano esserci numerose scelte come l'ultima effettuata in Figura 3.1.
Nella Sezione 4.4.3.1, vengono confrontati tramite performance profile, gli algoritmi precedentemente nominati.

3.2 Algoritmi di raffinamento

Una volta ottenuta una prima soluzione è necessario migliorarla per avvicinarsi il più possibile all'ottimo. Gli algoritmi utilizzati con questo scopo sono detti *algoritmi di raffinamento*. Nel capitolo precedente sono già stati descritti due procedimenti di questo tipo, l'Hard Fixing e il Soft Fixing (vedi sottosezioni 2.2.1 e 2.2.2). In questa sezione verranno invece analizzati algoritmi di raffinamento che non utilizzino funzioni messe a disposizione da CPLEX.

3.2.1 Algoritmo di 2-ottimalità

Nelle soluzioni restituite dagli algoritmi euristici di costruzione sono spesso presenti incroci tra coppie di rami. La loro presenza implica che la soluzione non è ottima, in quanto per le proprietà dei triangoli esisterà sempre una tour che eviti l'incrocio e che sia di costo minore. Infatti, in

riferimento alla Figura 3.4,

$$\begin{cases} ad < \alpha + \delta \\ cb < \gamma + \beta \end{cases} \Rightarrow ad + cb < \alpha + \delta + \gamma + \beta$$

.

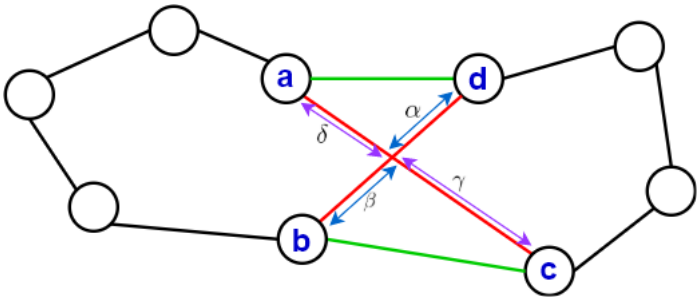

Figura 3.4: Non ottimalità di una soluzione con incroci.

L'algoritmo di 2-ottimilità prende il nome dalla modalità utilizzata iterativamente per modificare la soluzione ricevuta in ingresso. In ogni iterazione viene individuato un incrocio tra due rami, appartenenti al tour. Gli estremi di tali archi vengono collegati in maniera differente. Complessivamente per ogni incrocio, viene effettuato uno scambio tra coppie di rami (2-opt move) in modo da ridurre ulteriormente il costo della soluzione restituita.
Nell'implementare tale algoritmo non è necessario individuare ciascun incrocio della soluzione di partenza ma è sufficiente analizzare tutte le coppie di rami presenti e verificare se, scambiandole con un'altra coppia ammissibile, si verifichi un miglioramento della funzione obiettivo.

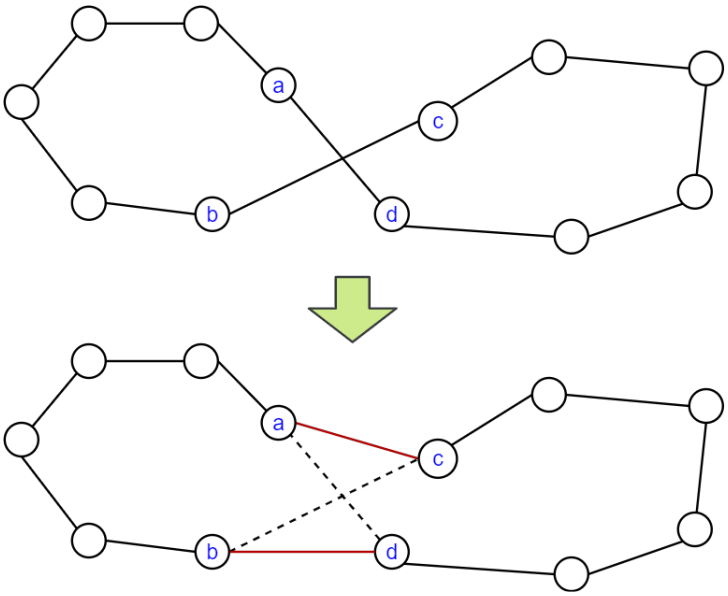

Figura 3.5: Esempio di eliminazione di un incrocio.

Riferendosi alla Figura 3.5, un possibile miglioramento viene calcolato come segue:

$$\Delta = (c_{ac} + c_{bd}) - (c_{ad} + c_{bc})$$

e solo nel caso in cui Δ sia negativo, la sostituzione viene effettuata.
Applicando una 2-opt move al circuito attuale, viene generato un tour appartenente all'intorno di

2-ottimalità della precedente soluzione. Ripetendo iterativamente tale procedimento si raggiunge un ottimo locale, in cui non esistono più possibili miglioramenti della funzione obiettivo. Questo processo è rappresentato in Figura 3.6.

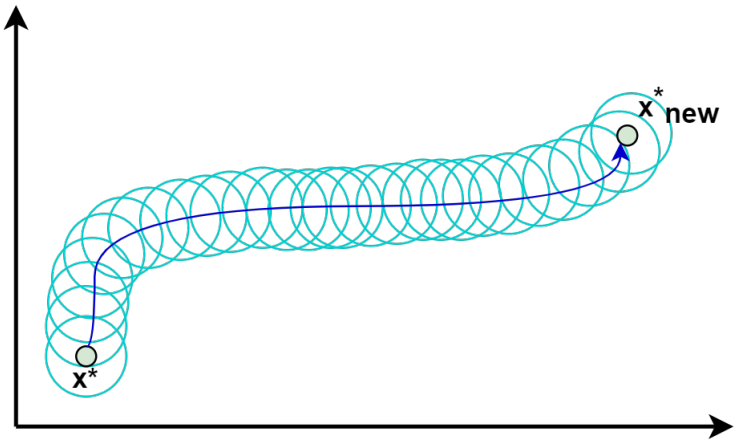

Figura 3.6: Aggiornamento della soluzione nell'intorno di 2 ottimalità.

Poichè il calcolo del Δ avviene in tempo costante e deve essere svolto per ogni coppia di rami, il tempo complessivo per la computazione è $O(n^2)$, con n pari al numero di nodi dell'istanza.
Un procedimento analogo a tale algoritmo viene utilizzato anche nel Soft Fixing, in cui però la dimensione dell'intorno in cui cercare la nuova soluzione varia (vedi Figura 2.5).
Nel programma sviluppato, è stata utilizzata questo tecnica per rimuovere gli incroci all'interno di un tour.

3.2.2 Algoritmo di 3 ottimalità

L'algoritmo di 3 ottimalità è analogo a quello analizzato nella sezione precedente, ma considera intorni di grandezza maggiore. In questo caso, quindi, due soluzioni nell'intorno differiscono per 3 rami. In Figura 3.7 viene riportata la rappresentazione delle possibili 3-opt move.

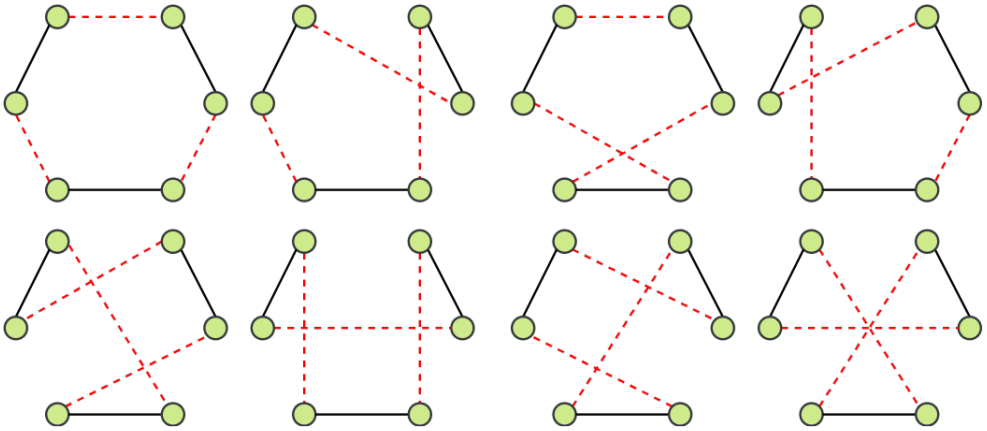

Figura 3.7: Tutte le possibili combinazioni di scambi di 3 ottimalità.

L'algoritmo impiega in tutto $O(n^3)$ (con n numero di nodi) per trovare un ottimo locale, essendo $O(n^3)$ il numero di terne di rami esistenti e poiché il numero di possibili scambi di 3 ottimalità è costante. Su istanze con un elevato numero di nodi, il tempo di calcolo risulterebbe essere troppo

lungo per riuscire a computare una soluzione. All'interno del programma sviluppato, non è stata implementato tale algoritmo.

3.3 Meta-euristici

Gli algoritmi di raffinamento appena visti si occupano di migliorare il più possibile una soluzione già calcolata, attraverso meccanismi di local search. In questo modo, dopo un determinato numero di iterazioni, viene raggiunto un nuovo ottimo locale.
Gli algoritmi descritti in questa sezione perturbano la soluzione allontanandola dall'attuale ottimo locale e cercando di avvicinarsi il più possibile all'ottimo globale.
Questi metodi rappresentano approcci più generali di quelli descritti precedentemente e sono applicabili anche a problemi differenti rispetto al TSP. Tali tecniche infatti permettono essenzialmente di esplorare lo spazio delle soluzioni, evitando di stazionare in minimi o massimi locali con valori della funzione obiettivo molto lontani dall'ottimo globale.

3.3.1 Multi-start

Un primo e intuitivo approccio per allontanarsi da un ottimo locale è quello descritto dalla politica multi-start. Questa consiste nel definire diverse soluzioni attraverso un algoritmo di costruzione tra quelli descritti, ad esempio, nella Sezione 3.1.
A ciascuna di esse viene poi applicato un algoritmo di raffinamento ed in seguito viene scelta solo la soluzione con costo minore tra quelle generate. In questo modo vengono analizzati diversi ottimi locali nello spazio delle soluzioni (vedi Figura 3.8).

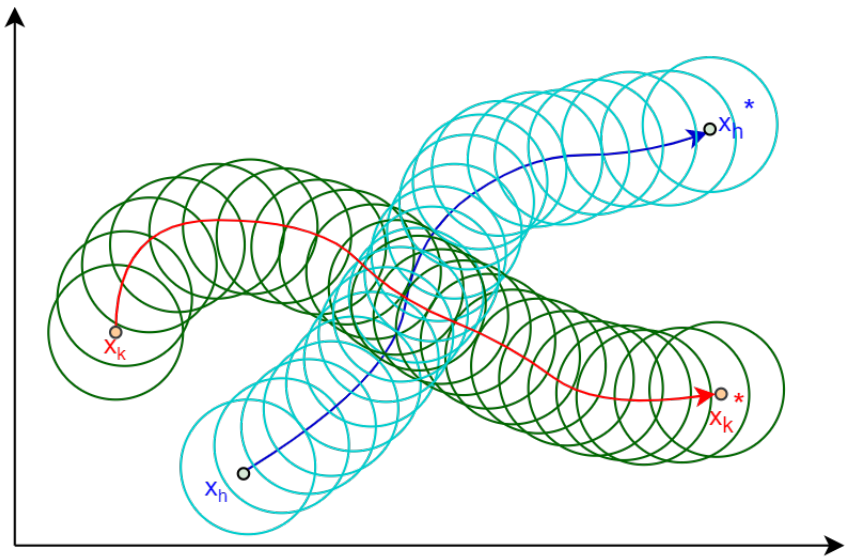

Figura 3.8: Due possibile esecuzioni di un algoritmo di raffinamento con partenze da due soluzioni diverse.

Un lato negativo di tale approccio consiste nel fatto che ogni volta che viene generata una nuova soluzione del problema, vengono perse le informazioni relative ai generati in precedenza. La soluzione implementata all'interno del programma è basata sul multithreading. Infatti viene generato un numero di soluzioni pari al numero di thread specificati.
Ciascun thread genera parallelamente agli altri una nuova soluzione e solo al termine della computazione, la confronta con la migliore. Nel caso in cui il costo della soluzione trovata sia inferiore a questa, la aggiorna.
Nella Sezione 4.4.3.1 viene analizzato il costo ottenuto dalla nostra implementazione, utilizzando 12 thread e modificando l'algoritmo di costruzione utilizzato.

3.3.2 Variable Neighborhood Search

Il Variable Neighborhood Search (VNS) è un algoritmo che cerca di migliorare l'ottimo locale attuale, ricevuto in input, analizzando intorni di ottimalità di grandezza differente.
Nel caso in cui non si trovi una nuova soluzione, di costo migliore di quella attuale nei vari intorni di grandezza k, l'algoritmo prevede che venga scelto un certo numero di rami in maniera randomica da sostituire con altri lati[8]. In questo modo si impone un aggiornamento peggiorativo in termini di costo, nella speranza che nel nuovo intorno selezionato sia possibile trovare una soluzione che si allontani dall'ottimo locale di partenza (vedi Figura 3.9).

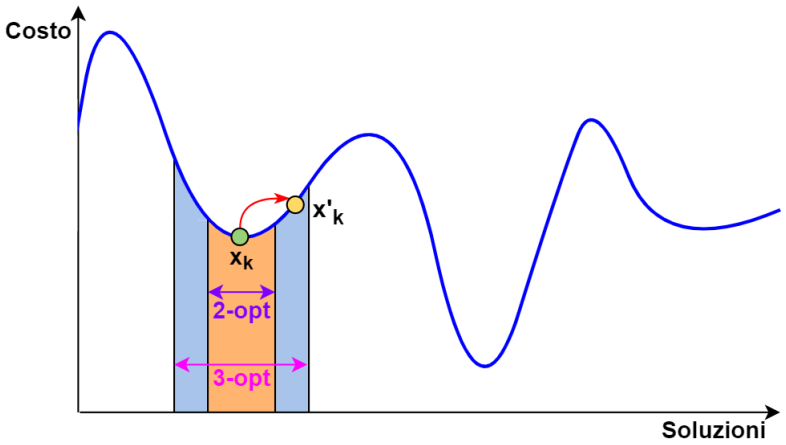

Figura 3.9: Aggiornamento di una soluzione in un minimo locale.

L'algoritmo termina allo scadere di una deadline, inserita dall'utente, o dopo un determinato numero di iterazioni, restituendo la miglior soluzione trovata fino a quell'istante. Utilizzando questo approccio gran parte della soluzione di partenza viene conservata evitando di perdere le informazioni elaborate precedentemente eseguendo l'algoritmo.
La nostra implementazione dell'algoritmo VNS non è però quella classica ma una variante detta VNS ibrido[12]. Nell'Algoritmo 4 viene descritto lo pseudo-codice di tale metodo.
Partendo dalla sequenza di visita dei nodi nella soluzione locale, passata in ingresso all'algoritmo, la procedura analizza tutti i tour ottenuti invertendo due nodi a distanza k nella sequenza, con $k = 1, ..., n/2$. Se viene individuata una soluzione con costo inferiore, allora questa viene considerata come il nuovo ottimo locale. Invece se non dovesse esistere alcuno scambio migliorativo, viene scelta in maniera randomica una nuova soluzione dall'insieme:

$$\left\{ \{y\} \cup \bigcup_{k=1}^{\lceil n/2 \rceil} N_k(y) \right\}$$

con probabilità pari a:

$$p = \frac{costo^{-1}(x)}{\sum_{z \in \bigcup_{k=1}^{\lceil n/2 \rceil} N_k(y)} costo^{-1}(z)}$$

Nell'implementazione sono state sviluppate due versioni di questo algoritmo, in base alle probabilità utilizzate nel selezionare due nodi da scambiare:

- **Hybrid VNS**
  questa prima soluzione segue la procedura precedentemente descritta e seleziona una nuova sequenza dall'insieme $\{y \cup \bigcup_{k=1}^{\lceil n/2 \rceil} N_k(y)\}$ e poi vi applica un algoritmo di raffinamento.

- **Hybrid VNS Uniform**
  questa variante invece seleziona due nodi della sequenza di visita, secondo una probabilità uniforme, e li scambia. La sequenza risultante, dopo essere stata migliorata mediante un algoritmo di raffinamento, viene utilizzata come nuovo minimo locale.

Algoritmo 4: VNS ibrido

Input: $\mathtt{x} = \{\mathtt{x}_1, \mathtt{x}_2, ..., \mathtt{x}_n\}$= sequenza di visita dei nodi nella soluzione di partenza (ottimo locale)

Output: x= soluzione migliore aggiornata dall'algoritmo

```
1  while 'deadline not expired' ∧ count < MAX_COUNT do
2      y ←x
3      k ←1
4      while k ≤ ⌈n/2⌉ do
5          N_k(y) = {(y_1, ..., y_{(i+k) mod n}, ..., y_i, ..., y_n) : 1 ≤ i ≤ n}
6          if min_{z∈N_k(y)} {cost(z)} < cost(x) then
7              x ←argmin_{z∈N_k(y)} (cost(z))
8              y ←argmin_{z∈N_k(y)} (cost(z))
9              break
10         k ← k + 1
11     if k > ⌈n/2⌉ then
12         y ← new_random_sol(y)
13     count ← count + 1
```

3.3.3 Tabu Search

Il metodo Tabu Search fu ideato da Fred W. Glover. Dato un ottimo locale, l'idea di Glover permette di modificare la soluzione corrispondente anche peggiorandone il costo, con l'obiettivo di esplorare maggiormente lo spazio delle soluzioni. Ad ogni iterazione dell'algoritmo la soluzione viene modificata con un nuovo tour, appartenente al suo intorno 2-ottimalità.
Per evitare che queste modifiche portino ad esplorare minimi locali già visitati, viene creata una lista di "mosse vietate", detta *Tabu List*. In questo modo il costo del tour attuale viene aumentato per un certo numero di iterazioni, finché le uniche modifiche ammissibili della soluzione corrente permettano solo di migliorare il valore della funzione obiettivo. Questa fase rappresenta la diversificazione della soluzione.
In seguito viene effettuata una fase di intensificazione, mediante l'applicazione dell'algoritmo di 2-ottimalità, fino al raggiungimento di un nuovo minimo locale.
Nella nostra implementazione la lista viene riempita con i rami che vengono rimossi dal tour attuale per creare la nuova soluzione.
Aumentando costantemente di dimensione la lista Tabu si rischia, ad un certo punto, che non sia più possibile modificare il tour presente. Per evitare ciò generalmente viene scelta una capienza massima della lista, detta *tenure*. Una volta riempita la Tabu list i rami vengono aggiunti rispettando la politica FIFO (First In First Out).
Per aumentare le prestazioni dell'algoritmo è possibile far variare le dimensioni della Tabu List tra due valori. Nella fase di diversificazione le dimensioni della lista aumentano fino ad una soglia massima, mentre durante l'intensificazione la tenure diminuisce fino ad un valore minimo fissato, per lasciare maggiore libertà all'algoritmo di 2-ottimalità. Nella nostra implementazione la tenure

massima ha valore pari ad 1/5 del numero di nodi dell'istanza da risolvere, mentre quella minima è pari alla metà di quest'ultima (vedi Figura 3.10). Questa variante dell'algoritmo, implementata anche nel nostro programma, in letteratura è conosciuta con il nome di *Reactive Tabu Search*.

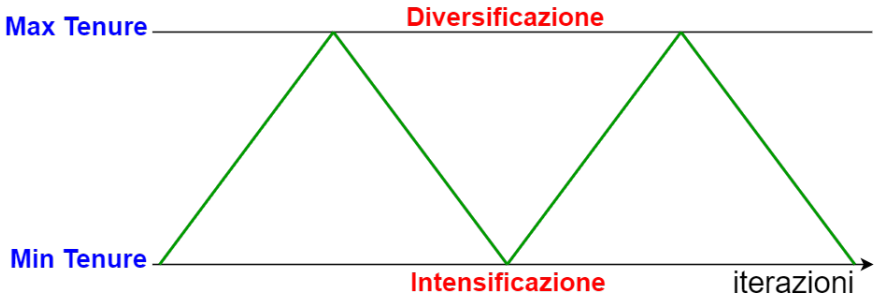

Figura 3.10: Variazione della tenure.

Come per l'algoritmo precedente, il criterio di terminazione è dato dallo scadere del tempo a disposizione o dal raggiungimento di un numero massimo di iterazioni. La soluzione restituita dall'algoritmo è la migliore individuata fino a quell'istante.

Algoritmo 5: Tabu Search

Input: `x`= soluzione di un'istanza di TSP corrispondente ad un ottimo locale
`tenure`= dimensione massima della Tabu List
`deadline`= time limit complessivo dell'algoritmo
`num_iterations`= numero massimo di iterazioni
`num_nodi`= numero di nodi dell'istanza tsp

Output: `y`= miglior soluzione trovata

```
1  n = 0
2  while n < num_iterations ∧ 'deadline not expired' do
3      x' ← move_random_2opt(x)
4      while (check_tabu_list(x') == 'valid move') do
5          x' ← move_random_2opt(x)
6      x ← x'
7      add_tabu_list(edges_removed, tenure)
8      if cost(x') < cost(x) then
9          x ← greedy_refinement(x, tabu_list)
10     n ←n + 1
11 y = best_solution()
```

3.3.4 Simulated Annealing

L'algoritmo Simulated Annealing si ispira al processo di temperamento dei metalli, in cui un materiale viene raffreddato molto lentamente e in maniera controllata, affinché raggiunga la configurazione di minima energia[9]. Analogamente, nell'algoritmo viene scelta una funzione, generalmente esponenziale, che simuli la variazione della "temperatura" T.
Nel corso delle iterazioni, la soluzione corrente può essere aggiornata con un'altra qualsiasi nel suo intorno di 2 ottimalità (Figura 3.11). La probabilità di accettare un nuovo tour peggiorativo è una funzione $f(\Delta costo, T)$ che dipende dalla differenza, tra il costo della soluzione attuale e quello della

nuova candidata, e dal valore corrente della temperatura. Nel caso in cui avvenga un aumento del costo complessivo della soluzione, la temperatura viene decrementata. In questo modo si riduce la probabilità di accettare un aggiornamento peggiorativo durante l'iterazione successiva.

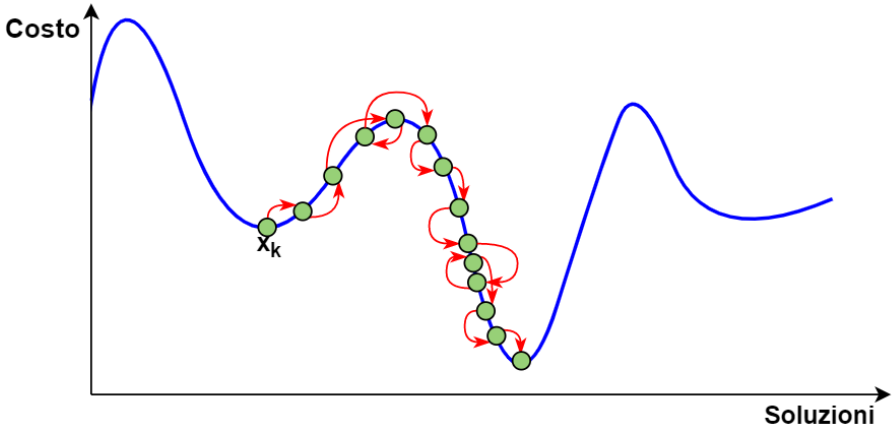

Figura 3.11: Esempio di esecuzione dell'algoritmo Simulated Annealing.

Per variare la temperatura, è stata utilizzata la seguente formula [9]:

$$T = \alpha^i \, T\_max \; + \; T\_min$$

dove T_max è la massima temperatura iniziale, T_min è la minima temperatura che può essere raggiunta, i è il numero di iterazioni eseguite fino a quel momento e α è un parametro scelto all'inizio dell'esecuzione. Nello sviluppare questo procedimento sono stati utilizzati i seguenti valori:

- $T\_max = 5000$
- $T\_min = 100$
- $\alpha = 0.99$

Una volta raggiunta la temperatura minima, nel caso in cui non sia scaduto il tempo a disposizione, T viene impostata nuovamente a T_max e il numero di iterazioni viene resettato, per poter continuare la computazione sfruttando l'intera deadline fornita dall'utente.
La funzione di probabilità da noi adottata nello sviluppo di quest'algoritmo è la seguente[9]:

$$P = \begin{cases} 1 & se \; \Delta costo \leq \; 0 \\ exp(-\frac{\Delta costo}{T}) & se \; \Delta costo \; > \; 0 \end{cases}$$

Per poter gestire al meglio valori della probabilità troppo vicini allo zero, $exp(\Delta costo/T)$ viene fattorizzato in notazione scientifica, analizzando il rapporto tra la variazione di costo e la temperatura.
Per questo motivo una soluzione peggiore viene accettata solo se si verificano entrambe le condizioni:

- viene estratto randomicamente il numero 1 dall'insieme $\{1, ..., 10\}$ per un numero di volte pari all'ordine di grandezza della quantità fattorizzata;
- viene estratto randomicamente 0 nell'intervallo $[0, mantissa)$.

Con il procedere delle iterazioni, l'aggiornamento ad un costo peggiore avviene sempre meno di frequente. Ciò permette di ottenere solo aggiornamenti con soluzioni più vantaggiose. Esiste un teorema secondo il quale se la temperatura varia in maniera estremamente lenta e si dispone di un numero estremamente elevato di iterazioni, questo algoritmo garantisce di trovare l'ottimo globale. Concretamente queste ipotesi sono molto difficili da realizzare, ma è statisticamente possibile dichiarare che l'approccio del Simulated Annealing restituisca una buona soluzione.

Algoritmo 6: Simulated Annealing

Input: x=soluzione di un'istanza di TSP corrispondente ad un ottimo locale
T_{min}=temperatura minima
T_{max}=temperatura massima
deadline=time limit complessivo dell'algoritmo

Output: y= miglior soluzione trovata

```
1  T ← T_min
2  while 'deadline not expired' do
3      x' ← move_random_2opt(x)
4      if cost(x') < cost(x) then
5          x' ← x
6          update_cost()
7      else
8          compute_prob(cost(x'), T)
9          if accepted(x') then
10             x' ← x
11             update_cost()
12             update_temperature(T)
13 y = best_solution()
```

3.3.5 Algoritmo genetico

L'algoritmo genetico è legato alla teoria dell'evoluzione di Darwin, con cui condivide numerosi concetti. Da un punto di vista teorico, l'algoritmo crea inizialmente una serie di individui, che costituiscono la popolazione.
In seguito, attraverso mutazioni dei singoli soggetti e la riproduzione di questi, viene creata una nuova popolazione. Il concetto fondamentale alla base di tale algoritmo è che associando ad un individuo uno punteggio, detto fitness, lo si possa selezionare con una certa probabilità per far evolvere la specie.
Applicando tale concetto al Travelling Salesman Problem, ad ogni individuo i viene associata una fitness pari a:

$$fitness_i = \frac{1}{costo_i}$$

dove $costo_i$ rappresenta il valore della funzione obiettivo per l'istanza i.
La popolazione iniziale è stata generata utilizzando l'algoritmo nearest neighbour ma utilizzando un diverso nodo di partenza per ciascuno di essi ed applicando anche la variante GRASP. Ogni individuo della popolazione è stato poi rappresentato come la sequenza di visita dei nodi della corrispondente soluzione.
In fase di testing, abbiamo notato che la creazione della popolazione costituisca il vero collo di bottiglia dell'intero algoritmo. Per ridurre al minimo il carico computazionale di tale fase, questa è stata implementata in multithreading.
I risultati sono complessivamente migliorati, anche se questa fase resta ancora un'operazione molto costosa in termini di tempo di calcolo. In seguito la fase di evoluzione della popolazione viene effettuata generando nuovi individui a partire dalla popolazione attuale ed utilizzando le seguenti tecniche:

- **Crossover**
  in questa operazione vengono selezionati in maniera randomica due individui della popolazione e a partire da questi, vengono creati nuovi tour che ereditano dai genitori delle caratteristiche. Nel nostro caso ciascun nuovo individuo eredita metà della sequenza di visita da uno dei suoi genitori, e la restante parte viene ereditata dall'altro.

- **Mutazione**
  in questa fase un certo numero di individui viene selezionato in maniera casuale e da ciascuno di questi, viene generato un nuovo tour attraverso una sua permutazione randomica.

Utilizzando iterativamente tali tecniche e rimuovendo gli individui di costo peggiore dalla popolazione, si riduce complessivamente il costo medio delle istanze nella popolazione e di conseguenza anche il costo della migliore soluzione. All'interno del programma sviluppato, le precedenti tecniche non sono state applicate contemporaneamente in ogni iterazione ma in istanti differenti, secondo quanto descritto nel seguente algoritmo.

Algoritmo 7: Evoluzione

Input: `population`= popolazione di numerosi tour generati mediante un algoritmo di costruzione

Output: `y`= sequenza di visita dei nodi nel tour ottenuto di costo minore

```
1 num_epochs ← 0
2 best_cost ← 0
3 while num_epochs < MAX_NUM_EPOCHS do
4     if num_epochs mod 5 == 0 then
5         crossover(population, best_cost)
6     else
7         mutation(population, best_cost)
8     remove_worst_members(population)
9 y ← best_member(population)
```

La fase di crossover viene applicata più di rado, poiché la generazione di nuovi figli a partire dai genitori richiede molto più tempo della creazione dello stesso numero di individui mediante mutazione. Gli algoritmi di crossover e mutazione, utilizzati nel programma sviluppato, sono descritti e illustrati di seguito.

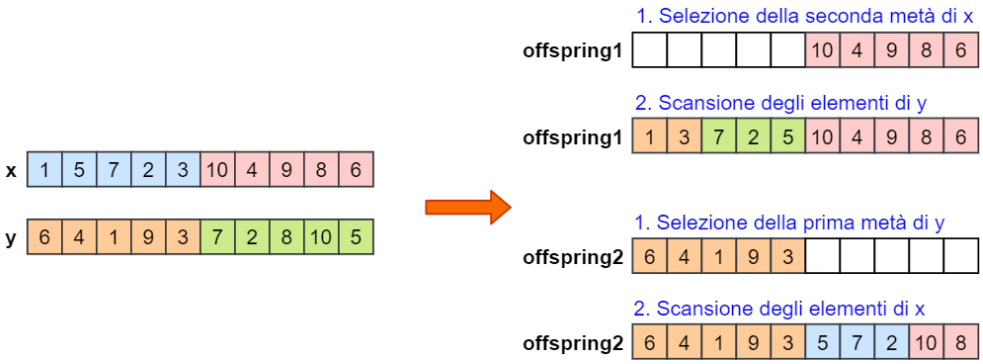

Figura 3.12: Esempio di applicazione del crossover su due istanze x e y.

Algoritmo 8: Crossover

Input: `population`= popolazione di numerosi tour generati mediante un algoritmo di costruzione
`sum_fitnesses`= somma delle fitness di tutti gli individui della popolazione
`best_cost`= costo della migliore soluzione attuale

Output: $\texttt{offspring}_1, \texttt{offspring}_2$= nuovi individui generati nella popolazione

1 $\texttt{n} \leftarrow$ numero di nodi dell'istanza tsp
2 $\texttt{x} = \{\texttt{x}_1, ..., \texttt{x}_\texttt{n}\} \leftarrow \texttt{RANDOM}(population, p = \frac{fitness_x}{sum\_fitnesses})$
3 $\texttt{y} = \{\texttt{y}_1, ..., \texttt{y}_\texttt{n}\} \leftarrow \texttt{RANDOM}(population, p = \frac{fitness_y}{sum\_fitnesses})$
4 $\texttt{offspring}_1[1, ...\texttt{n}/2] = x[1, ..., n/2]$
5 $\texttt{offspring}_2[\texttt{n}/2 + 1, ..., \texttt{n}] = y[n/2 + 1, ..., n]$
6 $\texttt{offspring}_1[\texttt{n}/2 + 1, ..., \texttt{n}] = \{y_i \in y \ : \ y_i \notin \{x_1, ..., x_{n/2}\}\}$
7 $\texttt{offspring}_2[\texttt{n}/2 + 1, ..., \texttt{n}] = \{x_i \in x \ : \ x_i \notin \{y_{n/2+1}, ..., y_n\}\}$
8 $\texttt{sum\_fitnesses} \leftarrow \texttt{update}(sum\_fitness, offspring_1, offspring_2)$
9 $\texttt{min\_cost} \leftarrow \texttt{min}(cost(offspring_1), cost(offspring_2))$
10 **if** $cost(offspring1) < best\_cost$ **then**
11 $\quad \texttt{best\_cost} \leftarrow min\_cost$

Figura 3.13: Esempio di applicazione della mutazione su un'istanza x.

Algoritmo 9: Mutazione

Input: `population`= popolazione di numerosi tour generati mediante un algoritmo di costruzione
`sum_fitnesses`= somma delle fitness di tutti gli individui della popolazione
`best_cost`= costo della migliore soluzione attuale

Output: `offspring1`, `offspring2`= nuovo individuo generato nella popolazione

1 $\texttt{n} \leftarrow$ numero di nodi dell'istanza tsp
2 $\texttt{x} = \{\texttt{x}_1, ..., \texttt{x}_\texttt{n}\} \leftarrow RANDOM(population)$
3 $\texttt{begin} \leftarrow RANDOM(\{1, ..., n/2\})$
4 $\texttt{end} \leftarrow RANDOM(\{n/2 + 1, ..., n\})$
5 **for** $\texttt{i} \leftarrow begin$ **to** end **do**
6 $\quad \texttt{offspring}[i] \leftarrow x[end - i + begin]$
7 **if** $cost(offspring) < best\_cost$ **then**
8 $\quad \texttt{best\_cost} \leftarrow cost(offspring)$

Capitolo 4

Performance

Le metriche di confronto, utilizzate nell'analisi degli algoritmi, sono il tempo complessivo di creazione e risoluzione del modello ed il costo della soluzione ottenuta. Ciascuna modalità di risoluzione viene applicata a diverse istanze di TSPlib, con un numero differente di nodi.

4.1 Performance variabilty

Nel corso degli anni '90, gli ingegneri di CPLEX scoprirono che il tempo di risoluzione variasse significativamente in diversi sistemi operativi. Con alcune istanze, le performance migliori si avevano su UNIX mentre con altre su Windows.
Il motivo di tale comportamento venne in seguito studiato ed attribuito alla diversa scelta effettuata dai sistemi operativi nel decretare l'ordine delle variabili su cui viene svolto l'albero decisionale. Le scelte svolte inizialmente, nella definizione dei primi nodi dell'albero, si ripercuotono sulla sua successiva evoluzione.
Proprio per questo motivo, su alcune istanze, UNIX riusciva a risolvere il problema in tempo minore rispetto a Windows, mentre su altre accadeva l'opposto.
Da questi studi, evinse che il Branch and Cut è un sistema caotico e che quindi piccole variazioni delle condizioni iniziali generano grandi differenze nei risultati finali.
Per questo motivo, alcuni algoritmi prensentati in questo report, sono stati studiati al variare di alcune condizioni iniziali:

- **Random Seed**
  definisce il seme da cui CPLEX genera una sequenza di numeri pseudo-casuali (vedi Sezione B.3). Nel momento in cui CPLEX nota che diverse variabili frazionarie hanno lo stesso valore, il risolutore sceglie casualmente su quale di queste applicare il Branch.

- **Gap**
  intervallo massimo, tra il valore della migliore funzione obiettivo intera e il valore della funzione di costo del miglior nodo rimanente, che permette di decretare il raggiungimento dell'ottimo secondo CPLEX (vedi Sezione B.3).

La variazione del primo di questi parametri permette di apportare significative modifiche al tempo di risoluzione, non modficando la reale ottimalità della soluzione.
La variazione dell'altro parametro permette invece di ottenere una soluzione euristica, ovvero un'approssimazione più lasca di quella ottima.

4.2 Analisi tabulare

Un metodo non molto efficiente per lo studio delle performance degli algoritmi, utilizza una struttura tabulare in cui viene inserita una riga per ogni istanza del problema.
Inoltre vengono riportati i tempi di esecuzione degli algoritmi su ognuno dei grafi analizzati. Nell'ultima riga per ciascun algoritmo viene riportata la media geometrica dei suoi tempi di esecuzione

(vedi esempio in Tabella 4.1).
Solitamente viene impostato un TIME LIMIT uguale per tutti gli algoritmi. Questo rappresenta nella tabella il valore del tempo di esecuzione per un algoritmo che ha impiegato un tempo maggiore o uguale a TIME LIMIT. Spesso viene dato più peso al TIME LIMIT, inserendolo nella tabella con, ad esempio, peso 10 (ovvero TIME LIMIT*10).
La debolezza di tale calcolo delle performance risiede nel fatto che non sempre la media descrive l'efficienza di un soluzione. Infatti non influisce unicamente il tempo di risoluzione del modello ma anche quello necessario alla sua creazione.

Istanza	**Sequential**	**Flow**	**Loop**
att48	212.3	12.5	4.3
...	...	...	...
a280	3200	2500.8	1300.5
	2120.3	1800.3	1000.4

Tabella 4.1: Tabella di performance con **TIME LIMIT=3200**.

4.3 Performance profiling

Questo metodo prevede la classificazione dei tempi di esecuzione degli algoritmi in base al numero percentuale di successi, rispetto a un fattore moltiplicativo (ratio) del tempo di esecuzione (vedi Figura 4.1).
L'andamento del performance profile di un algoritmo è monotono crescente. Il valore assunto per ogni ratio dagli algoritmi all'interno del grafico è la percentuale del numero di istanze che l'algoritmo risolve con quel fattore rispetto all'ottimo di quel caso.
Spesso questi grafici vengono rappresentati in scala logaritmica per notare al meglio le differenze ed avere una migliore raffigurazione.
Per creare il performance profile degli algoritmi implementati, è stato utilizzato il programma

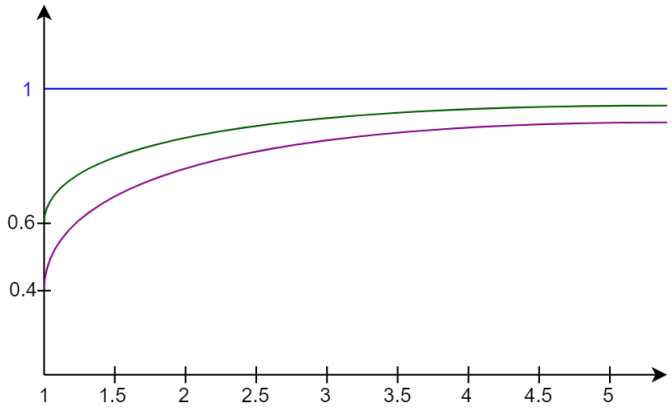

Figura 4.1: Performance profile di due algoritmi.

python riportato nella Sezione D.

4.4 Analisi degli algoritmi sviluppati

Nella seguente sezione vengono riportati i grafici relativi ai risultati ottenuti con le implementazioni degli algoritmi descritti nei precedenti capitoli. I valori ottenuti ed utilizzati per realizzare le immagini contenute in questa sezione, sono consultabili nell'Appendice E.

4.4.1 Algoritmi esatti

Gli algoritmi basati sul modello compatto, sono stati testati con un time limit di 20 minuti sulle seguenti istanze:

- att48.tsp
- berlin52.tsp
- burma14.tsp
- eil101.tsp
- eil51.tsp
- eil76.tsp
- gr96.tsp
- kroA100.tsp
- kroB100.tsp
- kroB150.tsp
- kroC100.tsp
- kroD100.tsp
- kroE100.tsp
- pr124.tsp
- pr136.tsp
- pr76.tsp
- rat99.tsp
- rd100.tsp
- st70.tsp
- ulysses16.tsp

Visionando il performance profile in Figura 4.2, risulta evidente come l'aggiunta dei vincoli come lazy constraint, nel metodo di Miller, Tucker e Zemlin, garantisca un notevole miglioramento delle prestazioni rispetto alla versione originale.

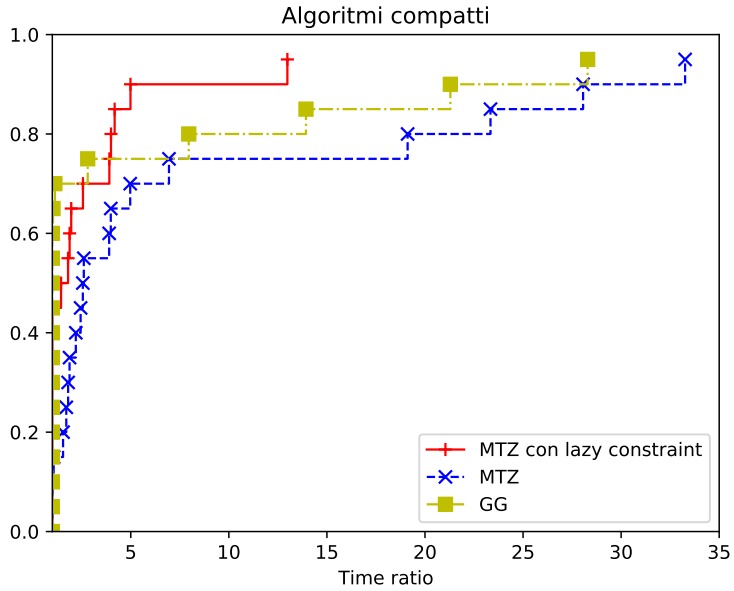

Figura 4.2: Performance profile degli algoritmi compatti.

Gli algoritmi esatti, che non prevedono l'uso di modelli compatti, sono invece stati testati con un time limit di 10 minuti sul seguente dataset:

- a280.tsp
- ali535.tsp
- att48.tsp
- att532.tsp
- berlin52.tsp
- bier127.tsp
- burma14.tsp
- ch130.tsp
- ch150.tsp
- d198.tsp
- d493.tsp
- d657.tsp
- eil51.tsp
- eil76.tsp
- eil101.tsp
- fl417.tsp
- gil262.tsp
- gr96.tsp
- gr137.tsp
- gr202.tsp

- gr229.tsp
- gr431.tsp
- gr666.tsp
- kroA100.tsp
- kroA150.tsp
- kroA200.tsp
- kroB100.tsp
- kroB150.tsp
- kroB200.tsp
- kroC100.tsp
- kroD100.tsp
- kroE100.tsp
- lin105.tsp
- lin318.tsp
- p654.tsp
- pcb442.tsp
- pr76.tsp
- pr107.tsp
- pr124.tsp
- pr136.tsp
- pr144.tsp
- pr152.tsp
- pr226.tsp
- pr299.tsp
- pr439.tsp
- rat99.tsp
- rat195.tsp
- rat575.tsp
- rat783.tsp
- rd100.tsp
- rd400.tsp
- st70.tsp

Dall'analisi dei risultati ottenuti (Figura 4.3), è evidente come l'utilizzo del Branch & Cut permetta di ottenere prestazioni migliori, ma soprattutto come l'utilizzo delle callback generiche abbia permesso di migliorare notevolmente le prestazioni, rispetto alla versione priva di esse.
L'utilizzo del patching ha avuto maggior effetto soprattutto nella prima parte dell'esecuzione del programma, in cui risulta essere molto di aiuto per CPLEX nel raggiungere l'ottimo. Successivamente le soluzioni passate al risolutore MIP, mediante le heuristic callback, vengono scartate da quest'ultimo sempre più spesso.

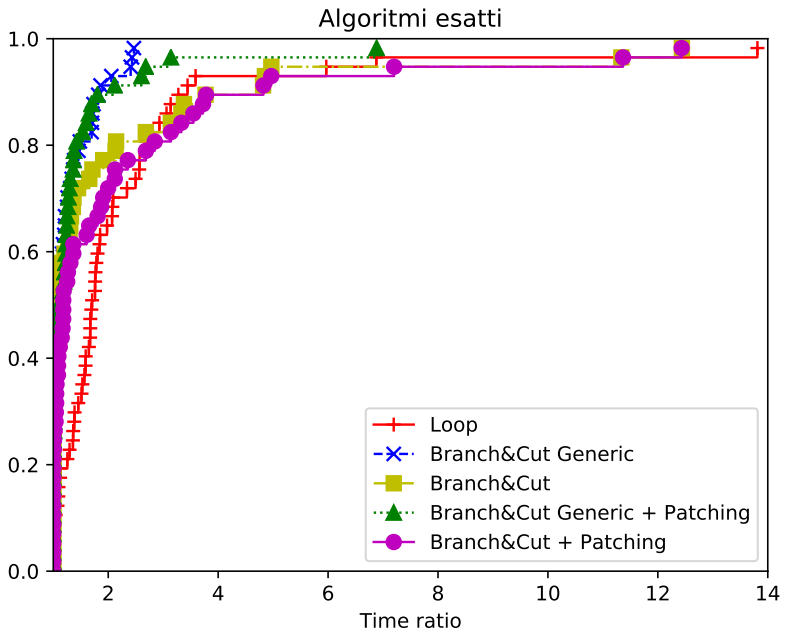

Figura 4.3: Performance profile dei tempi di esecuzione degli algoritmi esatti implementati.

In Figura 4.4 viene confrontato invece l'algoritmo loop attraverso l'utilizzo di diversi seed, ponendo l'attenzione su come la performance variability influenzi le prestazioni delle varie soluzioni implementative. Il performance profile riportato in Figura 4.5 confronta i risultati ottenuti mediante l'utilizzo dell'algoritmo loop nella sua versione classica con quelli ottenuti applicando la sua versione euristica. Nel grafico realizzato, le varianti euristiche individuano inizialmente una soluzione utilizzando il gap relativo evidenziato nel grafico ed in seguito impostano nuovamente il gap al valore di default ed individuano la soluzione ottima.

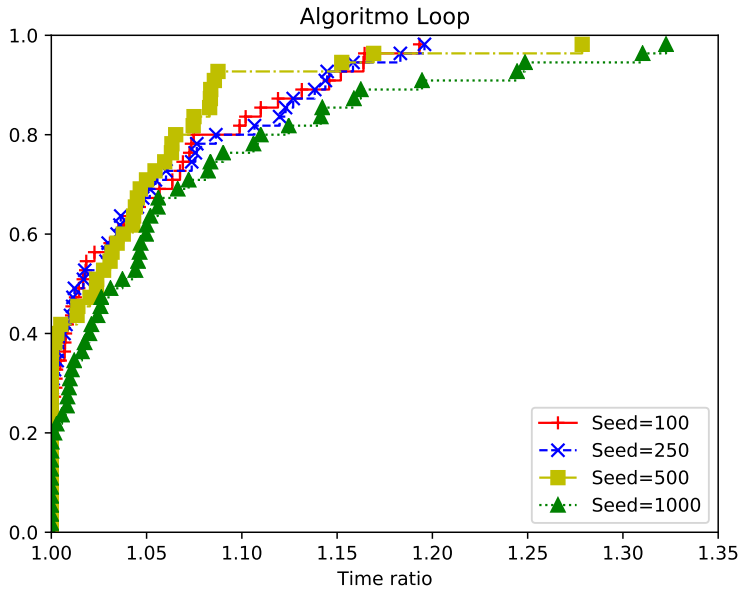

Figura 4.4: Performance profile dei tempi di esecuzione dell'algoritmo loop al variare del seed.

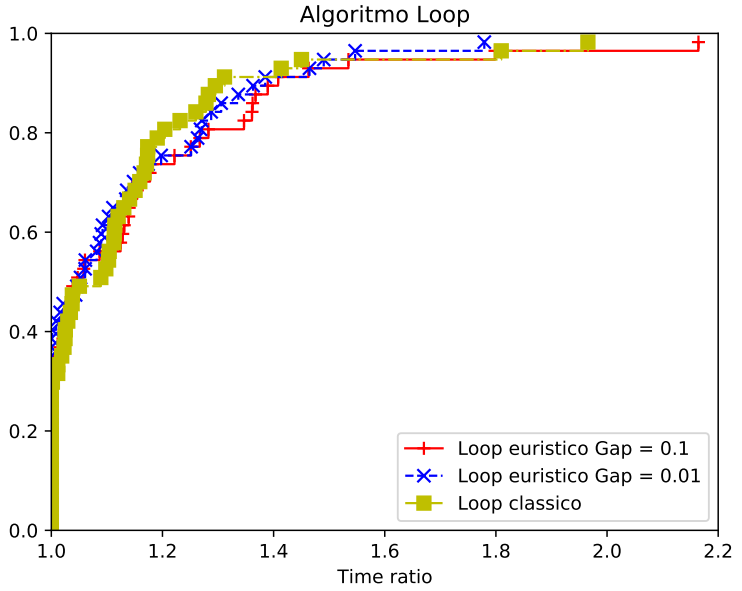

Figura 4.5: Performance profile dei tempi di esecuzione dell'algoritmo loop euristico al variare del gap.

4.4.2 Algoritmi math-euristici

Gli algoritmi math-euristici sviluppati sono stati testati sul seguente insieme di istanze, con un time limit 10 minuti:

- a280.tsp
- att532.tsp
- bier127.tsp
- d198.tsp
- d493.tsp
- eil76.tsp
- eil101.tsp
- fl417.tsp
- gr137.tsp
- gr202.tsp
- lin105.tsp
- lin318.tsp
- pcb442.tsp
- pr144.tsp
- pr264.tsp
- pr299.tsp
- pr439.tsp
- rat575.tsp
- rd400.tsp
- u159.tsp

Analizzando i risultati ottenuti in Figura 4.6 e 4.7) risulta evidente come le prestazioni dell'algoritmo Hard fixing siano migliori, sia nella variante che utilizza le callback generiche che in quella che non ne fa uso, rispetto ad entrambe le soluzioni ottenute dal Soft fixing.

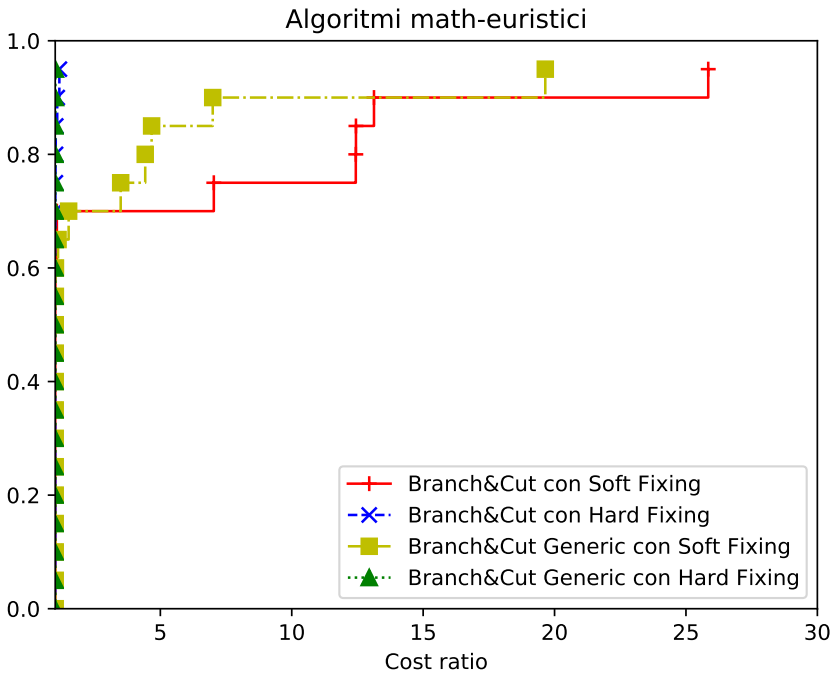

Figura 4.6: Confronto degli algoritmi math-euristici in base al costo della soluzione ottenuta.

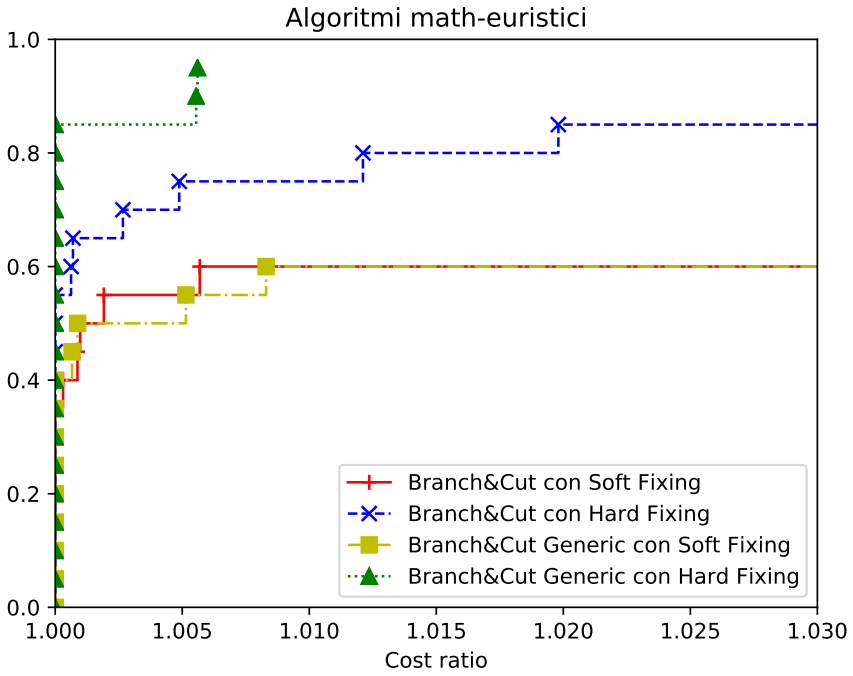

Figura 4.7: Dettaglio del confronto degli algoritmi math-euristici in base al costo della soluzione ottenuta.

4.4.3 Algoritmi euristici

4.4.3.1 Multi-start

In questa prima sezione, vengono analizzati i diversi algoritmi di costruzione implementati, applicando l'algoritmo multistart sulle seguenti istanze:

- a280.tsp
- ali535.tsp
- att532.tsp
- d1291.tsp
- d1665.tsp
- d2103.tsp
- d493.tsp
- d657.tsp
- dsj1000.tsp
- fl1400.tsp
- fl1577.tsp
- fl417.tsp
- gil262.tsp
- gr431.tsp
- gr666.tsp
- lin318.tsp
- nrw1379.tsp
- p654.tsp
- pcb1173.tsp
- pcb442.tsp
- pr1002.tsp
- pr299.tsp
- pr439.tsp
- rat575.tsp
- rat783.tsp
- rd400.tsp
- rl1304.tsp
- rl1323.tsp
- rl1889.tsp
- u1060.tsp
- u1432.tsp
- u1817.tsp
- u574.tsp
- u724.tsp
- vm1084.tsp
- vm1748.tsp

In Figura 4.8, viene riportato il performance profile ottenuto dall'esecuzione dell'algoritmo multistart, generando 40 differenti soluzioni per ciascuna istanza del problema e restituendo solo il costo della migliore tra queste.
Le soluzioni di costo minore sono quelle restituite dall'algoritmo Nearest Neighborhood, in particolare con l'aggiunta del metodo GRASP. Inoltre si è notato come il tempo necessario alla definizione di una soluzione mediante il metodo insertion sia molto elevato.

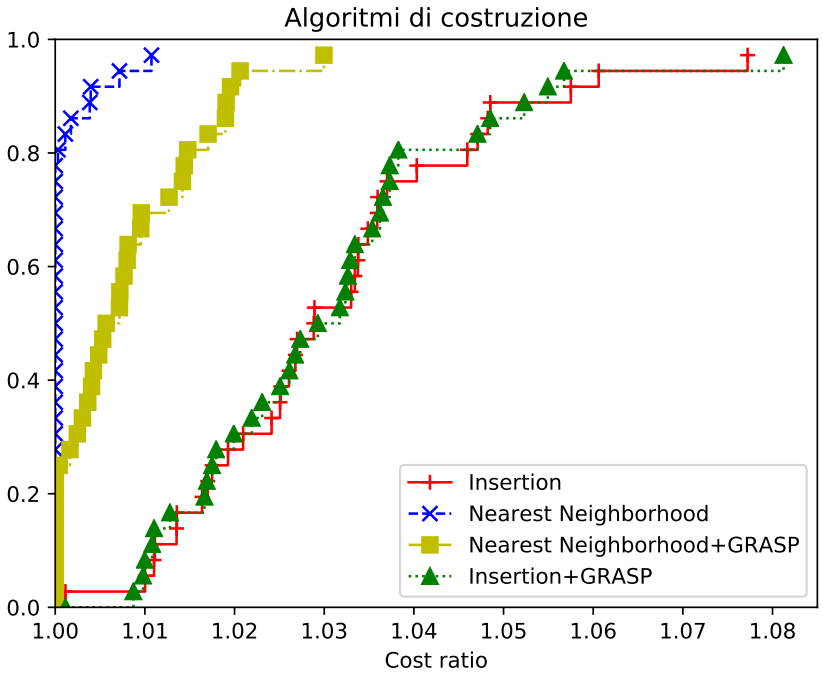

Figura 4.8: Confronto dei vari multistart in base all'algoritmo di costruzione utilizzato.

4.4.3.2 Algoritmi meta-euristici

Tutti gli algoritmi meta-euristici sviluppati, escluso il genetico, prevedono la possibilità di utilizzare uno qualsiasi tra gli algoritmi di costruzione definiti, semplicemente associando specifici valori a delle macro.
Per i motivi descritti nella sezione precedente, gli algoritmi sono stati testati però utilizzando il metodo di costruzione Nearest Neighborhood e con un time limit di 10 minuti. In questi test, è stata impostata una macro che permette al multistart di essere eseguito in multithreading in questo intervallo di tempo, generando iterativamente 8 soluzioni alla volta. Il primo dataset su cui sono stati svolti i test è il seguente:

- d1291.tsp
- d1655.tsp
- d2103.tsp
- dsj1000.tsp
- fl1400.tsp
- fl1577.tsp
- nrw1379.tsp
- pcb1173.tsp
- pr1002.tsp
- pr2392.tsp
- rl1304.tsp
- rl1323.tsp
- rl1889.tsp
- u1060.tsp
- u1432.tsp
- u1817.tsp
- u2152.tsp
- u2319.tsp
- vm1084.tsp
- vm1748.tsp

L'algoritmo genetico non è stato applicato a questo primo dataset. Il motivo di tale scelta, riguarda il tempo di calcolo troppo elevato per la generazione della popolazione.
Per tale motivo tutti gli algoritmi meta-euristici sono stati testati nuovamente sulle seguenti istanze di dimensione minore, per ottenere un confronto più equo della soluzione del genetico con le altre:

- a280.tsp
- bier127.tsp
- ch130.tsp
- ch150.tsp
- d198.tsp
- gil262.tsp
- gr137.tsp
- kroA150.tsp
- kroA200.tsp
- kroB150.tsp
- kroB200.tsp
- pr124.tsp
- pr136.tsp
- pr144.tsp
- pr152.tsp
- pr226.tsp
- pr264.tsp
- pr299.tsp
- rd400.tsp
- u159.tsp

Analizzando i risultati ottenuti in entrambi i test (Figura 4.9 e 4.10), le soluzioni con costo minore sono quelle ottenute mediante il Simulated Annealing e il VNS ibribo. Nel primo caso, con istanze di grandezza maggiore, la seconda implementazione del VNS genera soluzioni di costo minore mentre nel confronto con l'algoritmo genetico, la prima implementazione del VNS risulta migliore. I grafici in Figura 4.11, 4.12, 4.13 e 4.14 rappresentano invece l'andamento del costo applicando rispettivamente il VNS ibrido, il Tabu Search, il Simulated Annealing e l'algoritmo genetico all'istanza *bier127.tsp*.

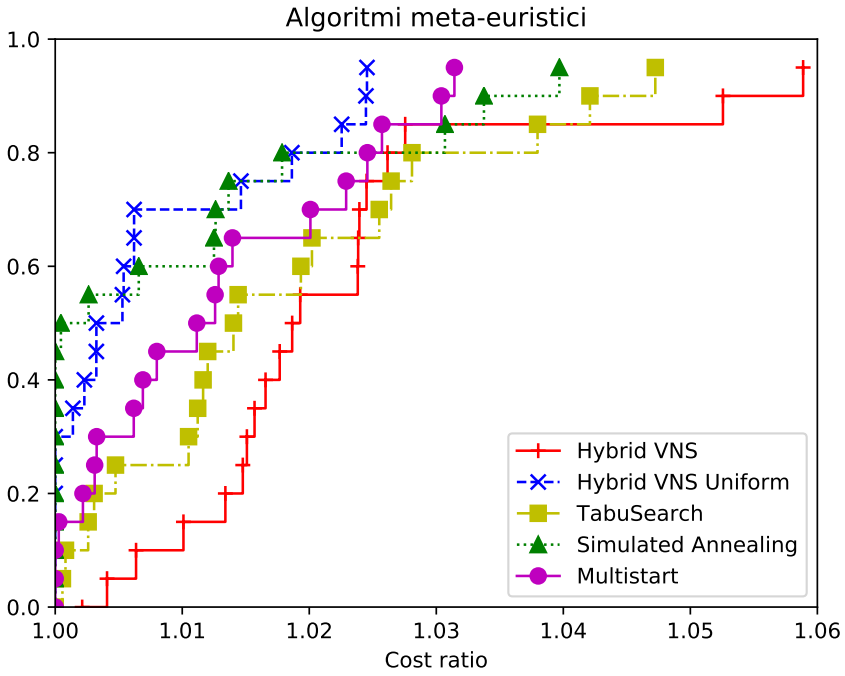

Figura 4.9: Confronto dei costi delle soluzioni ottenute mediante gli algoritmi meta-euristici, escluso il genetico.

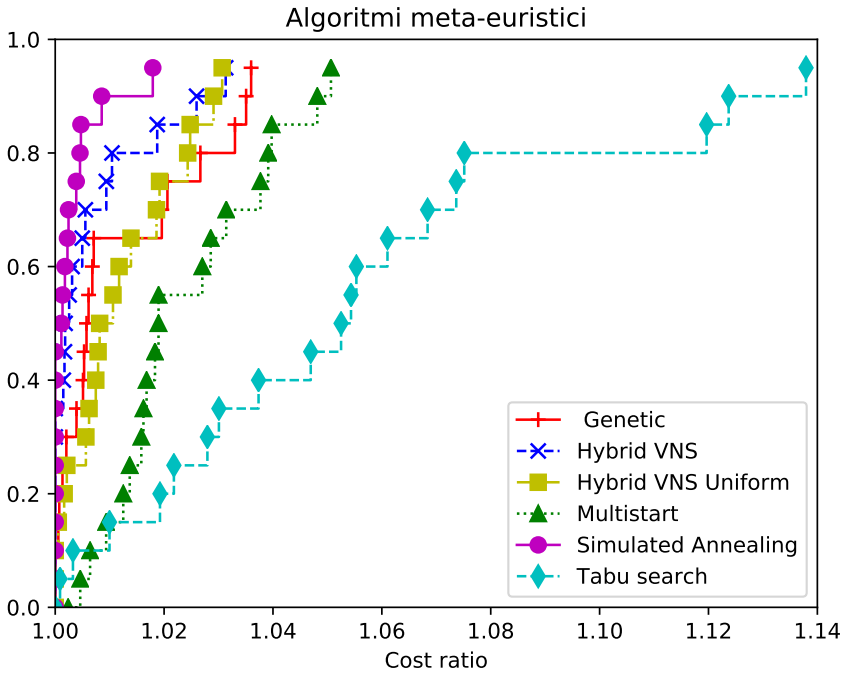

Figura 4.10: Confronto dei costi delle soluzioni ottenute mediante tutti gli algoritmi meta-euristici.

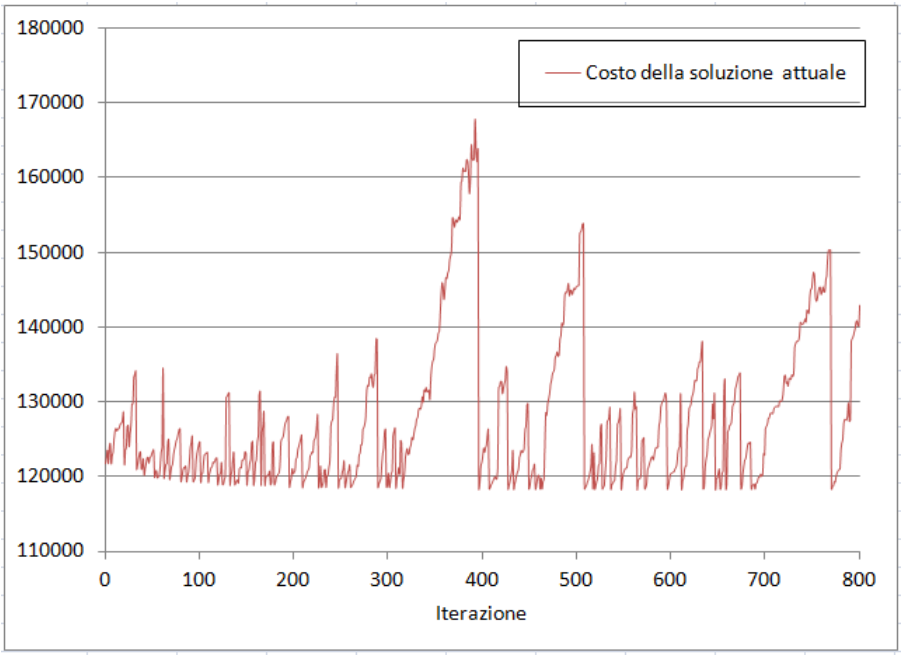

Figura 4.11: Evoluzione del costo della soluzione attuale nell'algoritmo VNS.

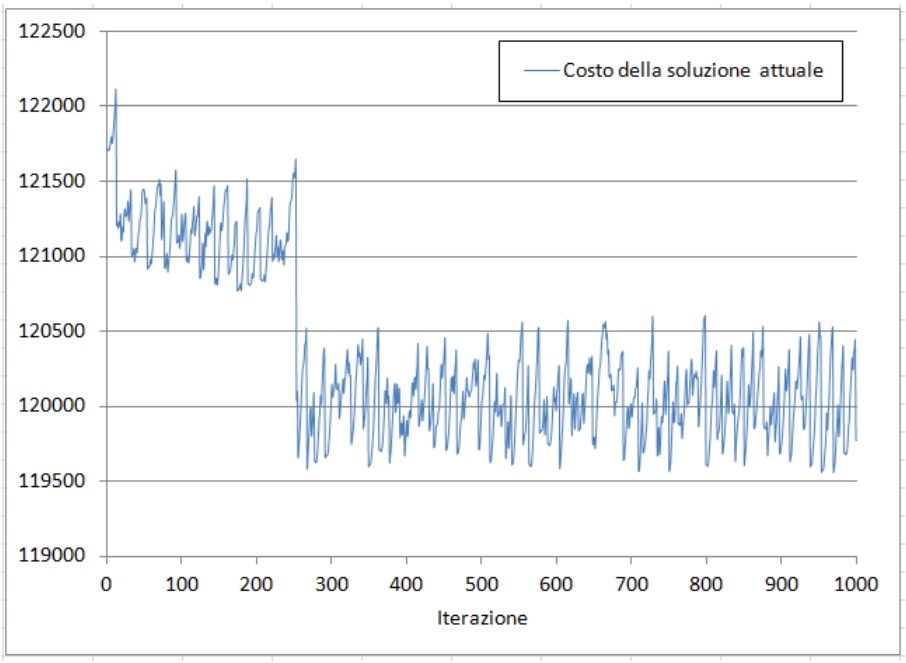

Figura 4.12: Evoluzione del costo della soluzione attuale nell'algoritmo Tabu search.

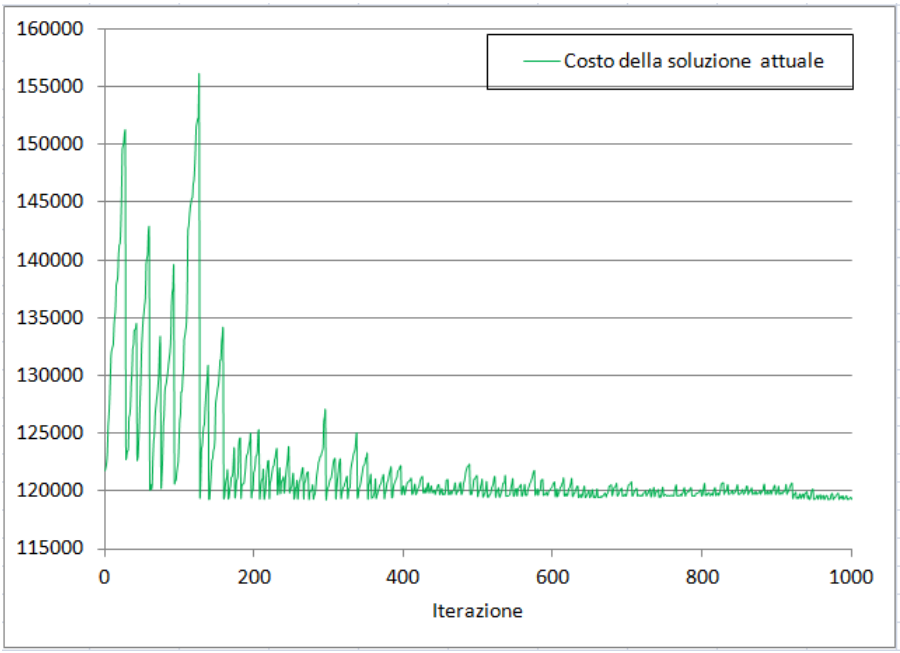

Figura 4.13: Evoluzione del costo della soluzione attuale nell'algoritmo Simulated Annealing.

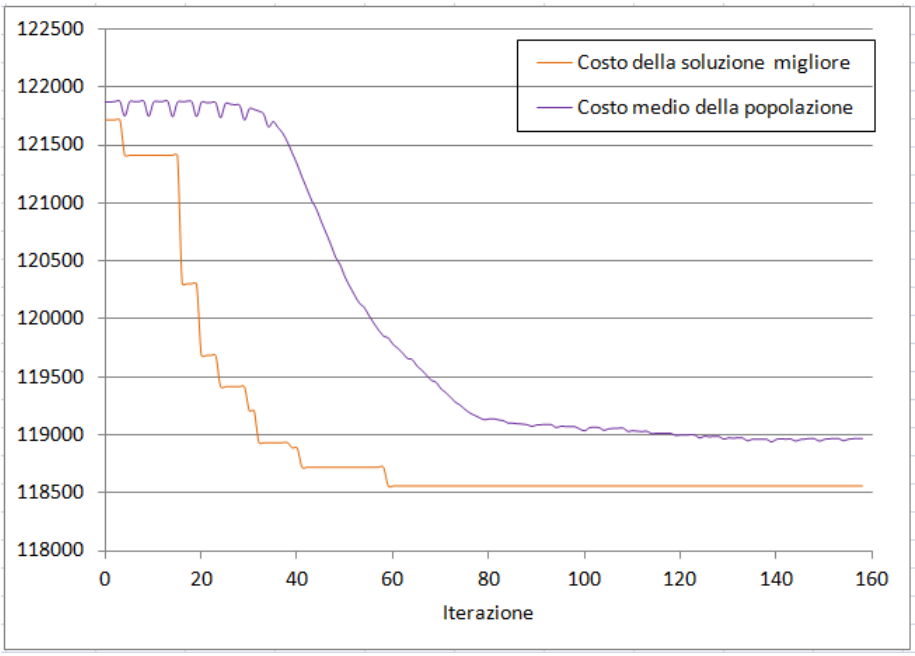

Figura 4.14: Evoluzione del costo medio della popolazione e del costo della soluzione migliore nell'algoritmo genetico.

Appendice A

TSPlib

Un'istanza del TSP viene definita normalmente da un grafo, per cui ad ogni nodo viene associata un numero intero (Ex. $\Pi = \{1, 2, 3, .., n\}$).
Una soluzione del problema è una sequenza di nodi che corrisponde ad una permutazione dell'istanza (es. $S = \{x_1, x_2, ..., x_n\}$ tale che $x_i \in \Pi \ \forall \ x_i \in S \ \wedge \ x_i \neq x_j \ \forall \ i \neq j$). Nella variante simmetrica, ogni tour può essere descritto da due versi di percorrenza e per questo motivo l'origine può essere un nodo qualsiasi del grafo.
La rappresentazione di tali istanze è stata svolta attraverso l'utilizzo del programma Gnuplot. Per avere dettagli riguardanti il suo utilizzo vedi Appendice C. Le istanze del problema, analizzate durante il corso, sono punti dello spazio 2D, identificati quindi da due coordinate (x,y).
Per generare istanza enormi del problema, si utilizza un approccio particolare in cui viene definito un insieme di punti a partire da un dipinto già esistente. La distanza tra i punti generati dipende dalle gradazioni di luce nella trasformazione dell'immagine in scala di grigi.
L'esempio più famoso di istanze generate in questo modo è quello ottenuto dal dipinto della Gioconda [6]. Le istanze che vengono risolte dai programmi, implementati durante il corso, sono file che utilizzano il template **TSPlib**. Di seguito viene riportato un esempio del contenuto di un file di questa tipologia:

```
NAME : esempio
COMMENT : Grafo costituito da 5 nodi
TYPE : TSP
DIMENSION : 5
EDGE_WEIGHT_TYPE : ATT
NODE_COORD_SECTION
1 6734 1453
2 2233 10
3 5530 1424
4 401 841
5 3082 1644
EOF
```

Listing A.1: esempio.tsp

Le parole chiave più importanti, contenute in questi file, sono:

- **NAME**
  seguito dal nome dell'istanza TSPlib
- **COMMENT**
  seguito da un commento associato all'istanza
- **TYPE**
  seguito dalla tipologia dell'istanza
- **DIMENSION**
  seguito dal numero di nodi nel grafo (num_nodi)

- **EDGE_WEIGHT_TYPE**
  seguito dalla specifica del tipo di calcolo che viene effettuato per ricavare il costo del tour
- **NODE_COORD_SECTION**
  inizio della sezione composta di *num_nodi* righe in cui vengono riportate le caratteriste di ciascun nodo, nella forma seguente:

```
indice_nodo  coordinata_x  coordinata_y
```

- **EOF**
  decreta la fine del file

In alternativa alla sezione **NODE_COORD_SECTION** è possibile avere una sezione chiamata **EDGE_WEIGHT_SECTION**, in cui sono riportati, all'interno di una matrice, i pesi di tutti gli archi e non le caratteristiche dei nodi. Gli algoritmi implementati e descritti nel report, sono applicabili unicamente a istanze con sezione di tipologia **NODE_COORD_SECTION**.

Appendice B

ILOG CPLEX

In questa sezione verranno approfondite alcune funzioni di CPLEX necessarie ad implementare gli algoritmi descritti nei capitoli precedenti.

B.1 Funzionamento

Per poter utilizzare gli algoritmi di risoluzione forniti da CPLEX è necessario costruire il modello matematico del problema, legato all'istanza precedentemente descritta.
CPLEX ha due meccanismi di acquisizione dell'istanza:

1. **modalità interattiva**
   in cui il modello viene letto da un file precedentemente generato (*model.lp*)
2. **creazione nel programma**
   il modello viene creato attraverso le API del linguaggio usato per la scrittura del programma

Le strutture utilizzate da CPLEX sono due (vedi Figura B.1):

- **ENV:** contiene i parametri necessari all'esecuzione e al salvataggio dei risultati
- **LP:** contiene il modello che viene analizzato da CPLEX durante la computazione del problema di ottimizzazione

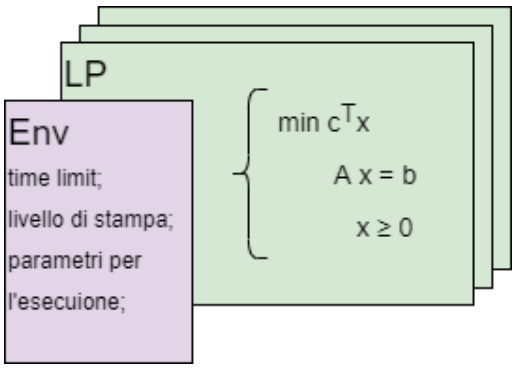

Figura B.1: Strutture CPLEX

Ad ogni ENV è possibile associare più LP, in modo da poter risolvere in parallelo più problemi di ottimizzazione, ma nel nostro caso ne sarà sufficiente solo uno.
Per convenzione è stato deciso di etichettare i rami (i, j) dell'istanza rispettando la proprietà $i < j$.
In Figura B.2 è riportato lo schema degli indici che vengono utilizzati per etichettare le variabili.
In questa figura le celle (i, j) bianche, sono quelle effettivamente utilizzate per indicare un arco

secondo la convenzione. Il numero all'interno di queste caselle rappresenta invece l'ordine in cui queste variabili vengono inserite nel modello e quindi gli indici associati da CPLEX per accedere alla soluzione.

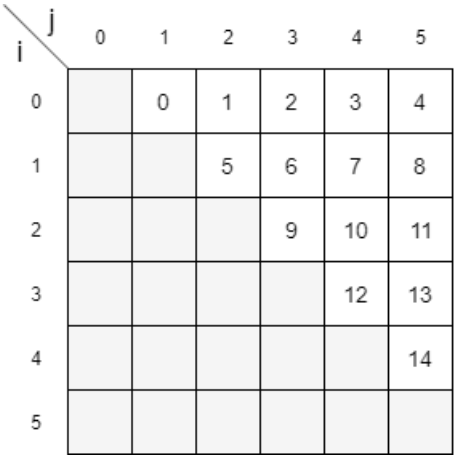

Figura B.2: Indici della matrice

B.2 Funzioni

B.2.1 Costruzione e modifica del modello

Per poter costruire il modello da analizzare, come prima cosa, è necessario creare un puntatore alle due strutture dati utilizzate da CPLEX.

```
1  int error;
2  CPXENVptr env = CPXopenCPLEX(&error);
3  CPXLPptr lp = CPXcreateprob(env, &error, "TSP");
```

Listing B.1: modelTSP.txt

La funzione alla riga 2 alloca la memoria necessaria e riempie la struttura con valori di default. Nel caso in cui non termini con successo memorizza un codice d'errore in *error*.
La funziona invocata nella riga successiva, invece, associa la struttura LP all'ENV che gli viene fornito. Il terzo parametro passato, nell'esempio "TSP", sarà il nome del modello. Al termine di queste operazioni verrà quindi creato un modello vuoto. All'interno del nostro programma per inizializzarlo è stata definita la seguente funzione:

```
void cplex_build_model(tsp_instance* tsp_in, CPXENVptr env, CPXLPptr lp);
```

tsp_in: puntatore alla struttura che contiene l'istanza del problema (letta dal file TSPlib)

env: puntatore alla struttura ENV precedentemente creata

lp: puntatore alla struttura LP precedentemente creata

All'interno di **cplex_build_model()** viene aggiunta nel modello una colonna alla volta. Le variabili aggiunte corrispondono agli archi del grafo e per ciascuno di questi viene calcolato il costo come distanza euclidea. La funzione necessaria ad inserire colonne e definire la funzione di costo è la seguente:

```
int CPXnewcols( CPXCENVptr env, CPXLPptr lp, int ccnt, double const * obj,
    double const * lb, double const * ub, char const * xctype, char ** colname)
```

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
ccnt:	numero di colonne da inserire
obj:	vettore dei costi relativi agli archi da inserire
lb:	vettore contenente i lower bound dei valori assumibili dalle variabili da inserire
ub:	vettore contenente gli upper bound dei valori assumibili dalle variabili da inserire
xctype:	vettore contenente la tipologia delle variabili da inserire
colname:	vettore di stringhe contenenti i nomi delle variabili da inserire
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Tale funzione permette però di aggiungere più colonne contemporaneamente. Per questo la generica colonna **i**, aggiunta dalla funzione, sarà definita dalle informazioni contenute all'interno della posizione **i** degli array, ricevuti come parametri. Nel programma elaborato durante il corso, viene aggiunta una colonna alla volta all'interno del modello. Per far ciò, è necessario comunque utilizzare riferimenti alle variabili da aggiungere, in modo da ovviare il problema riguardante la tipologia di argomenti richiesti, che sono array. Ad esempio, nel nostro caso, la tipologia di una nuova variabile inserita sarà un puntatore al carattere **'B'**, che la identifica come binaria.
Per poter inserire il primo insieme di vincoli del problema

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall\, v \in V$$

viene invece sfruttata la seguente funzione:

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

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
rcnt:	numero di righe (vincoli) da inserire
rhs:	vettore dei termini noti dei vincoli
sense:	vettore di caratteri che specifica il tipo di vincoli da inserire. Ogni carattere può assumere: *'L'* per vincolo $\leq$ *'E'* per vincolo $=$ *'G'* per vincolo $\geq$ *'R'* per vincolo definito in un intervallo
rngval:	vettore di range per i valori di ogni vincolo (nel nostro caso è NULL)
rowname	vettore di stringhe contenenti i nomi delle variabili da inserire
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

In modo analogo all'inserimento delle colonne, nel nostro programma viene aggiunta una riga alla volta nel modello. L'**i**-esima riga aggiunta corrisponderà al vincolo imposto sul nodo **i**-esimo,

imponendo a 1 il coefficiente della variabile x_{kj} se $k = i \wedge j = i$ per ogni variabile del modello. In questo modo però viene aggiunto un vincolo in cui è necessario cambiare i coefficienti delle variabili che ne prendono parte. Per fare ciò è necessaria la funzione:

```
int CPXchgcoef(CPXCENVptr env, CPXLPptr lp, int i, int j, double newvalue);
```

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
i:	intero che specifica l'indice della riga in cui modificare il coefficiente
j:	intero che specifica la colonna in cui si trova la variabile per cui modificare il coefficiente
newvalue:	nuovo valore del coefficiente
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

L'utilizzo di **CPXnewrows()** insieme a **CPXchgcoef()** per inserire nuovi vincoli è però considerato inefficiente. Al suo posto è consigliato l'utilizzo della seguente funzione che inserisce il vincolo con già i coefficienti delle variabili impostati al valore corretto:

```
int CPXaddrows( CPXCENVptr env, CPXLPptr lp, int ccnt, int rcnt, int nzcnt,
    double const * rhs, char const * sense, int const * rmatbeg,
    int const * rmatind, double const * rmatval, char ** colname,
    char ** rowname );
```

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
ccnt:	numero di nuove colonne che devono essere aggiunte
rcnt:	numero di nuove righe che devono essere aggiunte
nzcnt:	numero di coefficienti non nulli nel vincolo aggiunto
rhs:	vettore con i termini noti per ogni vincolo da aggiungere
sense:	vettore con il tipo di vincoli da aggiungere, scelto tra: *'L'* per vincolo $\leq$ *'E'* per vincolo $=$ *'G'* per vincolo $\geq$ *'R'* per vincolo definito in un intervallo
rmatbeg:	vettore con le posizioni iniziali dei coefficienti nei vincoli
rmatind:	vettore contenente gli indici delle variabili appartenenti al vincolo
rmatval:	vettore con i coefficienti delle variabili del vincolo
colname:	vettore contenente i nomi delle nuove colonne
rowname:	vettore contenente i nomi dei nuovi vincoli
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Per rimuovere invece delle righe, viene utilizzata la seguente funzione:

```
int CPXdelrows( CPXCENVptr env, CPXLPptr lp, int begin, int end );
```

env: puntatore alla struttura ENV precedentemente creata
lp: puntatore alla struttura LP precedentemente creata
begin: indice numerico della prima riga da cancellare
end: indice numerico dell'ultima riga da cancellare
Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

Per poter impostare una variabile x_{ij} ad una valore fissato è necessario assegnare al suo lower e al suo upper bound determinati valori tramite la seguente funzione:

```
int CPXchgbds(CPXCENVptr env, CPXLPptr lp, int cnt, const int * indices,
    const char * lu, const double * bd);
```

env: puntatore alla struttura ENV
lp: puntatore alla struttura LP
cnt: numero totale di bound da cambiare
indices: vettore con gli indici delle colonne corrispondenti alle variabili per cui bisogna cambiare il bound
lu: array di caratteri che specificano il bound da modificare.
I possibili valori di tali caratteri sono:
'U' per selezionare l'upper bound
'L' per selezionare il lower bound
'B' per selezionare entrambi i bound
bd: vettore con i nuovi valori
Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

B.2.2 Calcolo della soluzione

Per ottenere la soluzione ottima del problema di ottimizzazione, correlato al modello definito in CPLEX, vengono utilizzate due fasi:

- **Risoluzione del problema di ottimizzazione**

```
int CPXmipopt(CPXCENVptr env, CPXLPptr lp);
```

env: puntatore alla struttura ENV precedentemente creata
lp: puntatore alla struttura LP precedentemente creata
Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

- **Ottenimento della soluzione**

```
int CPXgetmipx (CPXENVptr env, CPXLPptr lp, double *x, int begin, int end);
```

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
x:	puntatore a un vettore di double in cui verranno salvati i valori delle variabili ottenuti dalla soluzione ottima
begin:	primo indice della variabile di cui si vuole memorizzare ed analizzare il valore
end:	indice dell'ultima variabile di cui si vuole memorizzare ed analizzare il valore
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Questa funzione salva in x tutte le variabili che hanno indice $i \in [begin, end]$ e quindi x deve essere un vettore di almeno $end - begin + 1$ valori. Nel nostro programma, vengono analizzati i valori di tutte le variabili definite [1][2].
Per questo motivo **begin = 0** e **end = num_colonne - 1**.
In seguito il nostro programma analizza la correttezza della soluzione svolgendo la verifica su:

- *valori assunti dalle variabili*
  ciascun x_{ij} assume valore 0 o 1 con una tolleranza di $\epsilon = 10^{-5}$
- *grado di ciascun nodo*
  il tour è composto al massimo da due archi che tocchino lo stesso nodo

- **Gap relativo**
  La seguente funzione permette di ottenere il gap relativo della funzione obiettivo per un'ottimizzazione MIP.

```
int CPXgetmiprelgap( CPXCENVptr env, CPXCLPptr lp, double * gap_p );
```

env:	puntatore alla struttura ENV precedentemente creata
lp:	puntatore alla struttura LP precedentemente creata
gap_p:	puntatore alla variabile in cui verrà salvato il gap
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Per un problema di minimizzazione il gap relativo viene calcolato come:

$$\frac{|bestbound - bestinteger|}{10^{-10} + |bestinteger|}$$

dove **bestinteger** è il valore restituito dalla funzione **CPXgetobjval()** mentre **bestbound** da **CPXgetbestobjval()**.

B.2.3 Lazy constraint

Nel caso in cui si voglia verificare il soddisfacimento di un vincolo solo al termine della computazione della soluzione, è necessario inserire un "*lazy constraint*". Questi vincoli vengono dichiarati in fase di costruzione del modello e aggiunti ad un pool. Per fare ciò viene utilizzata la seguente funzione:

[1] numero di variabili=CPXgetnumcols(env,lp);
[2] numero di vincoli=CPXgetnumrows(env,lp);

```
int CPXaddlazyconstraints( CPXCENVptr env, CPXLPptr lp, int rcnt, int nzcnt,
    double const * rhs, char const * sense, int const * rmatbeg,
    int const * rmatind, double const * rmatval, char ** rowname );
```

env: puntatore alla struttura ENV precedentemente creata

lp: puntatore alla struttura LP precedentemente creata

rcnt: numero di vincoli da inserire

nzcnt: numero di coefficienti non nulli nel vincolo

rhs: vettore dei termini noti dei vincoli

sense: vettore di caratteri che specifica il tipo di vincoli da inserire.
Ogni carattere può assumere uno dei seguenti valori:
'L' per un vincolo $\leq$
'E' per un vincolo $=$
'G' per un vincolo $\geq$
'R' per un vincolo definito in un intervallo

rmatbeg: vettore con le posizioni iniziali dei coefficienti nei vincoli

rmatind: vettore contenente gli indici delle variabili appartenenti al vincolo

rmatval: vettore con i coefficienti delle variabili del vincolo

rowname: vettore con i nomi dei vincoli

Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

In modo analogo alle due funzioni precedentemente descritte per l'aggiunta di righe e colonne, nel nostro modello viene inserito un vincolo per volta. Per impostare correttamente i coefficienti delle variabili presenti nel vincolo, vengono sfruttati i due array *rmatinds* e *rmatval*. Come rappresentato in Figura B.3, all'interno della posizione i-esima del vettore di indici è presente la posizione dell'i-esima variabile del vincolo da inserire (nell'esempio in figura $rmatinds[i] = j$). Mentre l'i-esima posizione del vettore di valori contiene il valore del corrispondente coefficiente (c_j).

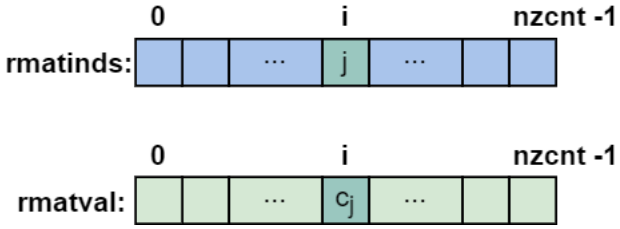

Figura B.3: Array dei lazy constraint.

B.2.4 Lazy Constraint Callback

Per poter utilizzare una lazy constraint callback, precedentemente implementata all'interno del programma, prima di tutto è necessario installarla. Questo viene fatto attraverso la seguente funzione:

```
int CPXsetlazyconstraintcallbackfunc( CPXENVptr env,
    int(CPXPUBLIC *lazyconcallback)(CALLBACK_CUT_ARGS), void * cbhandle);
```

env:	puntatore alla struttura ENV
lazyconcallback:	puntatore alla callback chiamata
cbhandle:	puntatore ad una struttura dati contenente le informazioni da passare alla callback
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

CPLEX, non sapendo se la funzione implementata dall'utente sia thread safe, impedisce di default lo svolgimento di elaborazioni in parallelo utilizzando le callback. Se l'utente volesse svolgere le azioni in multithreading, deve impostare il parametro *CPX_PARAM_THREADS* appena l'installazione della callback, mediante la funzione descritta nella Sezione B.3. Per poter conoscere il numero di thread della macchina su cui si sta lavorando, CPLEX fornisce la seguente funzione:

```
int CPXgetnumcores(CPXCENVptr env, int * numcores_p);
```

env:	puntatore ad una struttura ENV
numcores_p:	puntatore alla variabile in cui scrivere il numero di core
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Le callback sono funzioni lasciate appositamente vuote da CPLEX, affinché l'utente possa implementarle in maniera personalizzata. Queste però hanno una dichiarazione standard, qui riportata:

```
static int CPXPUBLIC name_function(CPXCENVptr env, void* cbdata, int wherefrom,
     void* cbhandle, int* useraction_p);
```

env:	puntatore una struttura ENV
cbdata:	puntatore che contiene specifiche informazioni necessarie alla callback
wherefrom:	contiene dove è stata invocata la callback durante l'ottimizzazione
cbhandle:	puntatore a dati privati dell'utente
useraction_p:	puntatore ad un intero che specifica a CPLEX come proseguire la computazione al termine della callback dell'utente
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

L'utente deve impostare il valore di *useraction_p* con una delle seguenti macro:

- **CPX_CALLBACK_DEFAULT**:
  l'utente dichiara a CPLEX di non aver trovato nessuna soluzione;
- **CPX_CALLBACK_FAIL**:
  CPLEX esce dall'ottimizzazione;
- **CPX_CALLBACK_SET**:
  l'utente dichiara a CPLEX di aver trovato una soluzione e gli chiede di usare la soluzione fornita;

Nell'implementare la callback bisogna prestare attenzione nello svolgere azioni thread safe, nel caso in cui si voglia sfruttare l'architettura parallela del processore. Per avere accesso alla soluzione attuale dal nodo che invoca la callback è possibile chiamare la seguente funzione:

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

env:	puntatore una struttura ENV
cbdata:	puntatore a specifiche informazioni necessarie alla callback
wherefrom:	punto dell'ottimizzazione in cui è stata invocata la callback
x:	vettore in cui memorizzare le variabili
begin:	indice della prima variabile da memorizzare
end:	indice dell'ultima variabile da memorizzare
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Per ottenere informazioni riguardanti il problema di ottimizzazione che si sta risolvendo all'interno di una callback implementata dall'utente, è possibile utilizzare la seguente funzione:

```
int CPXgetcallbackinfo(CPXCENVptr env, void * cbdata, int wherefrom, int whichinfo,
    void * result_p);
```

env:	puntatore ad una struttura ENV
cbdata:	puntatore a specifiche informazioni necessarie alla callback
wherefrom:	punto dell'ottimizzazione in cui è stata invocata la callback
whichinfo:	macro che specifica l'informazione che si desidera conoscere
result_p:	puntatore di tipo void in cui verrà memorizzata l'informazione richiesta
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Alcune macro utili che possono essere inserite come *whichinfo* sono:

- **CPX_CALLBACK_INFO_MY_THREAD_NUM**:
  identifica il thread che ha eseguito la chiamata;
- **CPX_CALLBACK_INFO_BEST_INTEGER**:
  valore della miglior soluzione intera;

Per conoscere il costo della soluzione legata al nodo corrente che invoca la callback, può essere utilizzata la seguente funzione:

```
int CPXgetcallbacknodeobjval(CPXCENVptr env, void * cbdata, int wherefrom,
    double * objval_p);
```

env:	puntatore ad una struttura ENV
cbdata:	puntatore a specifiche informazioni necessarie alla callback
wherefrom:	punto dell'ottimizzazione in cui è stata invocata la callback
objval_p:	puntatore ad una variabile in cui memorizzare il costo
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

All'interno della lazy callback è necessario aggiungere il taglio voluto al nodo corrente che la invoca. Questo può essere fatto in due diverse modalità: globale o locale.

Nel primo caso il vincolo aggiunto sarà visibile da tutti i nodi. Inoltre, in caso non lo ritenga più necessario, CPLEX potrà eliminarlo dal modello. Quest'operazione viene detta *purge* e si verifica, ad esempio, quando un taglio non viene applicato per molte iterazioni consecutive. Per un vincolo globale viene chiamata la seguente funzione, che ne aggiunge uno alla volta:

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

env: puntatore ad una struttura ENV

cbdata: puntatore a specifiche informazioni necessarie alla callback

wherefrom: punto dell'ottimizzazione in cui è stata invocata la callback

nzcnt: numero di coefficienti non nulli

rhs: valore del termine noto

sense: tipologia del taglio da aggiungere, a scelta tra:

'L' per il vincolo $\leq$

'E' per il vincolo $=$

'G' per il vincolo $\geq$

cutind: vettore contente gli indici dei coefficienti del taglio

cutval: vettore contenente i coefficienti delle variabili nel taglio

purgeable: intero che specifica in che modo CPLEX deve trattare il taglio, consigliato 0

Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

Nella modalità locale invece, il taglio aggiunto sarà visibile solo ai nodi discendenti di quello che invoca la callback. In questo caso il taglio viene aggiunto mediante la seguente chiamata:

```
int CPXcutcallbackaddlocal( CPXCENVptrenv, void *cbdata, int wherefrom,
    int nzcnt, double rhs, int sense, int const *cutind, double const *cutval )
```

env: puntatore ad una struttura ENV

cbdata: puntatore a specifiche informazioni necessarie alla callback

wherefrom: punto dell'ottimizzazione in cui è stata invocata la callback

nzcnt: numero di coefficienti non nulli

rhs: valore del termine noto

sense: tipologia del taglio da aggiungere, a scelta tra

'L' per il vincolo $\leq$

'E' per il vincolo $=$

'G' per il vincolo $\geq$

cutind: vettore contente gli indici dei coefficienti del taglio

cutval: vettore contenente i coefficienti delle variabili nel taglio

Return Value: 0 in caso di successo, un valore diverso da 0 altrimenti

B.2.5 Heuristic Callback

Per poter suggerire a CPLEX una soluzione del problema calcolata dall'utente, è necessario utilizzare un particolare tipo di callback, detta *heuristic callback*. Questa, dopo essere stata installata, verrà invocata ogni volta che viene trovata una nuova soluzione per il sottoproblema corrente.
Per installare la callback viene utilizzata la seguente funzione:

```
int CPXsetheuristiccallbackfunc(CPXENVptr env,
     int(CPXPUBLIC *heuristiccallback)(CALLBACK_HEURISTIC_ARGS),
     void * cbhandle);
```

env:	puntatore ad una struttura ENV
heuristiccallback:	puntatore all'heuristic callback scritta dall'utente
cbhandle:	puntatore a dati privati dell'utente
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

La callback creata dall'utente deve rispettare la seguente dichiarazione:

```
int callback (CPXCENVptr env, void *cbdata, int wherefrom, void *cbhandle,
   double *objval_p, double *x, int *checkfeas_p, int *useraction_p);
```

env:	puntatore ad una struttura ENV
cbdata:	puntatore a specifiche informazioni necessarie alla callback
wherefrom:	punto dell'ottimizzazione in cui è stata invocata la callback
cbhandle:	puntatore a dati privati dell'utente
objval_p:	puntatore ad una variabile che inizialmente contiene il costo della soluzione attuale e in seguito verrà riempito con il costo della soluzione trovata dall'utente
x:	vettore contenente inizialmente la soluzione attuale del problema ed in seguito sostituito dalla soluzione trovata dall'utente
checkfeas_p:	puntatore che specifica se CPLEX debba verificare la fattibilità della soluzione trovata oppure no
useraction_p:	puntatore ad un intero che specifica a CPLEX come proseguire la computazione al termine della callback dell'utente.
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

L'utente deve impostare il valore di *useraction_p* con una delle seguenti macro:

- **CPX_CALLBACK_DEFAULT**:
  l'utente dichiara a CPLEX di non aver trovato nessuna soluzione;
- **CPX_CALLBACK_FAIL**:
  CPLEX esce dall'ottimizzazione;
- **CPX_CALLBACK_SET**:
  l'utente dichiara a CPLEX di aver trovato una soluzione e chiede di usare la soluzione fornita;

B.2.6 Generic Callback

Per evitare che alcune procedure interne a CPLEX vengano disattivate nel momento dell'installazione di una callback, recentemente è stata sviluppata una particolare tipologia di callback, detta *generic*. Questa non è relativa a una specifica versione di callback, come quelle descritte nelle sezioni precedenti, ma può essere invocata in contesti diversi e con diversi scopi.
Per installare una generic callback viene utilizzata la seguente funzione, in cui è necessario specificare il contesto in cui invocare la callback:

```
int CPXcallbacksetfunc ( CPXENVptr env, CPXLPptr lp, CPXLONG contextmask,
    CPXCALLBACKFUNC *callback, void * userhandle );
```

env:	puntatore ad una struttura ENV
lp:	puntatore alla struttura LP
contextmask:	contesto sul quale deve essere invocata la callback (è possibile metterne più di uno utilizzando l'or bitwise e gestire poi i singoli casi dall'interno della funzione)
callback:	puntatore alla callback scritta dall'utente
userhandle:	puntatore ad una struttura che contiene i dati da passare alla callback
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Alcuni possibili valori di *contextmask* sono:

- **CPX_CALLBACKCONTEXT_CANDIDATE**:
  la callback verrà invocata quando viene trovata da CPLEX una nuova soluzione ammissibile che l'utente poi potrà rifiutare;
- **CPX_CALLBACKCONTEXT_LOCAL_PROGRESS**:
  la callback verrà invocata nel momento in cui un thread effettua un progresso, non ancora noto globalmente, nella soluzione del problema. In questo contesto l'utente può suggerire a CPLEX una soluzione da cui proseguire la computazione (analogamente alle heuristic callback).

La callback implementata dall'utente deve avere questa dichiarazione:

```
static int CPXPUBLIC name_general_callback(CPXCALLBACKCONTEXTptr
                    context, CPXLONG contextid, void* userhandle);
```

context:	puntatore alla struttura del contesto della callback
contextid:	intero che specifica il contesto in cui viene invocata la callback
userhandle:	argomento passato alla callback nell'installazione
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

L'utente può installare una sola callback, ma al suo interno può distinguere il contesto in cui è stata invocata grazie al parametro *contextid*.
Per poter accedere alla soluzione candidata e al suo costo, deve essere presente la seguente chiamata, che è specifica per il contesto **CPX_CALLBACKCONTEXT_CANDIDATE**:

```
int CPXcallbackgetcandidatepoint( CPXCALLBACKCONTEXptr context, double *x,
    CPXDIM begin, CPXDIM end, double *obj_p);
```

context:	contesto (lo stesso passato alla callback scritta dall'utente)
x:	vettore dove memorizzare le variabili richieste
begin:	indice prima variabile richiesta
end:	indice dell'ultima variabile richiesta
obj_p:	puntatore alla variabile in cui memorizzare il costo della soluzione candidata, può essere NULL
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Per poter scartare una soluzione, nel caso in cui violi alcuni tagli specificati nella chiamata stessa, viene utilizzata la seguente funzione. Anche questa è specifica per il contesto **CPX_CALLBACKCONTEXT_CANDIDATE**.

```
int CPXcallbackrejectcandidate( CPXCALLBACKCONTEXTptr context, int rcnt, int nzcnt,
    double const *rhs, char const *sense, int const *rmatbeg, int const *rmatind,
    double const *rmatval );
```

context:	contesto (lo stesso passato alla callback scritta dall'utente)
rcnt:	numero di vincoli che tagliano la soluzione
nzcnt:	numero di coefficienti non nulli nel vincolo
rhs:	vettore dei termini noti
sense:	vettore con la tipologia dei vincoli
rmatbeg:	vettore di indici che specifica dove inizi ogni vincolo
rmatind:	vettore di indici delle colonne con coefficienti non nulli
rmatval:	coefficienti non nulli delle colonne specificate
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Per suggerire a CPLEX una soluzione da sfruttare nel proseguimento della computazione dev'essere utilizzata la seguente funzione:

```
int CPXcallbackpostheursoln( CPXCALLBACKCONTEXTptr context, CPXDIM cnt,
      CPXDIM const * ind, double const * val, double obj,
      CPXCALLBACKSOLUTIONSTRATEGY strat );
```

context:	contesto (lo stesso passato alla callback scritta dall'utente)
cnt:	numero di elementi nei vettori ind e val
ind:	vettore di indici non nulli dei valori della soluzione
val:	vettore di valori non nulli della soluzione (possono essere NaN nel caso in cui la soluzione sia parziale)
obj:	costo della nuova soluzione
strat:	strategia con cui CPLEX deve completare la nuova soluzione, nel caso sia parziale
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

La soluzione proposta dall'utente può anche essere incompleta, CPLEX provvederà poi a completarla seguendo le istruzioni fornite dal parametro *strat*. Questo può assumere i seguenti valori:

- **CPXCALLBACKSOLUTION_NOCHECK**:
  in questo caso CPLEX non controllerà la fattibilità della soluzione proposta, che deve però essere completa;
- **CPXCALLBACKSOLUTION_CHECKFEAS**:
  in questo caso CPLEX si limiterà a controllare la fattibilità della soluzione proposta, senza completarla, la soluzione non può perciò essere parziale;
- **CPXCALLBACKSOLUTION_PROPAGATE**:
  se la soluzione fornita è parziale, in questo caso CPLEX cercherà di completare la soluzione attraverso la propagazione del bound;
- **CPXCALLBACKSOLUTION_SOLVE**:
  se la soluzione fornita è parziale, CPLEX fisserà le variabili specificate nella soluzione e cercherà di risolvere il risultante problema ridotto;

CPLEX utilizzerà la soluzione proposta dall'utente solo nel caso in cui questa abbia costo inferiore all'incumbent.

B.3 Parametri

Con le seguenti funzioni è possibile modificare i parametri di configurazione di CPLEX, altrimenti impostati ai valori di default. Nel caso in cui si tratti di parametri di tipo `int` è necessario invocare:

```
int CPXsetintparam(CPXENVptr env, int whichparam, int newvalue);
```

mentre se di tipo `double`:

```
int CPXsetdblparam(CPXENVptr env, int whichparam, double newvalue);
```

In entrambe le funzioni:

env:	puntatore alla struttura ENV di cui si vogliono cambiare i parametri
whichparam:	id del parametro da modificare (vedi Tabella B.1)
newvalue:	nuovo valore (rispettivamente intero o double) del parametro
Return Value:	0 in caso di successo, un valore diverso da 0 altrimenti

Parametro	Descrizione
CPX_PARAM_EPGAP	tolleranza dell'intervallo tra la migliore funzione obiettivo intera e la funzione obiettivo del miglior nodo rimanente.
CPX_PARAM_INTSOLLIM	numero di soluzioni MIP intere che CPLEX deve trovare prima di terminare la computazione.
CPX_PARAM_MIPCBREDLP	permette, dalla callback che la chiama, di accedere al modello originale del problema e non a quello ridotto.
CPX_PARAM_NODELIM	massimo numero di nodi da risolvere prima che l'algoritmo termini senza aver aggiunto l'ottimalità (0 impone di fermarsi alla radice).
CPX_PARAM_POLISHTIME	tempo in secondi dedicato da CPLEX nel fare il polish della soluzione.
CPX_PARAM_POPULATELIM	numero di soluzioni MIP generate per il pool di soluzioni durante

	ogni chiamata alla procedura populate.
CPX_PARAM_RANDOMSEED	random seed utilizzato da CPLEX.
CPX_PARAM_RINSHEUR	frequenza (ogni quanti nodi) con cui deve essere invocato da CPLEX l'algoritmo euristico Rins.
CPX_PARAM_SCRIND	abilita la stampa dei messaggi di log di CPLEX.
CPX_PARAM_THREADS	numero massimo di thread utilizzabili.
CPX_PARAM_TIMELIMIT	tempo massimo che CPLEX ha a disposizione nel calcolo della soluzione.

Tabella B.1: Parametri.

B.4 Costanti utili

Di seguito vengono riportate alcune macro utili di CPLEX, insieme ai loro corrispondenti valori:

Macro	Valore	Descrizione
CPX_ON	**1**	valore da assegnare ad alcuni parametri per abilitarli
CPX_OFF	0	valore da assegnare ad alcuni parametri per disabilitarli
CPX_INFBOUND	$+\infty$	massimo valore intero utilizzabile in CPLEX (10^{20})

Tabella B.2: Macro.

Appendice C

Gnuplot

Nella nostra implementazione, una volta ottenuta la soluzione del problema di ottimizzazione, ne viene disegnato il grafo per facilitare all'utente la comprensione della sua correttezza. Per fare ciò viene utilizzato GNUplot, un programma di tipo command-driven.
Per poterlo sfruttare all'interno del proprio programma esistono due metodi:

- Collegare la libreria ed invocare le sue funzioni all'interno del programma
- Collegare l'eseguibile interattivo al proprio programma. In questo caso i comandi deve essere passati all'eseguibile attraverso l'utilizzo di un file di testo e di un pipe.

In questa trattazione è stato scelto il secondo metodo. All'interno di un file vengono specificati i comandi da eseguire in Gnuplot e le caratteristiche grafiche che deve aver il grafo da rappresentare. Un esempio di tale file, viene riportato nelle seguenti righe:

```
set style line 1 \
    linecolor rgb '#0000ff' \
    linetype 1 linewidth 1 \
    pointtype 7 pointsize 0.5

plot 'solution.dat' with linespoints linestyle 1 title "Solution"
     ,'' using 1:2:(sprintf("%d", $3)) notitle with labels center
     offset 1

set term png
set output "solution.png"
replot
```

Listing C.1: style.txt

Nell'esempio citato, nella prima parte viene definito lo stile, il colore delle linee e la tipologia di punti, che verrano in seguito visualizzati all'interno del grafico prodotto.
In seguito viene effettuato il plot, analizzando il file **solution.dat**, contenente le informazioni relative alla soluzione del problema, in cui ciascuna riga ha la seguente forma:

```
coordinata_x    coordinata_y    posizione_nel_tour
```

coordinata_x rappresenta la coordinata x del nodo;
coordinata_y rappresenta la coordinata y del nodo;
posizione_nel_tour rappresenta l'ordine del nodo all'interno del tour, assunto come nodo di origine il nodo 1.

Il grafico viene generato dal comando **plot**, leggendo tutte le righe non vuote e disegnando un punto nella posizione **(coordinata_x, coordinata_y)** del grafico 2D. In seguito viene tracciata una linea solo tra coppie di punti legati a righe consecutive non vuote nel file **solution.dat**.

Attraverso le istruzioni riportate nelle righe 10-12 di **style.txt**, viene invece salvato il grafico appena generato nell'immagine **solution.png**.

Di seguito vengono riportate le varie fasi necessarie alla definizione di un pipe e all'utilizzo di questo per eseguire comandi in GNUplot:

- **Definizione del pipe**

```
1 FILE* pipe = _popen(GNUPLOT_EXE, "w");
```

  dove **GNUPLOT_EXE** è una stringa composta dal percorso completo dell'eseguibile di GNUplot, seguita dall'argomento **-persistent** (es. *"D:/Programs/GNUplot/bin/gnuplot.exe -persistent"*).

- **Passaggio delle istruzioni a GNUplot**

```
2  f = fopen("style.txt", "r");
3
4  char line[180];
5  while (fgets(line, 180, f) != NULL)
6  {
7    fprintf(pipe, "%s ", line);
8  }
9
10 fclose(f);
```

  viene passata una riga alla volta, del file **style.txt**, a GNUplot mediante il pipe precedentemente creato.

- **Chiusura del pipe**

```
11 _pclose(pipe);
```

Appendice D

Performance profile in python

Il programma utilizzato per la creazione del performance profile dei diversi algoritmo è perprof.py, sviluppato da D. Salvagnin nel 2016. Di seguito vengono riportati i principali argomenti da linea di comando che possono essere utilizzati: Di seguito viene riportato un esempio dell'esecuzione del

Argomento	Descrizione
-D delimiter	specifica che delimiter verrà usato come separatore tra le parole in una riga
-M value	imposta value come il massimo valore di ratio (asse x)
-S value	value rappresenta la quantità che viene sommata a ciascun tempo di esecuzione prima del confronto. Questo parametro è utile per non enfatizzare troppo le differenze di pochi ms tra gli algoritmi.
-L	stampa in scala logaritmica
-T value	nel file passato al programma, il TIME LIMIT è value
-P "title"	title è il titolo del plot
-X value	nome dell'asse x (default ='Time Ratio')
-B	plot in bianco e nero

programma, del suo input e del suo output:

- **comando**

```
python perfprof.py -D ',' -T 3600 -S 2 -M 20 esempio.csv out.pdf
                   -P "Algoritmi esatti"
```

- **file di input con i dati**
  Viene riportato parte del contenuto di esempio.csv:

```
3, Alg1, Alg2, Alg3
model_1.lp, 2.696693, 3.272468, 2.434147
model_2.lp, 0.407689, 1.631921, 1.198957
model_3.lp, 0.333669, 0.432553, 0.966638
```

  La prima riga deve necessariamente contenere in ordine il numero di algoritmi analizzati e i loro nomi. Nelle righe seguenti viene riportato invece il nome del file lp e i tempi di esecuzione ottenuti, elencati secondo la sequenza di algoritmi specificata all'inizio. Ogni campo di ciascuna riga deve essere separato dal delimitatore specificato all'avvio del programma attraverso l'opzione -D.

- **immagine di output**
  Il grafico viene restituito nel file out.pdf specificato da linea di comando.

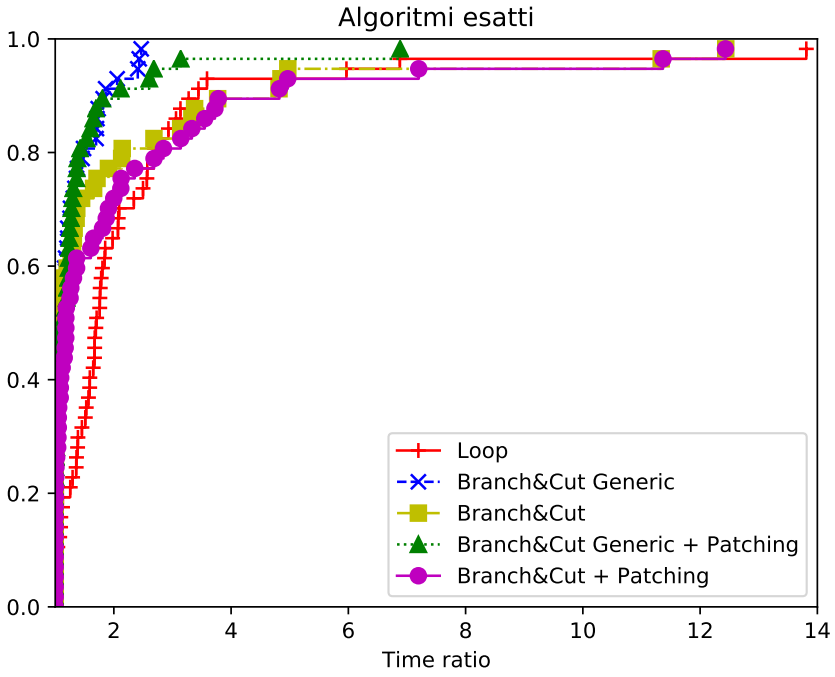

Figura D.1: Esempio di performance profile.

Appendice E

Risultati

Tabella E.1: Tempo di esecuzione degli algoritmi compatti con time limit di 20 minuti.

	MTZ con lazy constraint (s)	**MTZ (s)**	**GG (s)**
att48.tsp	186.679	405.781	12.532
berlin52.tsp	1.406	5.486	7.551
burma14.tsp	0.126	0.271	0.201
eil101.tsp	6.136	268.616	228.185
eil51.tsp	2.265	5.302	31.927
eil76.tsp	2.615	30.064	96.272
gr96.tsp	731.232	575.945	371.457
kroA100.tsp	TIME LIMIT	TIME LIMIT	468.402
kroB100.tsp	TIME LIMIT	TIME LIMIT	638.129
kroB150.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT
kroC100.tsp	TIME LIMIT	TIME LIMIT	299.991
kroD100.tsp	TIME LIMIT	TIME LIMIT	239.841
kroE100.tsp	1203.42	TIME LIMIT	306.457
pr124.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT
pr136.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT
pr76.tsp	425.937	1112.979	482.098
rat99.tsp	10.136	281.254	167.077
rd100.tsp	459.557	623.765	254.192
st70.tsp	260.757	TIME LIMIT	60.892
ulysses16.tsp	1.801	2.749	0.630

Tabella E.2: Tempo di esecuzione degli algoritmi esatti con time limit di 10 minuti.

	Loop (s)	**B&C Generic (s)**	**B&C (s)**	**B&C Generic + Patching (s)**	**B&C + Patching (s)**
a280.tsp	13.857	9.509	4.187	8.211	4.191
ali535.tsp	251.844	119.427	TIME LIMIT	314.189	TIME LIMIT
att48.tsp	0.972	0.252	0.202	0.251	0.189
att532.tsp	313.906	376.933	600.842	567.807	TIME LIMIT
berlin52.tsp	0.141	0.179	0.063	0.191	0.051
bier127.tsp	2.439	1.213	1.283	1.749	1.291
burma14.tsp	0.066	0.034	0.031	0.031	0.021
ch130.tsp	2.411	1.031	1.474	1.771	1.159

ch150.tsp	5.951	1.841	3.211	2.679	2.491
d198.tsp	26.099	6.578	39.322	11.712	13.501
d493.tsp	317.945	266.471	TIME LIMIT	179.002	TIME LIMIT
d657.tsp	458.962	TIME LIMIT	TIME LIMIT	TIME LIMIT	TIME LIMIT
eil51.tsp	0.749	0.172	0.167	0.156	0.121
eil76.tsp	0.285	0.185	0.094	0.212	0.113
eil101.tsp	1.081	0.261	0.271	0.354	0.269
fl417.tsp	209.508	306.274	TIME LIMIT	122.981	TIME LIMIT
gil262.tsp	26.517	30.878	18.357	13.989	27.911
gr96.tsp	2.131	0.765	0.421	0.551	0.708
gr137.tsp	3.428	0.947	1.064	1.121	1.121
gr202.tsp	19.909	4.104	6.073	5.531	6.329
gr229.tsp	16.537	14.832	4.919	12.648	9.122
gr431.tsp	72.726	35.762	46.931	46.832	86.982
gr666.tsp	TIME LIMIT	190.031	TIME LIMIT	TIME LIMIT	TIME LIMIT
kroA100.tsp	2.011	0.829	0.861	0.651	0.882
kroA150.tsp	10.519	2.603	2.796	3.412	2.271
kroA200.tsp	24.679	13.581	34.077	8.682	74.918
kroB100.tsp	4.225	1.552	0.869	1.271	0.659
kroB150.tsp	14.636	4.824	5.791	4.472	6.111
kroB200.tsp	12.244	7.161	6.088	10.427	8.239
kroC100.tsp	2.628	0.739	0.631	0.711	0.769
kroD100.tsp	2.113	0.496	0.832	0.673	0.709
kroE100.tsp	2.477	1.357	0.681	0.948	0.931
lin105.tsp	1.674	0.642	0.472	0.311	0.419
lin318.tsp	47.514	25.434	37.783	35.308	100.161
p654.tsp	TIME LIMIT	157.174	85.571	TIME LIMIT	308.872
pcb442.tsp	243.732	15.792	36.125	22.309	35.702
pr76.tsp	4.225	2.546	4.716	1.921	9.161
pr107.tsp	0.614	0.112	0.081	0.131	0.081
pr124.tsp	8.843	1.707	1.149	2.021	1.721
pr136.tsp	2.754	1.582	1.093	1.409	0.842
pr144.tsp	11.239	3.109	2.891	3.848	3.149
pr152.tsp	5.479	4.995	2.038	4.842	2.771
pr226.tsp	66.161	17.555	9.497	14.279	9.422
pr299.tsp	108.237	125.979	TIME LIMIT	51.142	TIME LIMIT
pr439.tsp	409.73	501.036	TIME LIMIT	362.521	TIME LIMIT
rat99.tsp	1.548	0.345	0.532	0.321	0.739
rat195.tsp	37.502	12.853	12.693	10.908	23.791
rat575.tsp	391.694	281.827	TIME LIMIT	352.712	TIME LIMIT
rat783.tsp	222.909	381.214	TIME LIMIT	TIME LIMIT	TIME LIMIT
rd100.tsp	2.306	1.122	0.573	0.684	0.968
rd400.tsp	128.746	80.335	TIME LIMIT	46.511	601.14
st70.tsp	0.466	0.193	0.217	0.232	0.221
u159.tsp	3.285	2.096	1.374	1.838	2.012
u574.tsp	157.913	161.905	TIME LIMIT	194.281	TIME LIMIT
u724.tsp	440.595	TIME LIMIT	TIME LIMIT	TIME LIMIT	TIME LIMIT
ulysses16.tsp	0.232	0.121	0.047	0.051	0.039
ulysses22.tsp	0.186	0.067	0.031	0.081	0.041

Tabella E.3: Tempo di esecuzione dell'algoritmo loop, con preprocessamento euristico, al variare del gap relativo e con time limit di 10 minuti.

	Gap		
	0.1	**0.01**	**DEFAULT**
a280.tsp	27.411	11.588	13.857
ali535.tsp	291.323	349.553	251.844
att48.tsp	0.684	0.629	0.972
att532.tsp	281.506	306.981	313.906
berlin52.tsp	0.166	0.141	0.141
bier127.tsp	2.330	2.918	2.439
burma14.tsp	0.121	0.099	0.066
ch130.tsp	2.624	2.360	2.411
ch150.tsp	6.972	7.128	5.951
d198.tsp	19.695	20.837	26.099
d493.tsp	330.015	241.929	317.945
d657.tsp	TIME LIMIT	445.697	458.962
eil101.tsp	1.173	0.805	1.080
eil51.tsp	0.566	0.471	0.749
eil76.tsp	0.342	0.259	0.285
fl417.tsp	217.655	313.192	209.508
gil262.tsp	22.302	25.560	26.517
gr137.tsp	5.941	4.857	3.428
gr202.tsp	19.001	16.674	19.909
gr229.tsp	12.574	10.783	16.537
gr431.tsp	91.827	66.614	72.726
gr666.tsp	486.959	516.687	TIME LIMIT
gr96.tsp	3.299	2.167	2.131
kroA100.tsp	2.245	3.097	2.011
kroA150.tsp	8.224	7.761	10.519
kroA200.tsp	31.376	29.324	24.679
kroB100.tsp	2.940	4.361	4.225
kroB150.tsp	12.204	13.437	14.636
kroB200.tsp	13.463	8.076	12.244
kroC100.tsp	2.561	2.196	2.628
kroD100.tsp	2.699	2.293	2.113
kroE100.tsp	3.065	1.997	2.477
lin105.tsp	1.54	1.829	1.674
lin318.tsp	56.505	49.786	47.514
p654.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT
pcb442.tsp	251.024	266.387	243.732
pr107.tsp	0.551	0.555	0.614
pr124.tsp	8.394	7.002	8.840
pr136.tsp	4.676	2.741	2.750
pr144.tsp	5.316	11.016	11.239
pr152.tsp	4.719	4.863	5.480
pr226.tsp	32.678	43.278	66.161
pr299.tsp	102.936	109.369	108.237
pr439.tsp	355.795	476.181	409.73
pr76.tsp	5.331	4.867	4.225
rat195.tsp	32.627	36.468	37.502

rat575.tsp	305.992	417.810	391.694
rat783.tsp	310.511	283.290	222.909
rat99.tsp	2.178	1.420	1.548
rd100.tsp	1.932	1.900	2.306
rd400.tsp	110.792	133.082	128.746
st70.tsp	0.467	0.411	0.466
u159.tsp	2.439	3.145	3.285
u574.tsp	285.595	245.335	157.913
u724.tsp	TIME LIMIT	551.932	440.595
ulysses16.tsp	0.191	0.158	0.232
ulysses22.tsp	0.176	0.146	0.186

Tabella E.4: Tempo di esecuzione dell'algoritmo loop al variare del random seed e con time limit di 10 minuti.

	Seed			
	100	**250**	**500**	**1000**
a280.tsp	16.556	17.365	14.718	16.114
ali535.tsp	237.82	244.416	204.061	268.006
att48.tsp	0.664	0.613	0.695	0.605
att532.tsp	337.888	349.073	306.43	317.942
berlin52.tsp	0.179	0.202	0.164	0.184
bier127.tsp	2.425	2.557	2.704	2.63
burma14.tsp	0.065	0.121	0.066	0.06
ch130.tsp	2.444	2.519	2.636	2.582
ch150.tsp	5.216	5.228	5.446	5.286
d493.tsp	298.674	256.219	268.271	292.941
d657.tsp	529.713	476.444	511.532	443.609
eil51.tsp	0.497	0.505	0.574	0.614
eil101.tsp	0.973	1.003	1.476	1.005
eil76.tsp	0.291	0.288	0.287	0.346
fl417.tsp	210.629	216.737	228.297	237.123
gil262.tsp	25.446	21.953	23.957	25.757
gr96.tsp	2.361	2.354	2.35	2.351
gr137.tsp	3.328	3.512	3.4	3.712
gr202.tsp	19.77	20.923	18.47	20.154
gr229.tsp	18.043	18.269	17.369	17.273
gr431.tsp	64.513	71.89	63.802	76.604
gr666.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT	TIME LIMIT
kroA100.tsp	2.198	2.468	2.154	2.263
kroA150.tsp	10.741	11.431	9.918	11.854
kroA200.tsp	26.872	23.063	24.315	31.149
kroB100.tsp	4.229	4.456	4.568	4.329
kroB150.tsp	14.127	13.261	14.596	15.419
kroB200.tsp	11.209	10.976	11.562	14.202
kroC100.tsp	2.441	2.694	2.361	2.578
kroD100.tsp	2.282	2.33	2.384	2.289
kroE100.tsp	2.224	2.204	2.555	2.306
lin105.tsp	1.623	1.608	1.596	1.783
lin318.tsp	43.449	50.029	47.338	54.559

p654.tsp	TIME LIMIT	TIME LIMIT	TIME LIMIT	TIME LIMIT
pcb442.tsp	232.039	232.068	238.458	233.961
pr76.tsp	4.827	4.587	4.213	4.625
pr107.tsp	0.488	0.481	0.587	0.502
pr124.tsp	9.863	11.128	10.569	10.933
pr136.tsp	2.71	2.653	2.412	2.897
pr144.tsp	11.542	11.505	10.922	11.568
pr152.tsp	6.568	5.744	5.657	6.087
pr299.tsp	94.165	98.643	97.54	90.642
pr439.tsp	437.816	434.683	463.188	459.081
rat195.tsp	39.787	39.102	42.18	40.973
rat575.tsp	403.654	385.722	402.831	426.853
rat783.tsp	230.542	254.016	214.378	218.696
rat99.tsp	1.571	1.547	1.569	1.52
rd100.tsp	2.189	2.086	2.119	2.037
rd400.tsp	112.157	114.066	115.761	108.751
st70.tsp	0.406	0.422	0.463	0.448
u159.tsp	3.771	3.376	3.579	3.489
u574.tsp	187.302	189.366	165.305	172.686
u724.tsp	454.542	423.676	542.278	428.793
ulysses16.tsp	0.164	0.214	0.137	0.149
ulysses22.tsp	0.145	0.127	0.131	0.133

Tabella E.5: Costo della soluzione ottenuta mediante algoritmi math-euristici con time limit di 10 minuti.

	Soft Fixing	**Hard Fixing**	**Generic Soft Fixing**	**Generic Hard Fixing**
a280.tsp	2724.77	2586.77	2608.26	2586.77
att532.tsp	1147110.24	94035.59	386595.10	87368.30
bier127.tsp	118293.52	118293.52	118293.52	118293.52
d198.tsp	15822.50	15808.65	15808.65	15808.65
d493.tsp	246364.95	35043.38	244907.83	35018.92
eil76.tsp	544.37	544.37	544.37	544.37
eil101.tsp	640.21	640.21	640.21	640.21
fl417.tsp	309353.68	13851.95	235146.98	11966.50
gr137.tsp	706.29	706.29	706.29	706.29
gr202.tsp	486.50	486.35	486.78	486.35
lin105.tsp	14383.00	14383.00	14383.00	14383.00
lin318.tsp	44649.54	42042.54	42258.85	42275.83
pcb442.tsp	50783.55	50783.55	177301.04	50783.55
pr144.tsp	58535.22	58535.22	58535.22	58535.22
pr299.tsp	48469.13	48323.36	48226.92	48194.92
pr439.tsp	107537.87	108632.65	501047.16	107332.47
rat575.tsp	84631.45	6934.26	7533.38	6799.55
rd400.tsp	15290.98	15285.50	23080.53	15275.98
u159.tsp	42075.67	42075.67	42075.67	42075.67

Tabella E.6: Costo della soluzione ottenuta mediante Multistart, generando 40 possibili soluzioni in multithreading e restituendo la migliore di queste.

	Insertion	Insertion + GRASP	Nearest Neighborhod	Nearest Neighborhod + GRASP
a280.tsp	2817.92	2796.68	2772.51	2784.32
ali535.tsp	2186.59	2194.68	2125.32	2156.74
att532.tsp	95278.72	95278.72	92197.86	92571.30
d1291.tsp	55837.31	55837.31	54471.31	55255.14
d1665.tsp	67388.14	67847.16	66719.25	67200.25
d2103.tsp	83354.07	83354.07	82445.24	84145.24
d493.tsp	37991.33	37937.91	36748.84	37047.52
d657.tsp	54818.10	54524.77	51685.30	52341.20
dsj1000.tsp	20741407.35	20657356.07	20359263.48	20214411.67
fl1400.tsp	21580.66	21646.28	21291.85	21301.27
fl1577.tsp	24438.44	24405.16	23307.58	23392.2
fl417.tsp	12706.85	12712.88	12326.09	12278.76
gil262.tsp	2614.51	2590.11	2521.21	2535.53
gr431.tsp	2046.59	2046.59	2044.25	2060.65
gr666.tsp	3374.16	3377.25	3317.72	3317.72
lin318.tsp	45986.47	45926.30	45532.06	45480.76
nrw1379.tsp	62273.17	62295.46	60523.18	60843.52
p654.tsp	38113.48	38255.21	35381.28	35883.25
pcb1173.tsp	62190.34	62811.54	62436.66	62787.08
pcb442.tsp	55455.11	55414.92	53002.57	52439.6
pr1002.tsp	285476.34	284900.40	278473.49	280469.29
pr299.tsp	52751.57	52282.79	50920.49	52446.53
pr439.tsp	117315.31	117717.59	113483.70	115637.95
rat575.tsp	7421.49	7420.70	7184.38	7206.17
rat783.tsp	9670.53	9664.22	9418.27	9508.13
rd400.tsp	16649.07	16515.94	16307.31	16432.46
rl1304.tsp	282135.94	283443.11	274723.68	274723.68
rl1323.tsp	297641.77	301249.17	290628.52	291343.66
rl1889.tsp	352153.29	352256.86	339944.06	346590.70
u1060.tsp	251325.95	250829.17	241584.55	242756.36
u1432.tsp	168218.63	167766.50	165972.05	166245.09
u1817.tsp	63120.91	63120.91	62143.87	62035.52
u574.tsp	40965.17	41019.72	39122.03	39867.53
u724.tsp	47088.64	47271.16	44935.42	44921.80
vm1084.tsp	260985.29	261148.58	256049.81	260414.62
vm1748.tsp	375447.73	372341.92	358951.32	358951.32

Tabella E.7: Costo della soluzione ottenuta mediante algoritmi euristici con time limit di 10 minuti.

	Genetic	Hybrid VNS	Hybrid VNS Uniform	Multistart	Simulated Annealing	Tabu Search
a280.tsp	2690.74	2678.60	2661.63	2695.19	2597.20	2653.86
bier127.tsp	118336.91	118639.82	119269.09	120322.98	118336.91	118722.26
ch130.tsp	6127.97	6153.70	6169.78	6232.01	6119.81	6452.45
ch150.tsp	6580.71	6608.86	6617.40	6630.18	6540.62	6939.93
d198.tsp	15935.33	15872.40	15941.29	16079.85	15898.05	15823.36
gil262.tsp	2454.80	2413.46	2449.23	2486.13	2391.00	2570.8
gr137.tsp	709.06	706.29	706.69	719.73	707.55	719.92
kroA150.tsp	26724.72	26715.75	26733.47	27004.19	26583.71	29872.56
kroA200.tsp	29621.32	29514.53	29654.05	29746.61	29470.83	29764.78
kroB150.tsp	26349.62	26243.64	26563.59	26906.92	26199.03	27428.81
kroB200.tsp	30024.80	29448.20	30012.64	30865.99	29487.73	33509.93
pr124.tsp	59074.80	59030.74	59030.74	59408.90	60088.84	60805.78
pr136.tsp	97551.40	96890.77	97919.03	99934.70	97108.59	104028.67
pr144.tsp	58568.77	58535.22	58535.22	58673.98	58761.43	58587.14
pr152.tsp	73687.11	73821.25	73844.13	74608.12	74022.66	75746.97
pr226.tsp	80644.46	80479.82	80612.15	80849.50	80570.65	84705.91
pr264.tsp	50217.59	50125.26	50712.35	51695.67	49203.39	51041.27
pr299.tsp	50274.10	49930.99	49572.62	50571.52	48667.16	51996.17
rd400.tsp	16118.40	15572.25	16025.16	16017.21	15610.31	16433.82
u159.tsp	42075.67	42075.67	42075.67	42874.46	42435.02	47110.63

Tabella E.8: Costo della soluzione ottenuta mediante algoritmi euristici con time limit di 10 minuti.

	Hybrid VNS	Hybrid VNS Uniform	Tabu Search	Simulated Annealing	Multistart
d1291.tsp	55579.70	54622.05	54893.90	54309.32	54284.98
d1655.tsp	66661.25	66694.86	66361.42	65102.31	66410.23
d2103.tsp	82371.89	82225.44	82288.44	82037.22	82693.33
dsj1000.tsp	20252290.51	19767771.54	20290495.55	20552389.00	20016586.26
fl1400.tsp	21508.98	20932.21	21727.11	21306.06	21224.34
fl1577.tsp	24382.62	23287.37	24140.02	23875.58	23164.80
nrw1379.tsp	60399.89	60456.74	60205.58	59350.93	60114.27
pcb1173.tsp	61446.54	60073.98	60164.11	59879.89	61700.28
pr1002.tsp	275153.89	277315.21	270890.66	270671.89	272345.57
pr2392.tsp	408313.97	406410.65	409085.65	404233.03	405488.72
rl1304.tsp	282035.92	266352.94	278936.04	275343.16	274723.68
rl1323.tsp	289177.86	282450.23	290383.43	286016.11	289391.20
rl1889.tsp	345043.74	339914.33	346784.47	344552.56	340655.74
u1060.tsp	237825.17	236736.18	239277.18	233324.89	239326.80
u1432.tsp	164646.37	162328.27	162102.60	162103.55	165815.35
u1817.tsp	62460.55	61316.67	62030.58	62083.04	61739.76
u2152.tsp	70952.18	70145.91	70100.98	70103.33	69919.59
u2319.tsp	246272.89	246234.31	244854.41	244715.40	244783.90
vm1084.tsp	254733.26	257034.98	254005.22	251366.15	254167.66
vm1748.tsp	358335.39	357576.31	362593.77	359928.19	358742.92

Bibliografia

[1] https://www.ibm.com/it-it/products/ilog-cplex-optimization-studio.

[2] https://www.fico.com/en/products/fico-xpress-optimization.

[3] https://www.gurobi.com.

[4] https://www.coin-or.org/Cbc/.

[5] http://www.math.uwaterloo.ca/tsp/concorde/index.html.

[6] http://www.math.uwaterloo.ca/tsp/data/ml/monalisa.html.

[7] H. Williams A. Orman. A survey of different integer programming formulations of the travelling salesman problem. In *Optimisation, Econometric and Financial Analysis*, pages 96–98. Springer, 2007.

[8] P. Hansen, N. Mladenovic, J. Brimberg, J. A. Moreno Pèrez, M. Gendreau, and J. Potvin. Variable neighborhood search. In *Handbook of Metaheuristics*, pages 61–86. Springer, 2010.

[9] S. Liu L. Fang, P. Chen. Particle swarm optimization with simulated annealing for tsp. 6:206–210, 2007.

[10] Fischetti M. Alcuni problemi np-difficili. In *Lezioni di Ricerca Operativa*, pages 183–185. Kindle Direct Publishing, 2018.

[11] A. Lodi M. Fischetti. Local branching. *Ser. B*, 98:23–47, 2003.

[12] A. Dewanji S. Hore, A. Chatterjee. Improving variable neighborhood search to solve the traveling salesman problem. volume 98, pages 83–91. Elsevier, 2018.