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)[4]. 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**
  è un soluzione analitica, sviluppato dall'IBM, e gratuita a livello accademico.

- **FICO® Xpress Optimization**
  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**
  è una soluzione, sviluppata dalla Gurobi Optimization, che viene rilasciata anche con una versione accademica.

- **COIN Branch and Cut solver (CBC)**
  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[1].
Nel Capitolo 3 vengono analizzati gli algoritmi euristici, sviluppati senza far uso di ILOG CPLEX.
Nel Capitolo 4 vengono invece riportari confronti, a livello temporale e di costo, delle soluzioni ottenute con i differenti algoritmi enunciati.
Nell' Appendice ??, C, D vengono descritti rispettivamente 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.
Tutte le soluzioni descritte sono state implementate in linguaggio C ed i sorgenti sono disponibili online[1].

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

Capitolo 2

Risoluzione del problema tramite CPLEX

2.1 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 **??**). 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. Entrambi i modelli descritti in questa sezione sono stati sviluppati ed il confronto dei tempi di esecuzione delle relative implementazioni viene riportato nella Sezione **??**.

2.1.1 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:

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

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

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

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

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

Esistono però due diversi modi per inserire i vincoli aggiuntivi, 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 verificarà la correttezza analizzando l'insieme di vincoli precedentemente definito e se dovesse trovarne uno violato, lo aggiunge al modello e ripete 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.

2.1.2 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:

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

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

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

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

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

$$y_{ij} \leq \ (n-1)\ x_{ij} \qquad \forall\ i, j \in V, i \neq j \tag{2.11}$$
$$y_{ii} = 0 \qquad \forall\ i \in V$$
$$y_{i1} = 0 \qquad \forall\ i \in V$$

In questo 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:

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

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

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

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

2.2 Algoritmi Esatti

2.2.1 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 risolvere tale problema, il metodo loop costruisce inizialmente il modello senza quei vincoli e li aggiunge solo in seguito durante la risoluzione del problema. Il metodo di Benders calcola una soluzione e valuta se questa rispetti tutti i vincoli non inseriti nel modello.
Nel caso in cui l'algoritmo dovesse trovarne uno che non viene rispettato, lo inserisce nel modello.
Nel caso del TSP, i vincoli in numero esponenziale sono i Subtour Elimination, che hanno la

seguente forma:

$$\sum_{e \in E(S)} x_e \leq \mid S \mid -1 \; \forall \; S \subsetneq V \; : \mid S \mid \geq 2 \tag{2.12}$$

o equivalentemente:

$$\sum_{e \in \delta(S)} x_e \geq 2 \; \forall \; S \subsetneq V \; : \mid S \mid \geq 2 \tag{2.13}$$

Definendo il modello per problema del commesso viaggiatore, vengono rimossi i vincoli di Subtour ELimination. In seguito viene risolto il problema e nel caso in cuila 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 ≥ 2 do
4 |   Aggiungi Σ_{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 vincoli di Subtour Elimination, solo nel momento in cui si presenta una loro violazione, permette di ridurre la dimensione del modello ma riduce 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 iterazione del loop.
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 viene effettuato l'algoritmo di Benders ma 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 metaeuristica ottenuta nella prima fase viene sfruttata come bound nella seconda.

2.2.2 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 calcola una nuova soluzione x^*, prima di dichiarare se è l'ottimo o di scartarla e proseguire a sviluppare i successivi rami dell'albero decisionale, applica dei tagli e degli algoritmi euristici per aggiornarla (vedi Figura 2.1). Nello sviluppo del Branch & Cut, per ciascun nodo

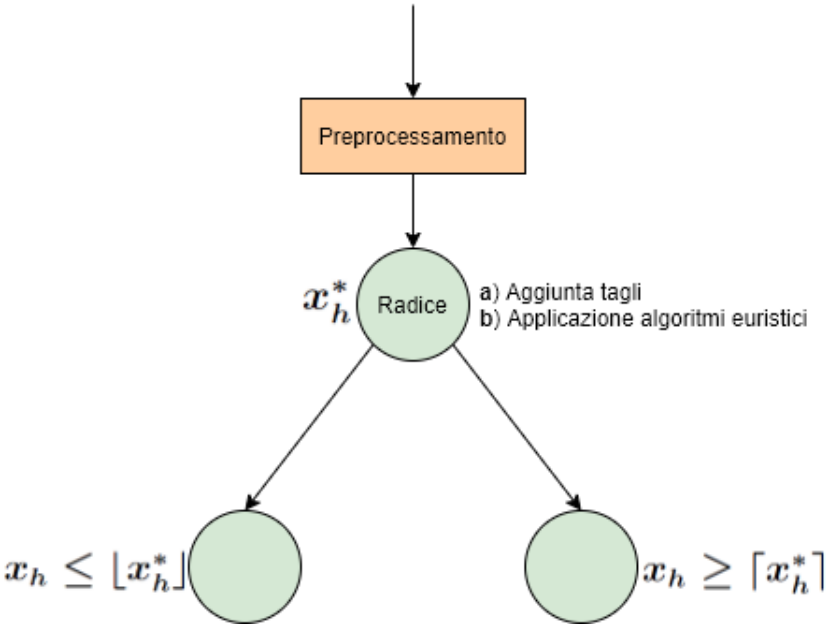


Figura 2.1: Albero decisionale del Branch and Cut

dell'albero decisionale, vengono considerati due bound:

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

- **Lower Bound**
  viene definito dal rilassamento del problema.

CPLEX permette di personalizzare i tagli da applicare nel Branch & Cut e inserendo iterativamente i Subtour Elimination relativi alle componenti connesse nella soluzione a cui verrà applicato il taglio. Per fare ciò CPLEX permette all'utente di implementarne questi vincoli mediante delle callback, *lazy constraints callback*, utilizzate per aggiungere lazy constraints al modello.
Le callback implementate vengono chiamate solo nel momento in cui deve essere aggiornato l'incumbent e se necessario aggiunge al modello i vincoli violati. Verrà quindi invocata 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" al momento 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, diventa evidente visiando i log di CPLEX.
Attraverso l'utilizzo delle callback è possibile accedere però 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.
La procedura più importante che viene bloccata è il dynamic search. Per evitare questo è possibile installare le callbacks con una modalità leggermente diversa, attraverso funzioni dette *general*.

2.2.3 Patching algorithm

Negli algoritmi analizzati nei precedenti capitoli può succedere che CPLEX, prima di restituire la soluzione ottima, computi soluzioni con più componenti connesse. Per evitare che vengano scartate senza essere sfruttate è possibile utilizzare questo semplice algoritmo euristico che si pone l'obbiettivo di convertirle in una soluzione ammissibile.
Date due componenti connesse all'interno della soluzione calcolata, queste vengono unite in una sola grazie all'eliminazione di un ramo ciascuna $\{a, a'\}$ e $\{b, b'\}$ e alla selezione di altri due rami che fungano da collegamento tra i quattro vertici selezionati ($\{a, b\}$ e $\{a', b'\}$ o $\{a, b'\}$ e $\{a', b\}$) (vedi figura 2.2). Per scegliere quale ramo per ogni componente connessa sia più conveniente eli-

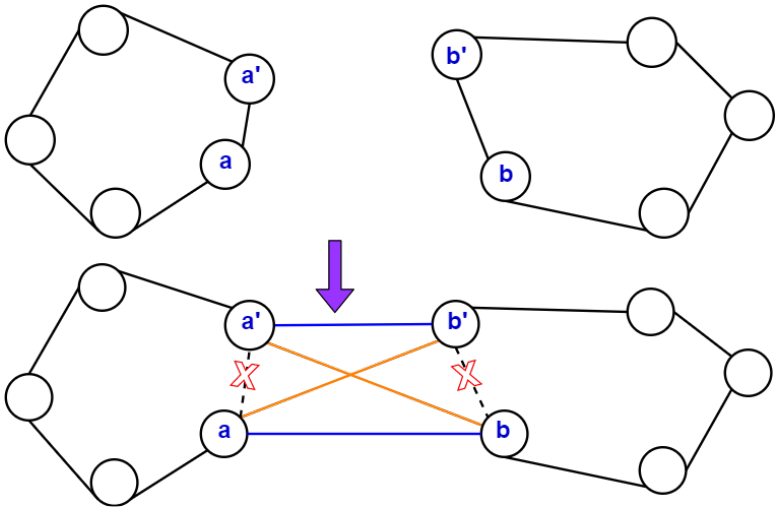


Figura 2.2: Esempio di patching

minare e con quale sia più conveniente sostituirlo, è necessario minimizzare la variazione di costo che quest'operazione comporterebbe, cioè scegliere il minimo tra:

$$min \begin{cases} \Delta\,(a,b) & = c_{ab'} + c_{ba'} - c_{aa'} - c_{bb'} \\ \Delta'\,(a,b) & = c_{ab} + c_{b'a'} - c_{aa'} - c_{bb'} \end{cases} \qquad \forall\, a, b \in V : comp(a) \neq comp(b)$$

L'operazione di fusione di due componenti connesse può essere ripetuta finché la soluzione non diventa ammissibile. In questo modo, al termine, è possibile restituire una soluzione accettabile, ma senza garanzia che sia ottima.
Nella nostra implementazione è stato scelto di mantenere invariata ad ogni iterazione una delle componenti connesse e di espanderla fondendola a quella più vicina. Grazie a questa scelta viene minimizzato il numero di rami per cui è necessario modificare la struttura dati che li memorizza.
Per poter implementare l'algoritmo è necessario utilizzare due diversi tipi di callback messe a disposizione da CPLEX. La prima appartiene alla tipologia delle lazyconstraintcallback 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 in una struttura dati accessibile anche dalla seconda callback dell'utente. Per garantire che la user-callback sia thread-safe, la struttura dati nominata è stata organizzata di modo tale che ogni thread accedesse ad una sua specifica porzione.
La seconda callback necessaria è un heuristic callback e permette all'utente di suggerire a CPLEX una soluzione da cui proseguire la computazione. Questa soluzione sarà quella memorizzata nella struttura dati dalla prima callback e verrà utilizzata dal programma solo nel caso in cui sia migliore dell'incumbent attuale.
Utilizzando, invece, le generic callback non è necessario implementare due diverse user-callback. è sufficiente invocare la callback in due diversi contesti (uno per ricevere la soluzione di cplex, (CPX_CALLBACKCONTEXT_CANDIDATE) e l'altro per suggerire il risultato del patching (CPX_CALLBACKCONTEXT_LOCAL_PROGRESS)). I due diversi casi andranno gestiti dal-

l'interno della funzione stessa.
Complessivamente il costo di quest'algoritmo è $O(n^2)$, dove n è il numero di nodi.(verificare)

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
```

2.3 Algoritmi Math-Heuristici

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 ma sfruttano la programmazione matematica, introducendo nuovi vincoli al modello. L'algoritmo che maggiormente rappresenta questo metodo è il Soft Fixing (vedi Sezione 2.3.2).
Durante la computazione della soluzione CPLEX utilizza diversi algoritmi euristici e math-euristici e variando alcuni parametri, è possibile variare la frequenza con cui vengono applicati o il tempo a loro dedicato.

2.3.1 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;

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 variabile 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 di diverse percentuali che si desidera utilizzare. All'ultima iterazione il time limit viene ricalcolato in base al tempo rimanente al programma, rispetto alla deadline complessiva inserita dall'utente. Ad ogni computazione il costo la 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 iterazioni, in modo da aumentare i gradi di libertà nella computazione della soluzione per CPLEX.
La selezione in maniera randomica degli archi nella **Fase 3**, garantisce che l'algoritmo termini

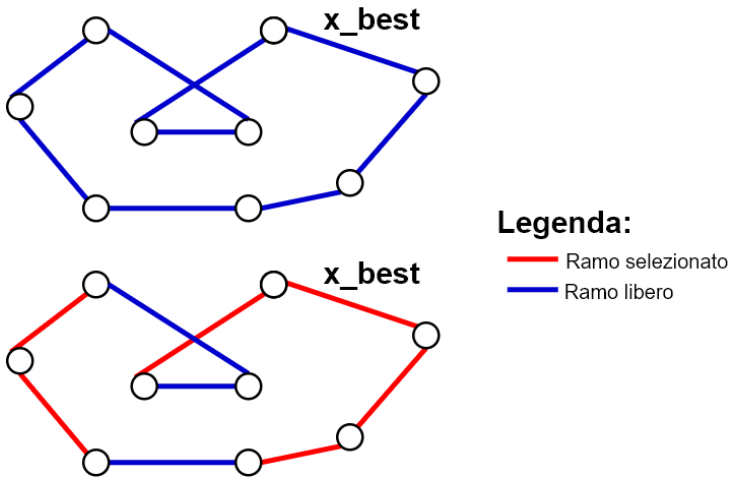


Figura 2.3: Selezione rami

poichè non vengano fissate sempre le stesse variabili. Particolare attenzione deve essere posta al fatto di lasciare nell'insieme dei rami scelti randomicamente solo quelli selezionate 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 leght(percentage)] then
8              Aggiungi x_best[j] to S where S = {edges to fix}
9      forall x_{i,j} ∈ S do
10         x_{i,j} ←1
11     n ←n + 1
```

2.3.2 Soft Fixing

Il metodo seguente fa utilizzo di vincoli aggiuntivi, detti di **Local Branching**. L'approccio utilizzato è simile a quello dell'Hard Fixing, ma la scelta delle variabili da imporre a 1 non viene fatta in maniera randomica ma viene lasciata a CPLEX.

Partendo da una soluzione intera ammissibile del TSP x^H, viene aggiunto un vincolo sulle variabili con valore 1 in x^H:

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

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, 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 e rappresenta i gradi di libertà di CPLEX nel calcolare la nuova soluzione.
Ad ogni iterazione viene aggiunto un nuovo vincolo di local branching, basato sull'attuale soluzione restituita da CPLEX, e viene rimosso il vincolo aggiunto nell'iterazione precedente.
Non scegliendo in maniera randomico 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 **local branching**, sarebbero gli stessi dell'iterazione precedente. Per ovviare tale problema, il valore di k viene inizializzata a 2 e, nel caso in cui non dovesse migliorare la soluzione, viene incrementata.
Da dati sperimentali, si è appurato come questo metodo aiuti CPLEX a convergere più velocemente alla soluzione ottima e che valori di k superiori a 20 non aiutino a raggiungere risultati migliori.
L'aggiunta di un local-branching permette di analizzare in maniera più semplice e veloce lo spazio delle soluzioni. Normalmente per farlo, dovrebbere essere enumerati gli elementi di questo spazio ma ciò comporterebbe la creazione di un numero molto elevato di possibli soluzioni per via della NP-difficoltà del TSP.
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 x^H (vedi Figura 2.4).
Se, invece del local branching, venisse utilizzata l'enumerazione delle soluzioni, dovrebbero essere

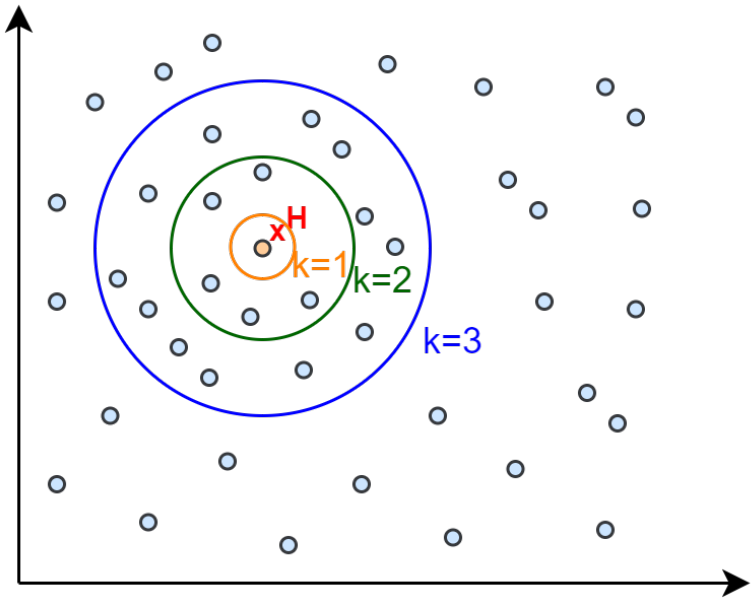


Figura 2.4: Spazio delle soluzioni e distanza di Hamming.

generate per **k** generico 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 TSP. Di seguito viene riportato l'aprroccio da seguire per generare tutte le soluzioni a distanza minore o uguale di R dalla soluzione euristica di partenza x_H:

$$min\{c^T x : \quad Ax = b,\ x \in \{0,1\}^n\} \tag{2.14}$$

$$\sum_{j \in E: {x_j}^H = 0} x_j \ + \sum_{j \in E: {x_j}^H = 1} 1 - x_j = \leq R \tag{2.15}$$

dove **(3.15)** rappresenta la distanza di Hamming dalla nuova soluzione computata x da x_H. L'obiettivo del Soft-fixing è cercare di migliorare il costo della soluzione, guardando quelle più vicine possibili a quella attuale. Nella **Figura 2.5** viene riportato un esempio di una possibile evoluzione dell'algoritmo nella ricerca dell'ottimo, evindenziandone le soluzioni trovate di volta in volta.

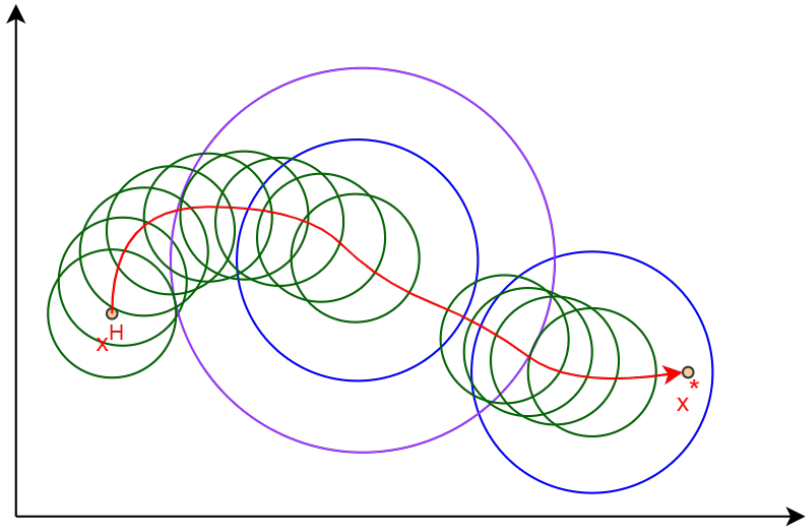


Figura 2.5: Esempio di esecuzione dell'algoritmo nello spazio delle soluzioni.

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 queste istanze la risoluzione del tableau attraverso CPLEX diventerebbe molto un'operazione molto onerosa per via dell'alto numero di variabili che verrano create e su cui verrà svolto il calcolo.
Attraverso gli algoritmi euristici, viene computata un'approssimazione della soluzione ottima e spesso però può essere sfruttata inizialmente dal risolutore CPLEX. Ad esempio questo può essere aggiunta prima della computazione, utilizzando la funzione *CPXaddmipstarts()*, o ,se già definita, può essere modificata tramite *CPXchgmipstarts()*.
Un algoritmo euristico, affinchè funzioni al meglio, 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 affinchè 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

Questo algoritmo è basato su un approccio di tipo greedy. L'algoritmo sceglie inizialmente un nodo generico tra quelli che compongono il grafo ed in seguito seleziona 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 aggiunto il ramo collega l'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. In

seguito tra queste possibli soluzioni, verrà selezionata quella di costo minore.

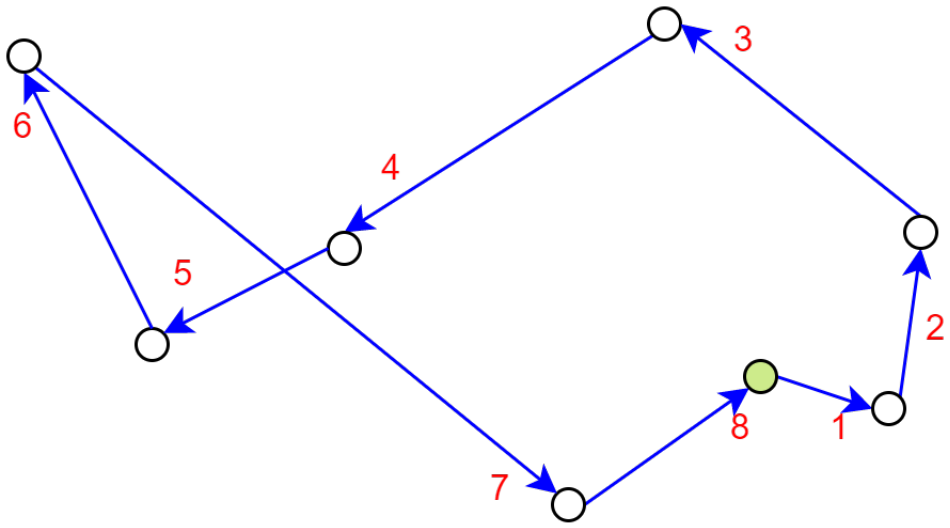


Figura 3.1: Esempio di esecuzione di Nearest Neighborhood.

3.1.2 Heuristic Insertion

L'algoritmo seguente usa un approcco simile al precedente ma prevede inizialmente la selezione di un ciclo a cui apportare 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 vegonono collegati mediante i due possibili archi orientati.

- **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):

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

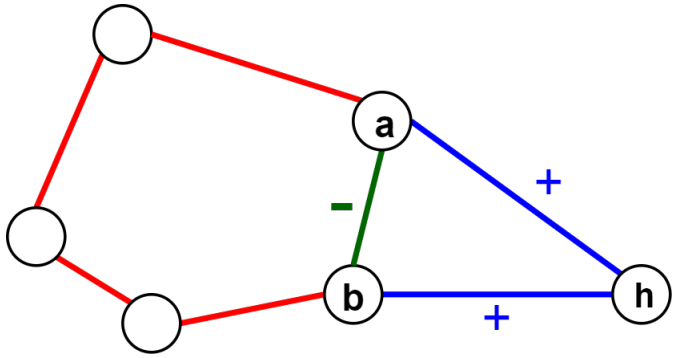


Figura 3.2: Parte del calcolo dell'extramileage del nodo **h**.

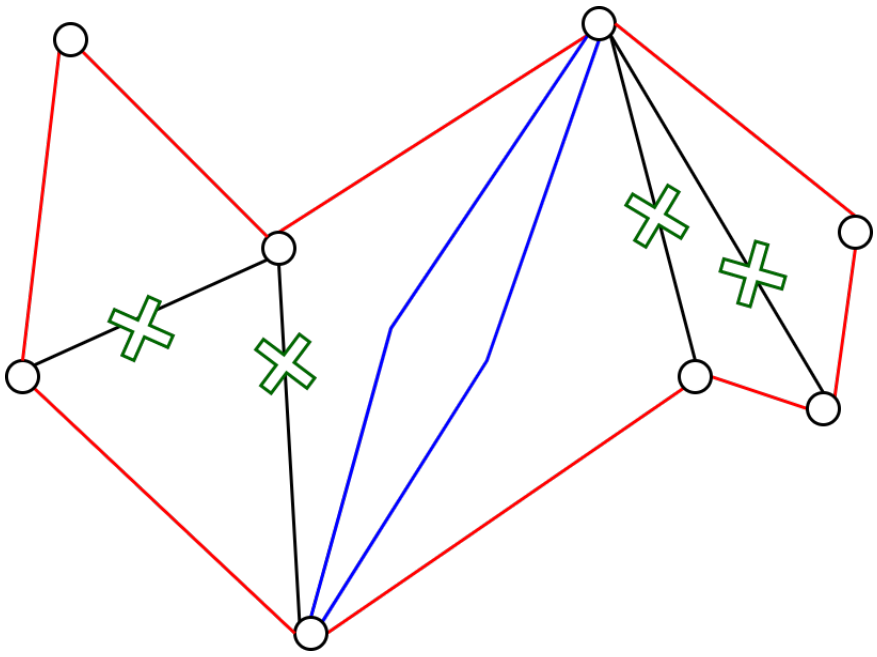


Figura 3.3: Esempio dell'applicatione di Heuristic insertion.

3.1.3 GRASP

Il metodo Greedy Adaptive Search Procedure (GASP) 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 impplementate 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 aleatoriamente 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 ??, 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 2.3.1 e 2.3.2). 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 (vedi Figura 3.4).

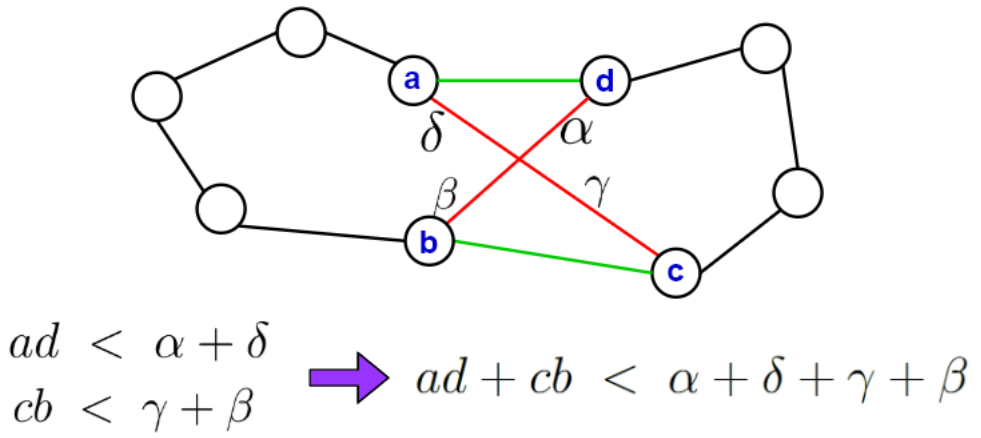


Figura 3.4: Non ottimalità di una soluzione con incroci.

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 effettutato 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.

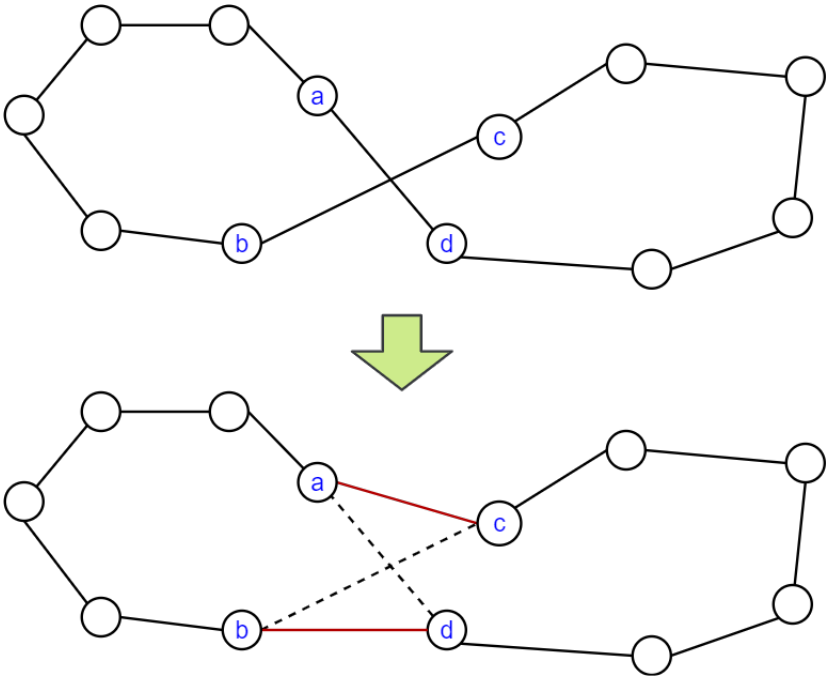


Figura 3.5: Esempio di eliminazione di un incrocio.

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.

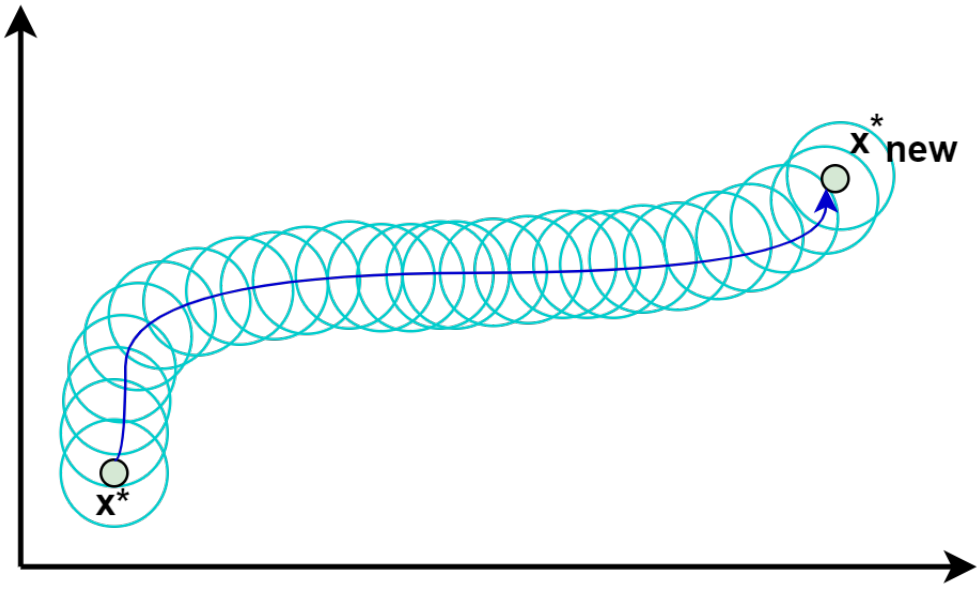


Figura 3.6: Aggiornamento della soluzione nell'intorno di 2 ottimalità.

Poichè il calcolo del Δ avviene in tempo costante e deve essere fatto per ogni coppia di rami, il tempo complessivo per la computazione è $O(n^2)$, con n nodi dell'istanza del problema.
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 algoritmo per rimuovere gli incroci all'interno del 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.

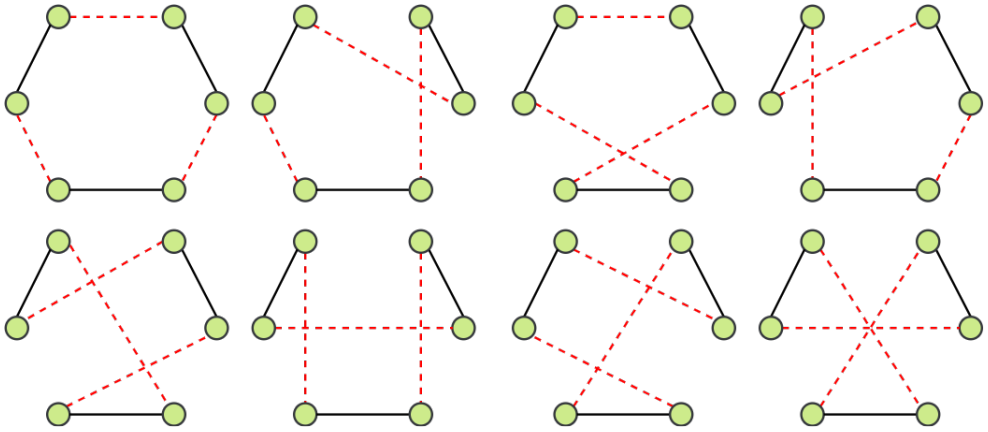


Figura 3.7: Tutte le possibili combinazioni di scambi di 3 ottimalità.

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 poichè 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 il calcolo di 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 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 plasmabili e 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 ciascuno di esse viene poi applicato un algoritmo di raffinazione e viene scelta solo quella con costo migliore tra tutte le soluzioni generate. In questo modo vengono analizzati diversi ottimi locali nello spazio delle soluzioni (vedi Figura 3.8.

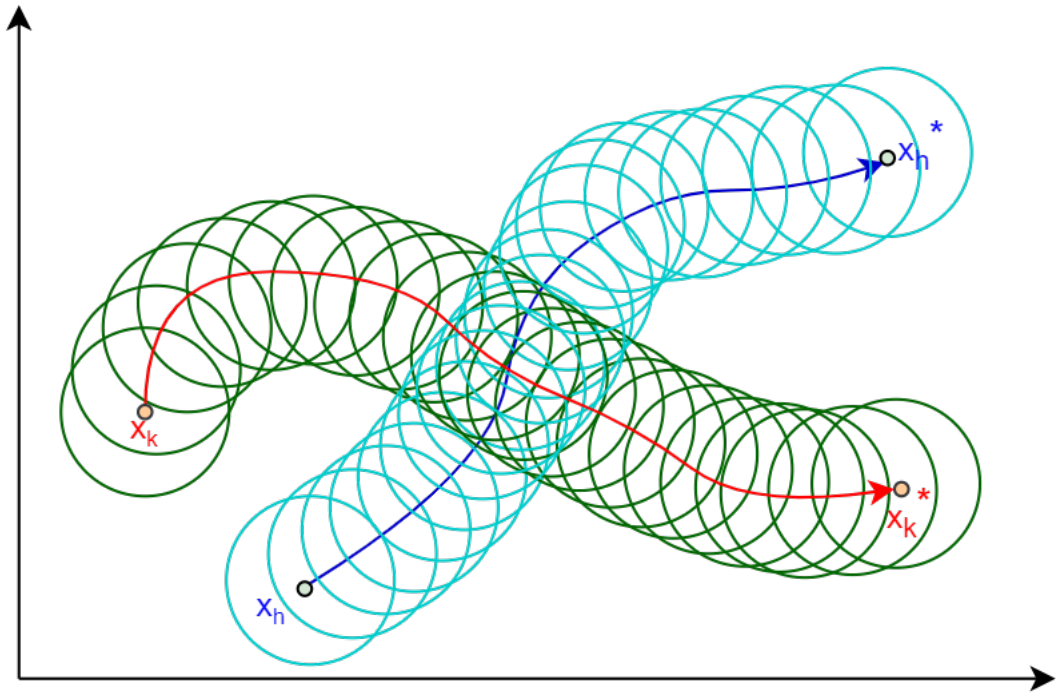


Figura 3.8: Due possibile esecuzioni di un algoritmo di raffinamento con partenze da due soluzioni diverse.

Un lato negativo di tale approccio consiste nel fatto che ogni volta che viene scelta una nuova soluzione di partenza, vengono perse le informazioni relative alla sequenza ottenuta utilizzando un altro tour iniziale. 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 archi[5]. 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).

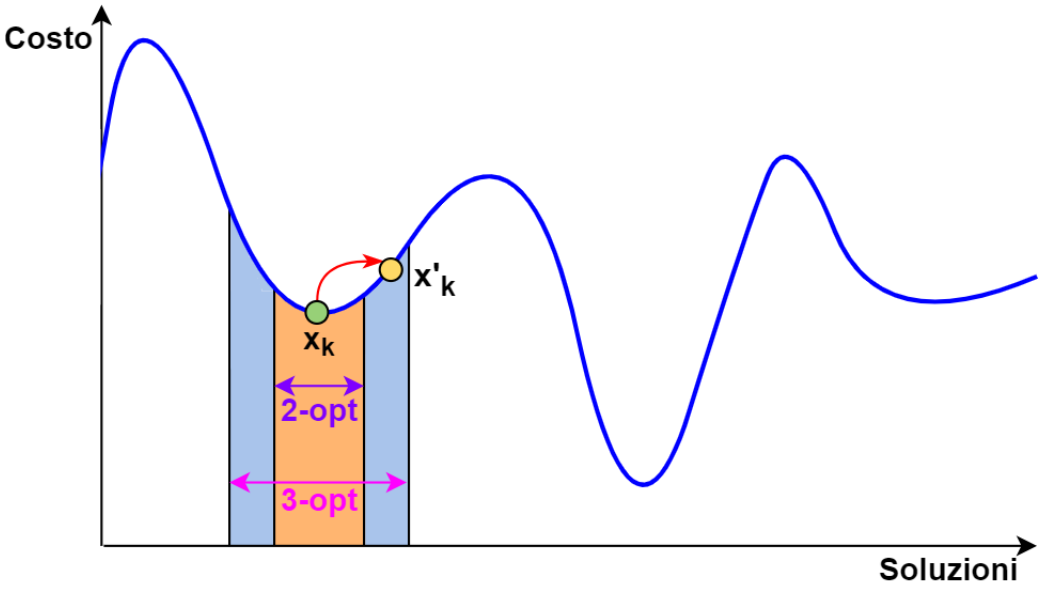


Figura 3.9: Aggiornamento di una soluzione in un minimo locale.

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[7]. Di seguito viene descritto lo pseudo-codice di tale metodo:

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: y= nuova soluzione appertenente all'intorno di k-ottimalità di x

```
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⌉ ∧ !found then
12         y ← new_random_sol(y)
13     count ← count + 1
```

Partendo dalla sequenza di visita dei nodi nella soluzione locale, passata in ingresso all'algorit-

mo, la procedura analizza tutte i circuiti 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 utilizzata 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.

la prima che seleziona uno scambio secondo questa probabilità, e la seconda che invece seleziona due elementi casuali della sequenza e li scambi secondo una distribuzione uniforme.

3.3.3 Tabu Search

Il metodo Tabu Search fu ideato da Fred W. Glover [8]. Dato un ottimo locale nello spazio delle soluzioni, l'idea di Glover permette di aggiornarla anche con una di costo più elevato, cercando di minimizzarlo. Per evitare che all'iterazione successiva si ritorni nella soluzione di partenza, viene creata una lista di "mosse vietate", detta Tabu List, che impedisca di raggiungere nuovamente lo stesso ottimo locale. In questo modo il costo della soluzione attuale aumenta rispetto all'ottimo locale di partenza per un certo numero di iterazioni, finchè non ricominci a decrescere per raggiungere un nuovo minimo. Nella nostra implementazione la lista viene riempita con i rami che vengono rimossi 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, detta *tenure*, della lista, una volta raggiunta 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 la lista viene fatta aumentare fino ad un massimo (nella nostra implementazione 1/5 del numero di nodi dell'istanza da risolvere). Nella fase di intensificazione, per lasciare un maggior numero di gradi di libertà al definizione delle nuove soluzioni, la tenure diminuisce fino ad un valore minimo impostato, che nel nostro caso corrisponde a 1/10 del numero di nodi del grafo (vedi Figura 3.10). Questo variante dell'algoritmo, implementata anche nel nostro programma, in letteratura è conosciuta con il nome di *Reactive Tabu Search*.

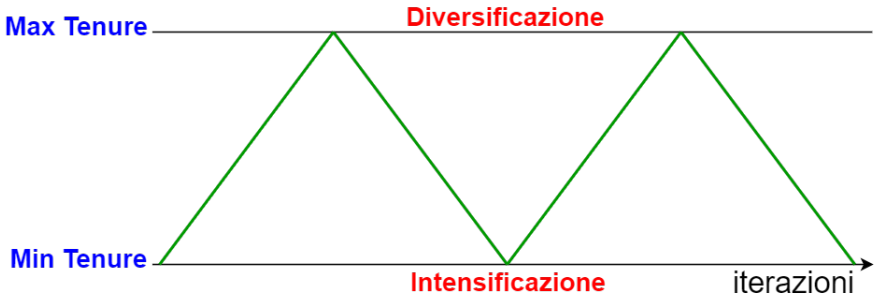


Figura 3.10: Variazione della tenure.

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, come dice il nome, è ispirato 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. Analogamente, nell'algoritmo viene scelta una funzione (generalmente esponenziale) che simula la variazione della "temperatura" T. Nella nostra implementazione abbiamo utilizzato la seguente formula [6]:

$$T = \alpha^{num\_iteration} * T\_max \ + \ T\_min$$

dove T_max è la massima temperatura iniziale, T_min è la minima temperatura che può essere raggiunta, $num\_iteration$ 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, per poter procedere con la computazione.
Ad ogni iterazione la soluzione corrente può essere aggiornata con un'altra qualsiasi interna all'intorno di 2 ottimalità, di costo minore o maggiore, con una probabilità funzione della variazioni di costo e della temperatura attuale, $f(\Delta costo, T)$. Nel nostro caso la seguente[6]:

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

Grazie a questa tecnica per la scelta dell'aggiornamento, non è necessario scandire tutto l'intorno, è sufficiente scegliere in maniera randomica 2 rami. Tali archi verranno sostituiti nella soluzione corrente secondo la funzione di probabilità (Figura 3.11). Per poter gestire al meglio i valori della probabilità e evitare che la maggior parte delle volte venissero approssimati a zero, nel nostro algoritmo sono stati riscritti in notazione scientifica, memorizzando solo l'ordine di grandezza e il coefficiente. Una nuova soluzione di costo peggiore viene, quindi, accettata nel caso in cui si verifichino entrambe le seguenti situazioni

- venga estratto per un numero di volte pari all'ordine di grandezza, un numero, nell'intervallo [0, 99], inferiore a 10;

- venga estratto, nell'intervallo [0, 10*(coefficiente della probabilità)], un numero inferiore a 10.

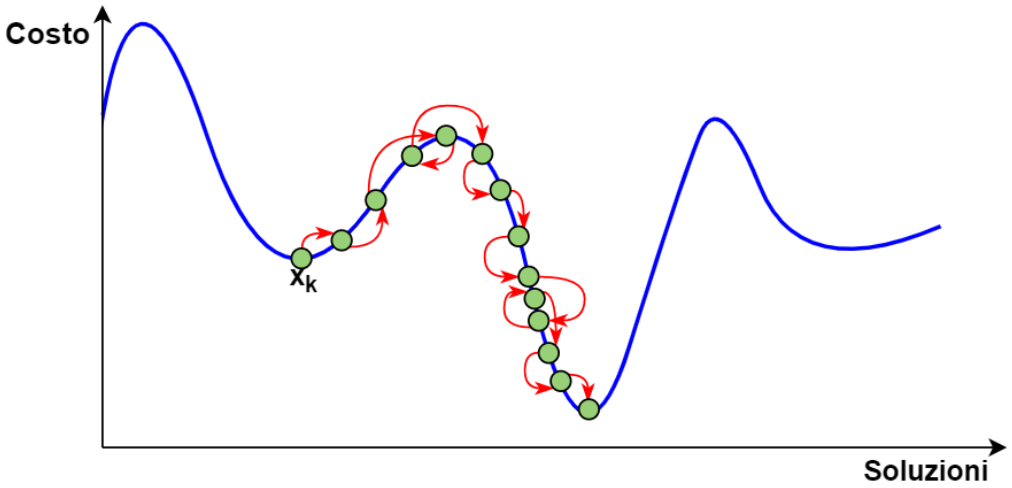


Figura 3.11: Esempio di esecuzione dell'algoritmo Simulated Annealing.

Con il procedere delle iterazioni, l'aggiornamento ad un costo peggiore avviene sempre meno frequentemente, fino ad ottenere solo aggiornamenti con soluzioni più vantaggiose. Esiste un teorema secondo il quale se la temperatura varia in maniera estremamente lenta ed è consentito effettuare un numero di iterazioni estremamente elevato, questo algoritmo garantisce di trovare l'ottimo globale. Concretamente queste ipotesi sono molto difficili da realizzare, ma è statisticamente comun-

que possibile dichiarare che l'approccio del Simulated Annealing restituisce 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 costruisce in prima battuta una serie di individui, che costituiscono una 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 certà 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 costoso 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 randomicamente 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 casuale.

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 3 == 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, poichè 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.

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 `n`= 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 $\quad \texttt{best\_cost} \leftarrow min\_cost$

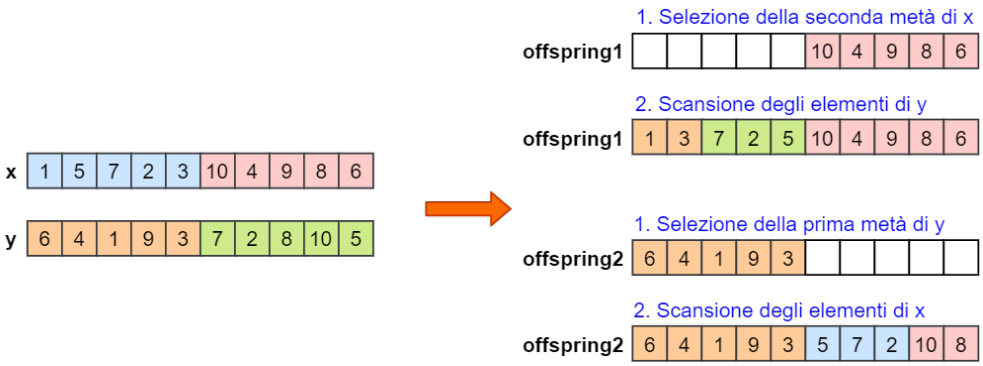


Figura 3.12: Esempio di applicazione del crossover su due istanze x e y.

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 `n` ← numero di nodi dell'istanza tsp
2 `x` = $\{x_1, ..., x_n\}$ ←$RANDOM(population)$
3 `begin` ← $RANDOM(\{1, ..., n/2\})$
4 `end` ← $RANDOM(\{n/2 + 1, ..., n\})$
5 **for** `i` ← *begin* **to** *end* **do**
6 `offspring`$[i] \leftarrow x[end - i + begin]$
7 **if** $cost(offspring) < best\_cost$ **then**
8 `best_cost` ← $cost(offspring)$

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

Capitolo 4

Performance

La metrica di confronto, utilizzata nell'analisi degli algoritmi, è il tempo complessivo di creazione e risoluzione del modello. 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).

- **Node limit**
  massimo numero di nodi risolti prima che l'algoritmo termini senza raggiungere l'ottimalità (vedi Sezione B.3).

- **Populate limit**
  massimo numero di soluzioni MIP generate per il pool di soluzioni durante ogni chiamata della procedura di popolazione (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 degli altri parametri permette invece di ottenere una soluzione meta-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 ammontare di 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 4.11).
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, più semplice da essere analizzata visivamente.

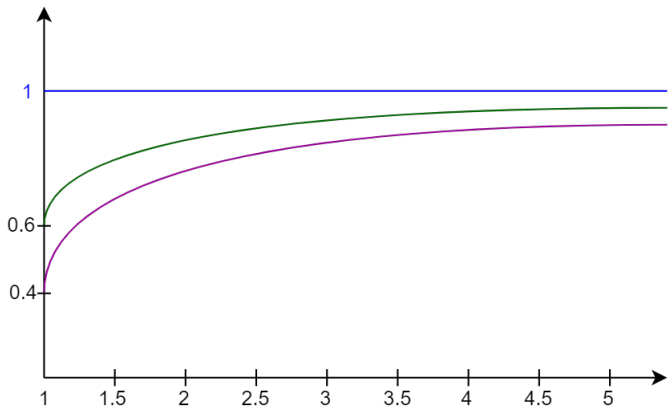


Figura 4.1: Performance profile di due algoritmi.

Per creare il performance profile degli algoritmi implementati, è stato utilizzato il programma python riportato nella Sezione D.

4.4 Analisi degli algoritmi sviluppati

4.4.1 Algoritmi esatti

4.4.1.1 Modelli simmetrici

- 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

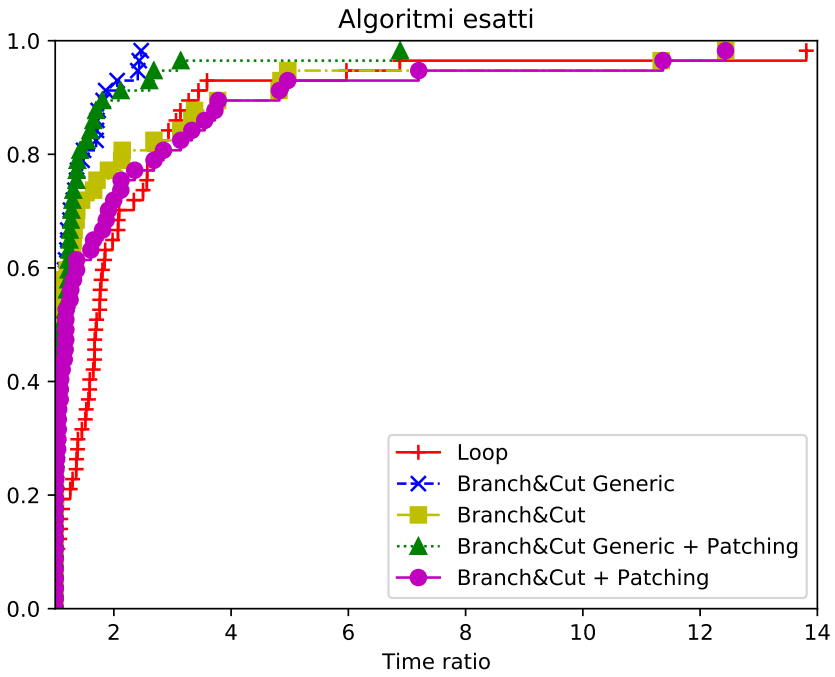


Figura 4.2: Performance profile degli algoritmi esatti.

4.4.1.2 Modelli compatti

- 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

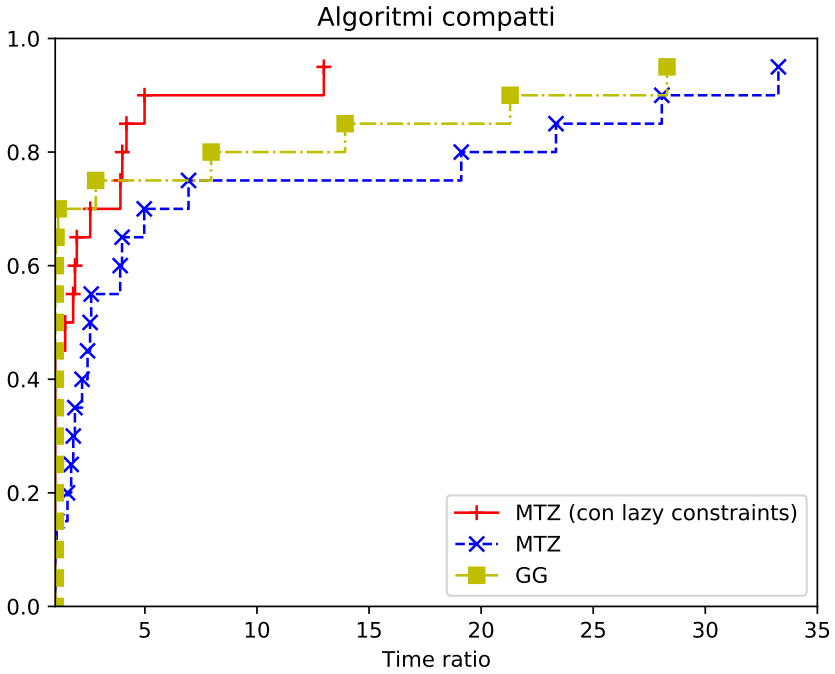


Figura 4.3: Performance profile degli algoritmi compatti.

4.4.2 Algoritmi math-euristici

- 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

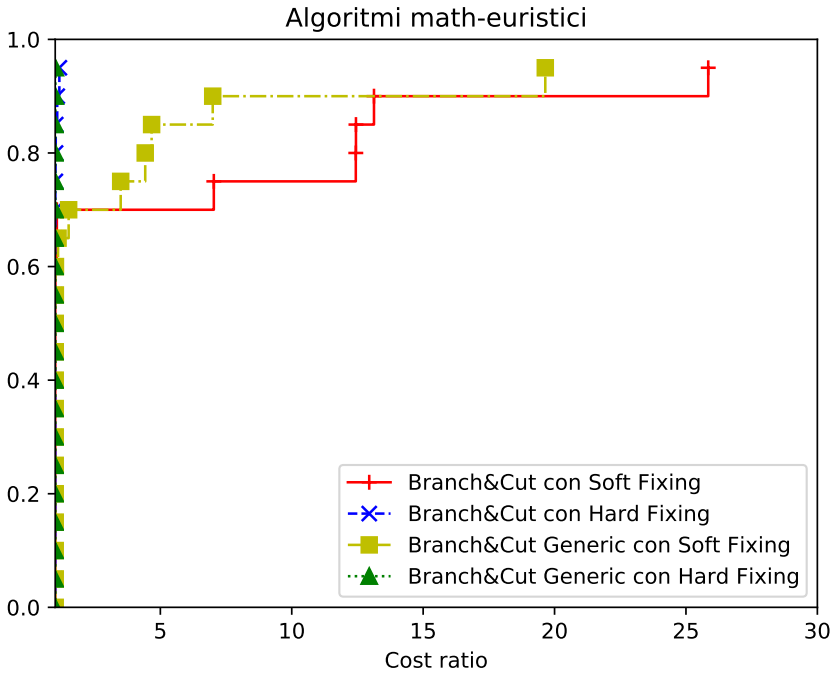


Figura 4.4: Confronto dei vari multistart in base all'algoritmo di costruzione.

4.4.3 Algoritmi euristici

4.4.3.1 Multi-start

- 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

4.4.3.2 Algoritmi meta-euristici

- d1291.tsp
- d1655.tsp
- d2103.tsp
- dsj1000.tsp
- fl1400.tsp
- fl1577.tsp
- nrw1379.tsp
- pcb1173.tsp
- pr1002.tsp
- pr2392.tsp
- rl1304.tsp
- rl1323.tsp

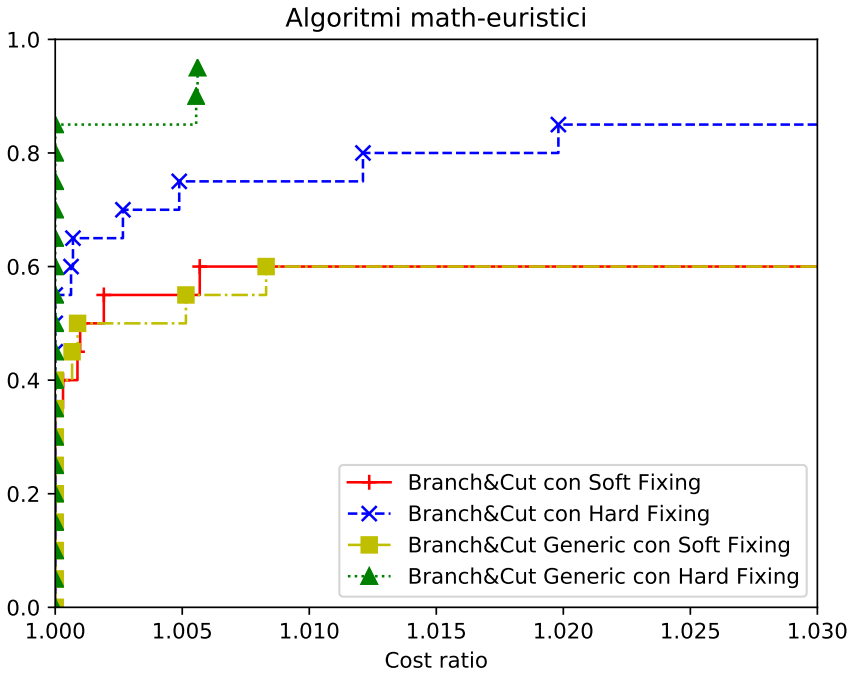


Figura 4.5: Confronto dei vari multistart in base all'algoritmo di costruzione.

- rl1889.tsp
- u1060.tsp
- u1432.tsp
- u1817.tsp
- u2152.tsp
- u2319.tsp
- vm1084.tsp
- vm1748.tsp

- 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

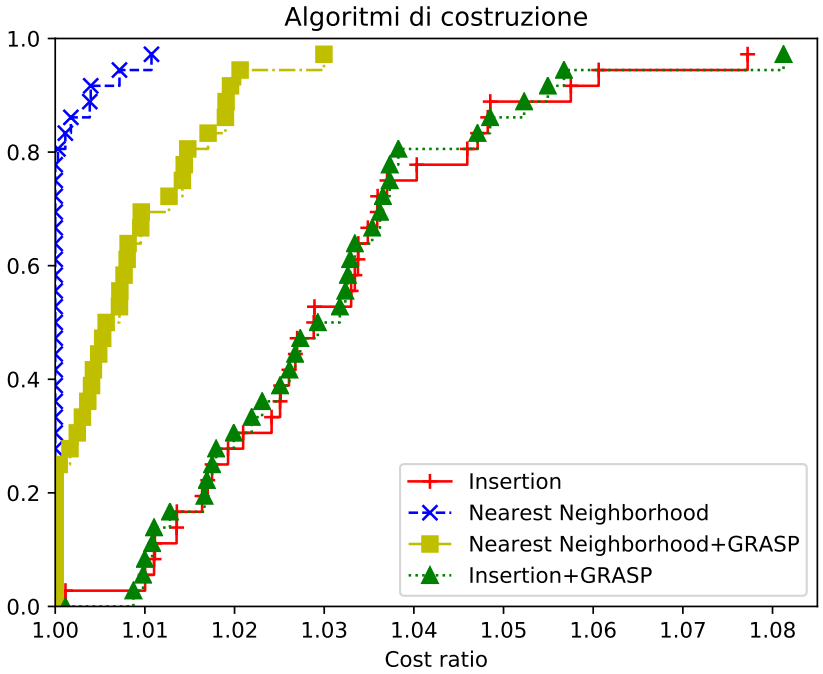


Figura 4.6: Confronto dei vari multistart in base all'algoritmo di costruzione.

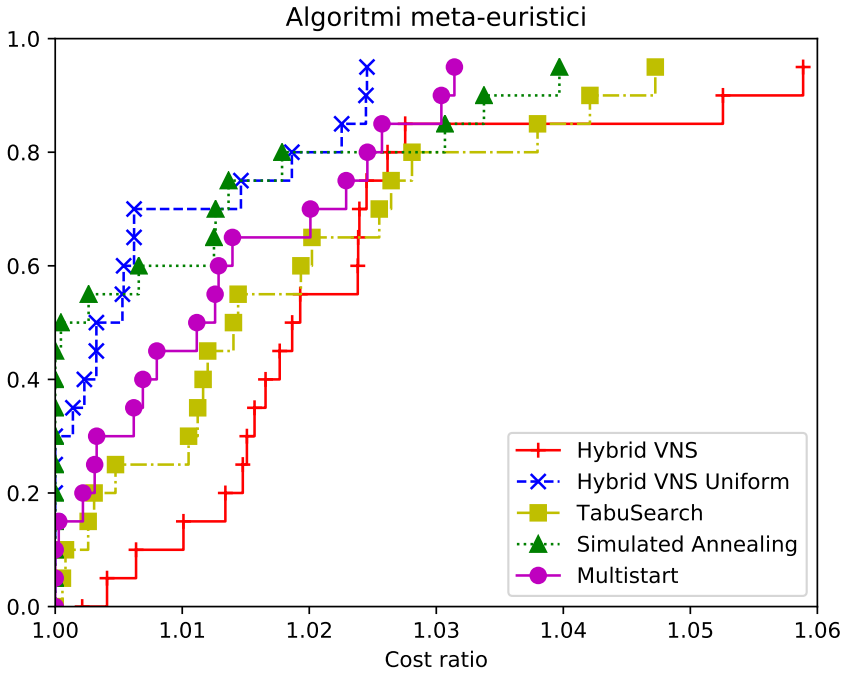


Figura 4.7: Confronto dei costi ottenuti algoritmi meta-euristici sviluppati utilizzando il Nearest Neighborhood come algoritmo di costruzione.

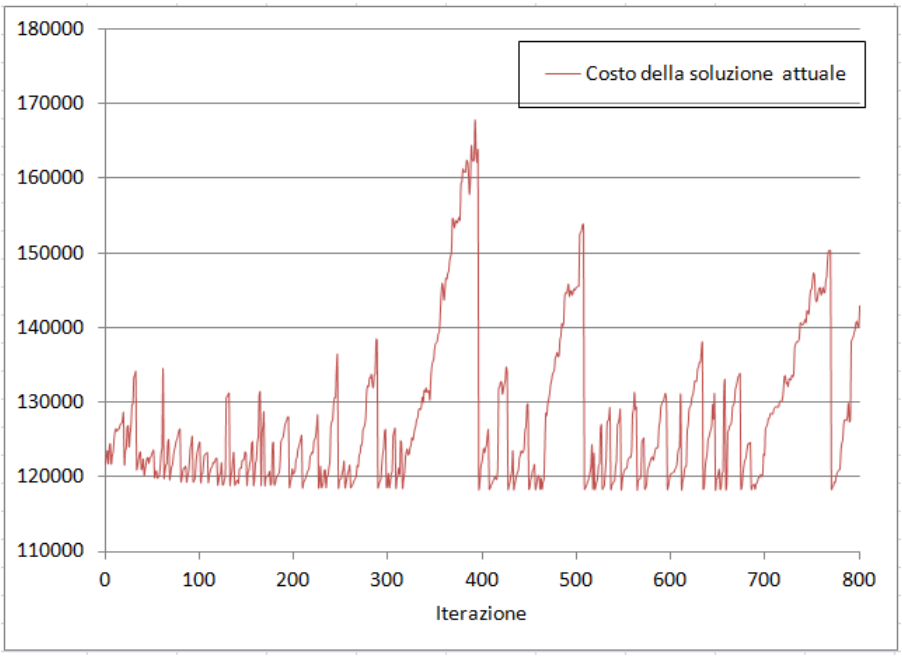


Figura 4.8: Confronto dei vari multistart in base all'algoritmo di costruzione.

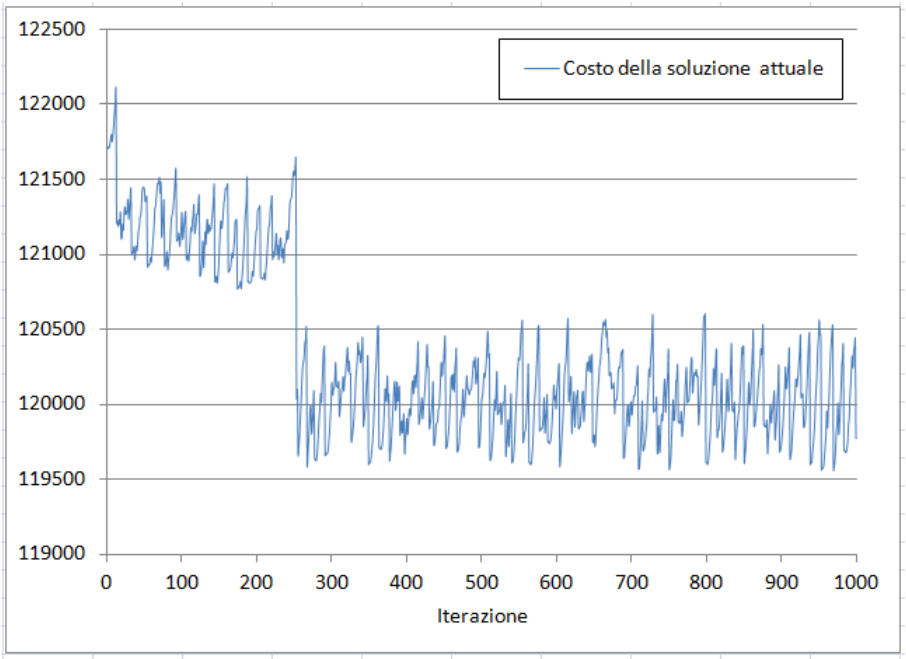


Figura 4.9: Confronto dei vari multistart in base all'algoritmo di costruzione.

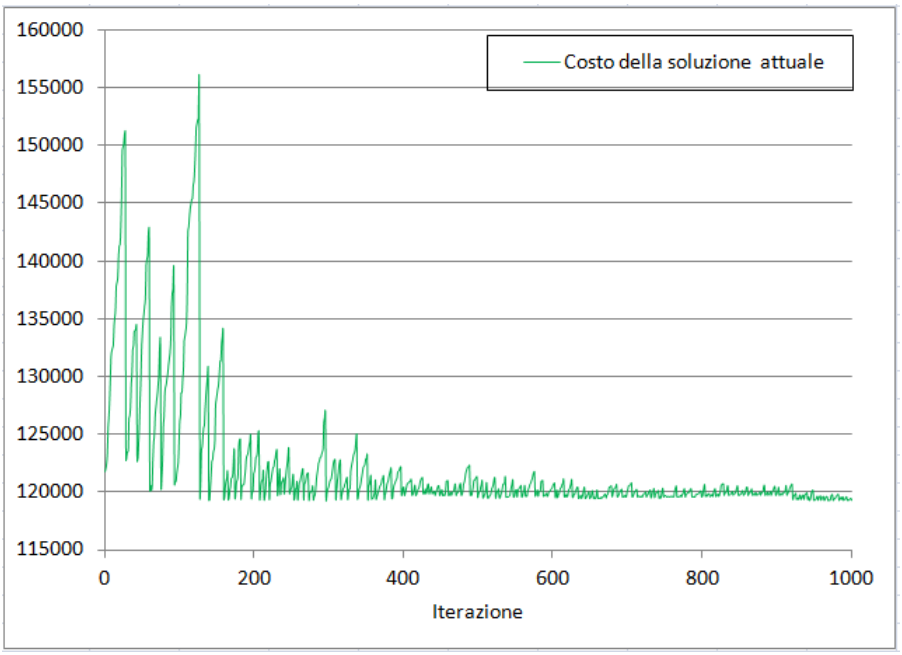


Figura 4.10: Confronto dei vari multistart in base all'algoritmo di costruzione.

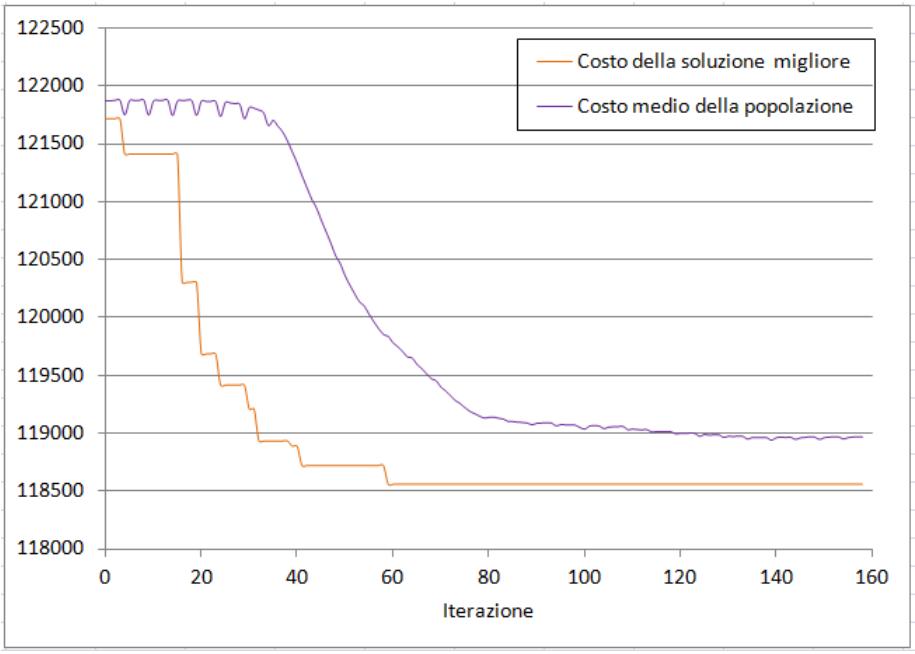


Figura 4.11: Confronto dei vari multistart in base all'algoritmo di costruzione.

Appendice A

TSPlib

Un'istanza di 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 $x_i = x_j \;\; x_i \in \Pi \; \forall \; x_i \in S \; \wedge \; x_i! = x_j \; \forall \; i \neq j$). Poichè in questa variante non esiste alcuna origine, ogni tour può essere descritto da due versi di percorrenza e 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'immagine già esistente.
La vicinanza dei punti generati dipende dalla scala di grigi all'interno dell'immagine (es. generazione di punti a partire dal dipinto della Gioconda[2]).
Le istanze che vengono elaborate dai programmi, creati durante il corso, utilizzano il template **TSPlib**. Di seguito viene riportato il contenuto di un file di questa tipologia.

```
1  NAME : esempio
2  COMMENT : Grafo costituito da 5 nodi
3  TYPE : TSP
4  DIMENSION : 5
5  EDGE_WEIGHT_TYPE : ATT
6  NODE_COORD_SECTION
7  1 6734 1453
8  2 2233 10
9  3 5530 1424
10 4 401 841
11 5 3082 1644
12 EOF
```

Listing A.1: esempio.tsp

Le parole chiave più importanti, contenute in questi file A.1, 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 **EDGE_WEIGHT_SECTION**, in cui sono riportati, all'interno di una matrice, i pesi di tutti gli archi. Questa tipologia di istanze non viene utilizzata dal programma da noi implementato.

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

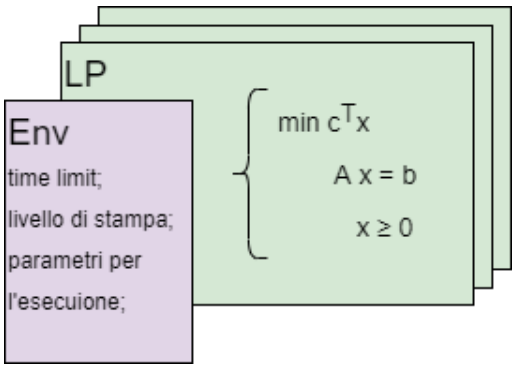


Figura B.1: Strutture CPLEX

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.

i \ j	0	1	2	3	4	5
0		0	1	2	3	4
1			5	6	7	8
2				9	10	11
3					12	13
4						14
5						

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 costruita 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 una colonna alla volta al modello, definendo quindi anche la funzione obiettivo. 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 se si verifica un errore

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 informazioni da inserire, 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 riferimento 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 se si verifica un errore

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 $x_{k,j}$ 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 di cui modificare il coefficiente

newvalue: nuovo valore del coefficiente

Return Value: 0 in caso di successo, un valore diverso da 0 se si verifica un errore

L'utilizzo di questa metodo per inserire nuovi vincoli è però considerato inefficiente. Al suo posto è consigliato l'utilizzo di una funzione che inserisca 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 per definire le righe da aggiungere

rmatind: vettore per definire le righe da aggiungere

rmatval: vettore per definire le righe da aggiungere

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 se si verifica un errore

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 se si verifica un errore

Per poter impostare una variabile $x_{i,j}$ ad una valore fissato è necessario rendere i suoi lower e upper bound alla quantità voluta. Per cambiare questi parametri viene utilizzata 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 indice delle colonne corrispondenti alle variabili di cui cambiare il bound

lu: array di caratteri che specificano il bound da modificare, a scelta tra:
'U' per upper bound
'L' per lower bound
'B' per entrambi

bd: vettore con i nuovi valori

Return Value: 0 in caso di successo, un valore diverso da 0 se si verifica un errore

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 se si verifica un errore

- **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 se si verifica un errore

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.
Per questo motivo **begin = 0** e **end = num_colonne - 1**[1][2].
In seguito il nostro programma analizza la correttezza della soluzione svolgendo la verifica su:

- *valori assunti dalle variabili*
  ciascun $x_{i,j}$ 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 in cui verrà salvato il gap
Return Value:	0 in caso di successo, un valore diverso da 0 se si verifica un errore

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

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

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

B.2.3 Lazy constraints

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 pull. 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:
'L' per vincolo $\leq$
'E' per vincolo $=$
'G' per vincolo $\geq$
'R' per vincolo definito in un intervallo

rmatbeg: vettore con le posizione iniziali dei coefficienti nei vincoli

rmatind: vettore di vettori contenenti gli indici delle variabili appartenenti al vincolo

rmatval: vettore di vettori 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 se si verifica un errore

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 corrispondente coefficiente (in questo caso c_j).

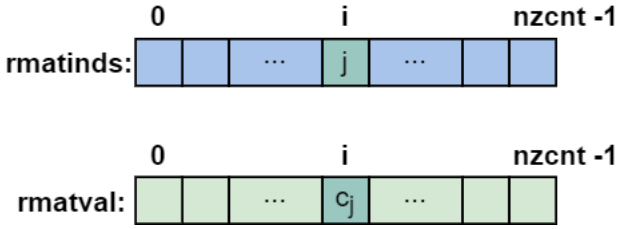


Figura B.3: Array lazy constraints

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 se si verifica un errore

Una volta installata la callback è necessario cambiare l'impostazione del numero di thread utilizzati dal programma. Infatti CPLEX, non sapendo se la funzione implementata dall'utente è thread safe, impedisce lo svolgimento di elaborazioni in parallelo con le callback. A meno che questo non venga esplicitamente dichiarato dall'utente con l'impostazione del corrispondente parametro. Per questo può tornare utile la seguente funzione, che restituisce il numero di core presenti nel computer:

```
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 se si verifica un errore

Come descritto nella sezione dedicata, le callback sono funzioni lasciate appositamente vuote da CPLEX, affinché l'utente possa implementarle in maniera personalizzata. Hanno però 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 per la callback
wherefrom:	contiene dove è stata invocata la callback durante l'ottimizzazione
cbhandle:	puntatore a dati privati dell'utente
useraction_p:	specifica le azioni da eseguire al termine della callback: CPX_CALLBACK_DEFAULT: usa il nodo di CPLEX selezionato CPX_CALLBACK_FAIL: esci dell'ottimizzazione CPX_CALLBACK_SET: usa il nodo selezionato come definito nel valore di ritorno
Return Value:	0 in caso di successo, un valore diverso da 0 se si verifica un errore

Nell'implementarla bisogna fare particolare attenzione a renderla thread safe, se si vuole utilizzarla su più processi in parallelo. Infatti, nel caso in cui il programma lavorasse contemporaneamente con più processori, non si devono verificare interferenze di accesso agli stessi dati da parte di invocazioni diverse della callback. Quest'aspetto è lasciato a completa gestione dell'utente.
Per avere accesso alle variabili utilizzate 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 che contiene specifiche informazioni per la callback
wherefrom:	contiene in che punto dell'ottimizzazione è stata invocata la callback
x:	vettore in cui memorizzare le variabili
begin	indice della prima variabile che si vuole venga restituita
end	indice dell'ultima variabile che si vuole venga restituita
Return Value:	0 in caso di successo, un valore diverso da 0 se si verifica un errore

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

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

env:	puntatore ad una struttura ENV
cbdata:	puntatore che contiene specifiche informazioni per la callback
wherefrom:	contiene in che punto dell'ottimizzazione 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 se si verifica un errore

Macro utili da utilizzare come parametro *whichinfo* possono essere:

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 valore della funzione obiettivo del problema legato al nodo corrente che invoca la callback:

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

env:	puntatore ad una struttura ENV
cbdata:	puntatore che contiene specifiche informazioni per la callback
wherefrom:	contiene in che punto dell'ottimizzazione 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 se si verifica un errore

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 che contiene specifiche informazioni per la callback

wherefrom: contiene in che punto dell'ottimizzazione 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 vincolo $\leq$

'E' per vincolo $=$

'G' per 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 se si verifica un errore

Nella seconda modalità, locale, il taglio aggiunto sarà visibile solo ai nodi discendenti di quello che invoca la callback. Viene implementata con 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 che contiene specifiche informazioni per la callback

wherefrom: contiene in che punto dell'ottimizzazione 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 vincolo $\leq$

'E' per vincolo $=$

'G' per 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 se si verifica un errore

B.2.5 Heuristic Callback

Per poter suggerire a CPLEX una soluzione del problema in esame calcolata dall'utente, è necessario utilizzare un particolare tipo di callback, detta *heuristic callback*. Questa, dopo essere stata installata, verrà invocata ad ogni nodo dell'albero del branch and cut.
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 se si verifica un errore

La callback dell'utente deve avere la dichiarazione specificata di seguito:

```
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 che contiene specifiche informazioni per la callback

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

cbhandle: puntatore a dati privati dell'utente

objval_p: puntatore ad una variabile che in ingresso contiene il valore della funzione obiettivo del problema e in uscita il valore della funzione obiettivo trovata nella funzione stessa, se esiste

x: vettore che in ingresso contiene una soluzione valida per il problema e in uscita i valori della soluzione trovata nella funzione, se presente

checkfeas_p: puntatore che specifica se CPLEX deve verificare la soluzione trovata oppure no

useraction_p: puntatore ad un intero che specifica a CPLEX come proseguire la computazione al termine della callback dell'utente, scelto tra:
CPX_CALLBACK_DEFAULT: nessuna soluzione trovata
CPX_CALLBACK_FAIL: uscire dall'ottimizzazione
CPX_CALLBACK_SET: usare la soluzione fornita dall'utente

Return Value: 0 in caso di successo, un valore diverso da 0 se si verifica un errore

B.2.6 Generic Callback

Per evitare che alcune procedure interne a CPLEX vengano disattivate nel momento dell'installazione di una callback, recentemente ne è stata sviluppata una particolare tipologia, 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:	specifica in quali contesti deve essere invocata la callback, è possibile metterne in or anche più di uno 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 se si verifica un errore

Il parametro *contextmask* può variare a seconda dello scopo della callback creata. Alcuni possibili valori sono:

- **CPX_CALLBACKCONTEXT_CANDIDATE**:
  la callback verrà invocata quando viene trovata da CPLEX una nuova soluzione possibile che l'utente 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).

L'utente può specificare più contesti con una sola installazione, è sufficiente separare le macro desiderate con l'operatore or bitwise ('|').
La user-callback implementata deve avere questa dichiarazione:

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

contex:	puntatore ad una struttura di 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 se si verifica un errore

L'utente può installare una sola user-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( CPXCALLBACKCONTEXTptr context, double *x,
    CPXDIM begin, CPXDIM end, double *obj_p, );
```

contex:	contesto, come passato alla callback scritta dall'utente
x:	vettore dove memorizzare i valori richiesti
begin:	indice prima colonna richiesta
end:	indice dell'ultima colonna richiesta

obi_p: buffer in cui memorizzare il costo della soluzione candidata, può essere NULL

Return Value: 0 in caso di successo, un valore diverso da 0 se si verifica un errore

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 );
```

contex: contesto, come passato alla callback scritta dall'utente

rcnt: numero di vincoli che tagliano la soluzione

nzcnt: numero di coefficienti non nulli nel vincolo

rhs: vettore di termini noti

sense: vettore con la tipologia dei vincoli specificati

rmatbeg: vettore di indici che specifica dove inizia 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 se si verifica un errore

Per suggerire a CPLEX la soluzione da cui proseguire nella computazione dev'essere utilizzata la funzione:

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

contex: contesto, come 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, scelta tra:

CPXCALLBACKSOLUTION_NOCHECK affinchè CPLEX non controlli l'attuabilità della soluzione (che deve essere completa)

CPXCALLBACKSOLUTION_CHECKFEAS affinchè CPLEX controlli solamente se la soluzione è attuabile (la soluzione proposta deve essere completa)

CPXCALLBACKSOLUTION_PROPAGATE affinchè CPLEX

cerchi di completare la soluzione attraverso la propagazione del bound
CPXCALLBACKSOLUTION_SOLVE affinchè CPLEX
fissi le variabili specificate nella soluzione e cerchi di risolvere il risultante
problema ridotto

Return Value: 0 in caso di successo, un valore diverso da 0 se si verifica un errore

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: intero corrispondente al 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 se si verifica un errore

CPX_PARAM_EPGAP	tolleranza dell'intervallo tra la migliore funzione obiettivo intera e la funzione obiettivo del miglior nodo rimanente.
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_POPULATELIM	limita il numero di soluzioni MIP generate per il pool di soluzioni durante ogni chiamata alla procedura populate.
CPX_PARAM_SCRIND	visione o meno dei messaggi di log di CPLEX.
CPX_PARAM_MIPCBREDLP	permette, dalla callback chiamata, di accedere al modello originale del problema e non a quello ridotto .
CPX_PARAM_THREADS	imposta il numero massimo di thread utilizzabili.
CPX_PARAM_RINSHEUR	imposta la frequenza (ogni quanti nodi) con cui deve essere invocato da CPLEX l'algoritmo euristico Rins.
CPX_PARAM_POLISHTIME	imposta quanto tempo in secondi deve dedicare CPLEX a fare il polish della soluzione.

CPX_PARAM_INTSOLLIM	imposta il numero di soluzioni MIP da trovare prima di fermarsi.
CPX_PARAM_TIMELIMIT	imposta il tempo massimo per il calcolo della soluzione.
CPX_PARAM_RANDOMSEED	imposta il random seed.

Tabella B.1: Parametri.

B.4 Costanti utili

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

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

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 sfruttato GNUplot, un programma di tipo command-driven.
Per poterlo utilizzare 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 un file di testo e l'utilizzo di un pipe.

In questa trattazione è stato scelto il secondo metodo. All'interno del file è possibile specificare a Gnuplot le caratteristiche grafiche che deve aver il grafo. Di seguito viene riportato un esempio di tale file.

```
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 sopra riportato, 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 del grafo in una finestra, utilizzando il primo e secondo valore di ciascuna riga del file **solution.dat** come coordinate mentre il terzo valore viene utilizzato come etichetta.

Il file **solution.dat** contiene le informazioni relative alla soluzione del grafico in cui ciascuna riga ha la seguente forma:

```
coordinata_x    coordinata_y    posizione_in_tour
```

coordinata_x rappresenta la coordinata x del nodo;
coordinata_y rappresenta la coordinata y del nodo;
posizione_in_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 all'interno di **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 al passaggio di questo al programma 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 -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[3]. Di seguito vengono riportati i principali argomenti da linea di comando che possono essere utilizzati:

-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= value
-P "title"	title è il titolo del plot
-X value	nome dell'asse x (default ='Time Ratio')
-B	plot in bianco e nero

Di seguito viene riportato un esempio dell'esecuzione del programma, del suo input e del suo output:

- **comando**

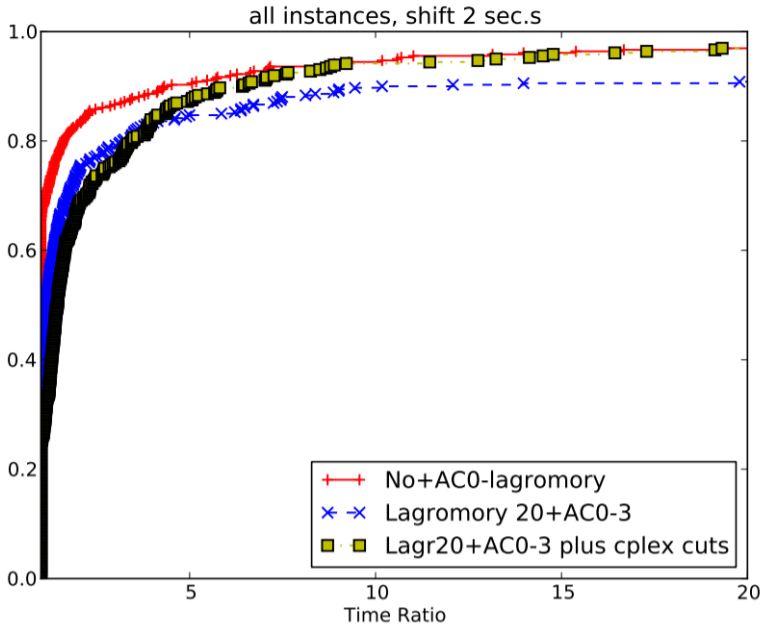
```
python perfprof.py -D , -T 3600 -S 2 -M 20 esempio.csv out.pdf
    -P "all instances, shift 2 sec.s"
```

- **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 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 chiamando il programma.



Bibliografia

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

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

[3] E. D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

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

[5] 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.

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

[7] Anup Dewanji Samrat Hore, Aditya Chatterjee. Improving variable neighborhood search to solve the traveling salesman problem. volume 98, pages 83–91. Elsevier, 2018.

[8] E. Taillard, A. Hertz, and D. de Werra. A tutorial on tabu search.