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

Introduzione

L'intera tesina verterà sul Travelling Salesman Problem. Quest'ultimo si pone l'obiettivo di trovare un tour ottimo,ovvero di costo minimo, all'intero di un grafo orientato.
In questa trattazione verranno analizzate soluzioni algoritmiche per una sua variante, detta simmetrica, che viene applicata a un grafo completo non orientato.
Di seguito viene riportata la formulazione matematica di tale versione:

$$\begin{cases} min \sum_{e \in E} c_e \, x_e & \\ \sum_{e \in \delta(v)} x_e = 2 & \forall \, v \in V \\ \sum_{e \in E(S)} x_e \leq |S| - 1 & \forall \, S \underset{\neq}{\subset} V : |S| \geq 3 \end{cases}$$

Un'istanza di tale problema viene definita normalmente da un grafo, per cui ad ogni nodo viene associata un numero intero (Ex. $\Pi = \{1, 2, 3, .., n\}$).
I risolutori che verranno applicati al problema sono di due tipologie:

- **Risolutori esatti**
  basati sul Branch & Bound. I più conosciuti sono:
  - **IBM ILOG CPLEX**
    gratuito se utilizzato solo a livello accademico.
  - **XPRESS**
  - **Gurobi**
  - **CBC**
    l'unico Open-Source tra questi

- **Risolutori euristici (meta-euristici)**
  algoritmi che forniscono una soluzione approssimata.

Esempio di risolutori: Concorde [1]
William Cook [2]

Istanze e soluzioni del problema

2.1 Istanze

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[3]).
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.

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

Listing 2.1: esempio.tsp

Le parole chiave più importanti, contenute in questi file 2.1, 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

2.2 Soluzioni

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 $x_i = x_j \quad x_i \in \Pi \; \forall \; x_i \in S \; \wedge \; x_i! = x_j \; \forall \; i \neq j$). 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 Sezione 5.2.

Risoluzione del problema

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 3.1):

- **ENV (enviroment):** 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

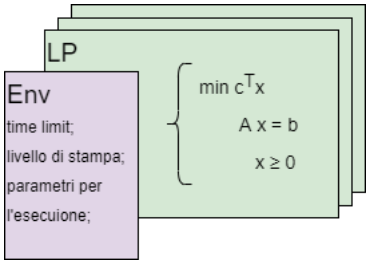


Figura 3.1: Strutture CPLEX

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 3.2 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.
Il modello così strutturato richiede però l'inserimento di un esponenziale numero di vincoli per l'eliminazione dei sub-tour, vengono quindi ora descritti altri modelli che ovvino a questo problema.

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 3.2: Indici della matrice

3.1 Modelli compatti

I modelli compatti del Travelling Salesman Problem, sono formulazioni il cui numero di variabili e di vincoli è polinomiale nella taglia dell'istanza. In particolare, in quelle analizzate in seguito, sono entrambi $O(n^2)$, con $n =$ *numero di nodi*.
I modelli compatti sono però applicabili solo a grafi orientati. Per poterli sfruttare per la risoluzione del TSP simmetrico, è necessario per ogni ramo dell'istanza (i, j), inserire nel modello i corrispondenti rami orientati in entrambe le direzioni (i, j) e (j, i). Questo comporta un significativo rallentamento nella computazione della soluzione, in quanto l'algoritmo, ogni volta che scarta un ramo (i, j) dalla soluzione ottima, verifica se il corrispondente

(j, i) potrebbe invece appartenerle. Questo non può però essere possibile, essendo i due rami in realtà lo stesso nella nostra istanza iniziale.

3.1.1 Formulazione sequenziale

Miller, Tucker e Zemlin, nella loro formulazione del modello, hanno introdotto una nuova variabile u_i per ogni nodo i e imposto che, nella soluzione ottima, il suo valore rispettasse dei nuovi vincoli. Questi servivano a garantire che venisse seguito un ordine di percorrenza di tutti i nodi. In questo modo hanno eliminato la creazione di sub-tour, mantenendo il numero di vincoli e di variabili polinomiale. Nello specifico il loro modello è così strutturato:

$$min \sum_{i \in V} \sum_{j \in V} c_{i,j}\ x_{i,j} \tag{3.1}$$

$$\sum_{i \in V} x_{ih} \ = \ 1 \qquad \forall\, h \in V \tag{3.2}$$

$$\sum_{j \in V} x_{hj} \ = \ 1 \qquad \forall\, h \in V \tag{3.3}$$

$$u_i - u_j + n\ x_{i,j} \ \leq \ n - 1 \qquad \forall\, i, j \in V - \{1\}, i \neq j \tag{3.4}$$

$$0 \leq u_i \ \leq \ n - 2 \qquad \forall\, i \in V - \{1\} \tag{3.5}$$

Esistono due diversi modi per implementare questo modello sfruttando le funzioni di CPLEX.
Nel primo i nuovi vincoli vengono aggiunti come visto in precedenza. In questo modo, durante la fase di preprocessamento, il programma è già a conoscenza di tutti i vincoli che dovrà rispettare la soluzione ottima. Ciò gli permette di migliorare i coefficienti presenti, prima ancora di iniziare la computazione dell'ottimo.
Il secondo metodo, invece, sfrutta l'inserimento nel modello di vincoli detti "lazy constraints". Questi non sono noti al programma dall'inizio, ma vengono inseriti all'interno di un pool di vincoli. Nel momento in cui viene calcolata una soluzione, CPLEX verifica che vengano rispettati tutti i vincoli presenti nel pool. Se ne trova uno violato lo aggiunge al modello e ripete la

computazione. Questo approccio permette, per risolvere lo stesso problema, di eseguire calcoli su un modello più piccolo, ma può aumentare i tempi di computazione non fornendo a CPLEX tutte le informazioni dall'inizio.

3.1.2 Formulazione basata sul flusso

Nella formulazione di Gavish e Graves, per impedire la formazione di subtour all'interno della soluzione ottima, viene introdotto un nuovo vincolo per ogni ramo del grafo. Questo permette di regolare il flusso $y_{i,j}$, con $i \neq j$, che lo attraversa.Inoltre è stato necessario aggiungere anche dei vincoli, detti "vincoli di accoppiamento", che collegassero i flussi alle variabili $x_{i,j}$. Il loro modello è quindi così strutturato:

$$min \sum_{i \in V} \sum_{j \in V} c_{i,j}\ x_{i,j} \tag{3.6}$$

$$\sum_{i \in V} x_{ih} \ = \ 1 \qquad \forall\, h \in V \tag{3.7}$$

$$\sum_{j \in V} x_{hj} \ = \ 1 \qquad \forall\, h \in V \tag{3.8}$$

$$\sum_{j \in V} y_{1,j} \ = \ 1 \tag{3.9}$$

$$\sum_{j \in V} y_{h,j} \ = \ \sum_{i \in V} y_{i,h} \ - 1 \qquad \forall\, h \in V - \{1\} \tag{3.10}$$

$$y_{i,j} \leq \ (n-1)\ x_{i,j} \qquad \forall\, i,j \in V, i \neq j \tag{3.11}$$

La soluzione di questo modello risulta però essere lontana dalla convex hull. Per migliorarla è possibile sostituire il vincolo *(3.11)* con

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

mentre per gli altri valori di i e j è necessario lasciare i vincoli originali. Per evitare che la soluzione ottima contenga sia l'arco $x_{i,j}$ che $x_{j,i}$, che nella

nostra istanza iniziale corrispondono allo stesso arco, viene anche aggiunto il seguente vincolo:

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

3.2 Loop

3.2.1 Formulazione di Benders

Negli anni '60, Jacques F. Benders sviluppò un approccio generale, applicabile a qualsiasi problema di programmazione lineare, per ridurre il numero esponenziale di alcuni vincoli specifici inseriti nel modello.
Utilizzando questo metodo, il modello viene scritto senza quei vincoli e poi questi verranno aggiunti in seguito durante la risoluzione del problema. Nel caso in cui la soluzione ottima, calcolata a partire da questo modello, non rispetti un vincolo di quelli rimossi, questo viene aggiunto al modello.
Nella seguente parte, viene riportata l'applicazione specifica di loop al problema TSP. I vincoli di Subtour Elimination sono in numero esponenziale e sono:

$$\sum_{e \in E(S)} x_e \leq \mid S \mid -1 \; \forall \; S \underset{\neq}{\subset} V \; : \mid S \mid \geq 2 \tag{3.12}$$

o equivalentemente:

$$\sum_{e \in \delta(S)} x_e \geq 2 \; \forall \; S \underset{\neq}{\subset} V \; : \mid S \mid \geq 2 \tag{3.13}$$

Viene definito un nuovo modello per il problema del commesso viaggiatore simmetrico, in cui vengono rimossi tali vincoli, aggiungendo così la possibilità di avere dei subtour nella soluzione finale.
Viene risolto il problema e nel caso in cui ci sia più di una componente connessa, viene aggiunto al modello un vincolo di subtour elimination per ogni ciclo generato.

All'aumentare del numero di vincoli, il costo della soluzione ottenuta da

Algorithm 1 Risoluzione del problema

Require: $MODEL$ = Modello TSP simmetrico senza vincoli di Subtour elimination
Ensure: x: soluzione intera senza subtour

$x \leftarrow solve(MODEL)$
$num\_comps \leftarrow comps(x)$

while $num\_comps \geq 2$ **do**
  Add $\sum_{e \in \delta(S_k)} x_e \leq \mid S_k \mid -1 \; \forall$ *componente connessa* S_k
  if $num\_comps \geq 2$ **then**
    $x \leftarrow solve(MODEL)$
    $num\_comps \leftarrow comps(x)$
  end if

end while

CPLEX peggiora o resta identica a quella elaborata all'iterazione precedente del metodo loop.
Il numero di iterazioni che vengono effettuate dall'algoritmo non è conosciuto e potrebbe essere anche molto elevato. Nel caso peggiore vengono inseriti tutti i vincoli di Subtour elimination, ovvero un numero esponenziale di disequazioni, soprattutto con istanze clusterizzate.
Inoltre il problema principale di questo algoritmo è la generazione, ad ogni iterazione, di un albero completo di branching, eliminando quello precedentemente sviluppato.
In passato, con le versioni del MIP solver di CPLEX degli anni '60, 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.
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 cominciare la risoluzione del problema. Nella fase di pre-processing infatti, vengono applicati algoritmi euristici e cambiamenti dei coefficienti nel modello, in base ai vincoli inseriti.
L'algoritmo può essere modificato svolgendo prima il metodo loop con l'aggiunta di parametri differenti da quelli utilizzati di default del risolutore

CPLEX. In seguito viene effettuato nuovamente l'algoritmo di Benders ma questa volta nella sua versione esatta, in modo da migliorare la soluzione meta-euristica trovata nella prima parte.
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, alcune delle soluzioni metaeuristiche ottenute nella prima fase vengono sfruttate come bound nella seconda.

3.2.2 Formulazione con callback

Un possibile miglioramento dell'algoritmo proposto da Benders, in termini di velocità della computazione, è il seguente.
Come precedentemente descritto, come prima cosa CPLEX effettua un preprocessamento in cui semplifica il modello, riducendo il termine noto e accorpando tra loro diverse variabili. 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 3.3).

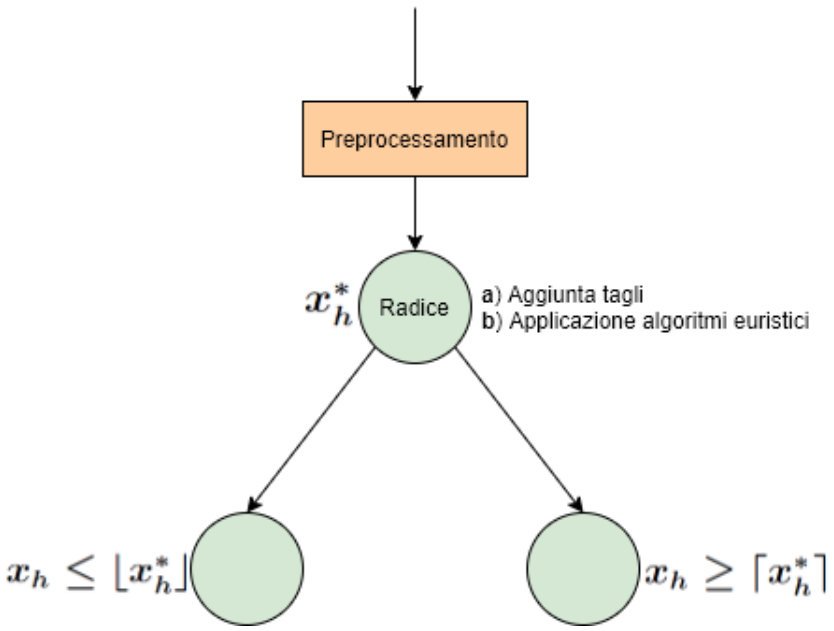


Figura 3.3: Albero decisionale del Branch and Cut

Nello sviluppo di ogni ramo l'upper bound sarà dato dagli algoritmi euristici utilizzati, mentre il lower bound dal rilassamento del problema.
Per poter velocizzare l'approccio proposto da Benders, è possibile personalizzare questa fase e scegliere quali tagli far applicare a CPLEX. Nel nostro caso, questi vengono utilizzati per eliminare l'eventuale presenza di subtour nella soluzione calcolata. Per fare ciò vengono sfruttate particolari funzioni fornite da CPLEX, dette *callback*. Queste sono state lasciate volutamente vuote dai creatori della libreria, affinché l'utente possa implementarne all'interno il suo specifico codice. In particolare, le funzioni utilizzate sono callback necessarie ad aggiungere lazy constraints al modello e per questo dette *lazy constraints callback*. La callback implementata viene chiamata solo al momento di aggiornare 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 constraints*, 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 proseguirne lo sviluppo aggiungendo nuovi rami. In particolari casi, però, CPLEX può ritenere più conveniente distruggere tutto l'albero decisionale fin'ora calcolato e ricominciare dalla radice. Questo può avvenire in qualunque punto dell'elaborazione della soluzione ottima.
Attraverso l'utilizzo delle callbacks è possibile accedere a molti dati interni all'elaborazione di CPLEX. Particolari procedure vengono quindi automaticamente disattivate, affinché l'utente non possa venirne a conoscenza. Per evitare questo è possibile installare le callbacks con una modalità leggermente diversa, attraverso funzioni dette *general*.

3.3 Algoritmi Euristici

Gli algoritmi euristici sono progettati per risolvere istanze del problema in tempi significativamente più brevi rispetto agli algoritmi esatti. Di conseguenza, però, al termine della computazione non garantiscono di ottenere una soluzione ottima, ma solo una sua buona approssimazione ammissibile.
Durante la computazione della soluzione CPLEX utilizza diversi algoritmi euristici, grazie alla variazione di alcuni parametri a loro associati è possibile variare la frequenza o il tempo a loro dedicato.

3.3.1 Hard Fixing

Un primo algoritmo euristico di semplice implementazione si basa sull'impostazione di una deadline da parte dell'utente ed è composto dalle seguenti fasi:

1. Impostazione di un time limit per la computazione della soluzione;
2. Calcolo della soluzione;
3. Selezione, in maniera randomica, di un sottoinsieme di rami appartenenti alla soluzione ottima (Figura 3.4). Il numero di questi sarà dato da una percentuale fissata del totale. I rami appartenenti alla selezione vengono fissati di modo tale che, in una successiva computazione del problema, appartengano alla soluzione restituita;

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.
Il time limit nominato nella **fase 1** è dato da una frazione della deadline complessiva e dipende dal numero di iterazioni totali che si desidera compiere. Ad ogni computazione la soluzione potrà essere solo migliore o uguale alla precedente (nel caso peggiore).
La percentuale scelta del numero di rami da fissare può variare ad ogni iterazione. Generalmente si cerca di avere una percentuale alta nelle prime iterazione, in cui la soluzione non è ancora stata raffinata, e di abbassarla man mano che si procede con l'algoritmo. In questo modo nelle ultime computazione CPLEX avrà un maggior numero di gradi di libertà per trovare la soluzione che più si avvicina all'ottimo. Poichè i rami selezionati nella **fase 3** sono scelti in maniera casuale, non si corre il rischio di entrare in un ciclo infinito, in cui viene risolto ogni volta la stessa istanza con le stesse variabili fissate. Particolare attenzione deve essere posta al fatto di lasciare nell'insieme dei rami scelti randomicamente solo quelli selezionate all'iterazione immediatamente precedente.
Nella nostra implementazione abbiamo scelto di iterare l'algoritmo per 6 volte, ognuna con una deadline di 10 minuti, e di utilizzare rispettivamente come percentuale i seguenti valori { 90, 75, 50, 25, 10, 0 }.
Di seguito viene inoltre riportato lo pseudocodice dell'algoritmo appena descritto.

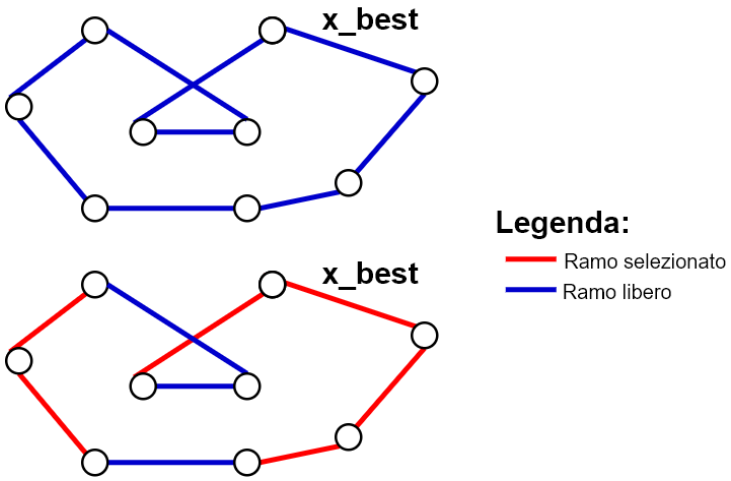


Figura 3.4: Selezione rami

Algorithm 2 Hard Fixing

Require: $modello$, $deadline$, $percentuale$
  $n = 0$
  while $(n < num\_iterazioni)$ **do**
    $set\_time\_limit(deadline/num\_iterazioni)$
    $solve(modello)$
    $x\_best \leftarrow soluzione\ \ calcolata$
    for $j \leftarrow\ 0, ..., num\_nodi - 1$ **do**
      $k = random(0, 1)$
      if $(100k \leq percentuale)$ **then**
        $Add\ \ x\_best[j]\ \ to\ \ S$
      end if
    end for
    for all $x_{i,j} \in S$ **do**
      $x_{i,j} = 1$
    end for
    $n++$
  end while

3.3.2 Soft Fixing

Il metodo seguente fa utilizzo di vincoli aggiuntivi, detti **Local Branching** e che ha dato il via alla sviluppo della **Math-Heuristic**, approccio di programmazione matematica (es. tramite CPLEX) unita all'algoritmica euristica[5]. 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 sui lati 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 che vengono preservate a 1 rispetto alla soluzione x^H e **n** indica il numero di archi selezionati, pari al numero di nodi + 1.
In questo caso, il vincolo permetterà a CPLEX di fissare il 90% dei rami scelti in x^H e avere il 10% di libertà. Per questo motivo, CPLEX riduce il numero di archi su cui lavorare, in quanto molti nodi hanno già un numero di archi selezionati pari a 2.
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 raggiungere la nuova soluzione.
Ad ogni iterazione viene aggiunto un nuovo local branching, basato sull'attuale soluzione restituita da CPLEX, e rimosso il vincolo aggiunto nell'iterazione precedente.
L'unico aspetto negativo di questo metodo riguarda un mancato miglioramento della soluzione da parte di CPLEX in seguito alla sua esecuzione con l'aggiunta del vincolo. Non scegliendo in maniera randomico i lati da selezionare della soluzione precedente, 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 di prima. Per superare tale problema, 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, vengono enumerati gli elementi di questo spazio, generando un numero molto elevatp di possibli soluzioni, considerando che TSP è un problema NP-hard.
Definita una soluzione intera ammissibile x^H e utilizzando la distanza di Hamming, vengono definite soluzioni $k - opt$ rispetto ad x^H, quelle che hanno distanza k da x^H (vedi Figura 3.5).

Se, invece del local branching, venisse utilizzata l'enumerazione delle solu-

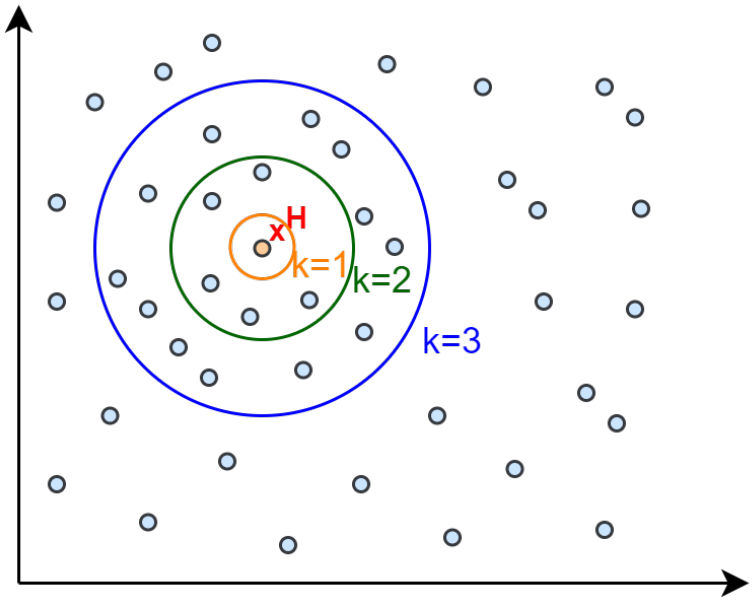


Figura 3.5: Spazio delle soluzioni e distanza di Hamming.

zioni, si dovrebbero generare per **k** generico circa n^k soluzioni a distanza k da x^H ed in seguito analizzarle tutte per trovare quella con costo minore e migliore 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 adottare per generare tutte le soluzioni a distanza minore o uguale di R dalla soluzione euristica di partenza x_H:

$$min\{c^T x : \quad Ax = b,\ x \in \{0,1\}^n\} \tag{3.14}$$

$$\sum_{j \in E: {x_j}^H = 0} x_j \ + \sum_{j \in E: {x_j}^H = 1} 1 - x_j = \leq R \tag{3.15}$$

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 3.6** seguito viene riportato un esmpio di una possibile evoluzione dell'algoritmo nella ricerca dell'ottimo, evindenziandone le soluzioni trovate di volta in volta.

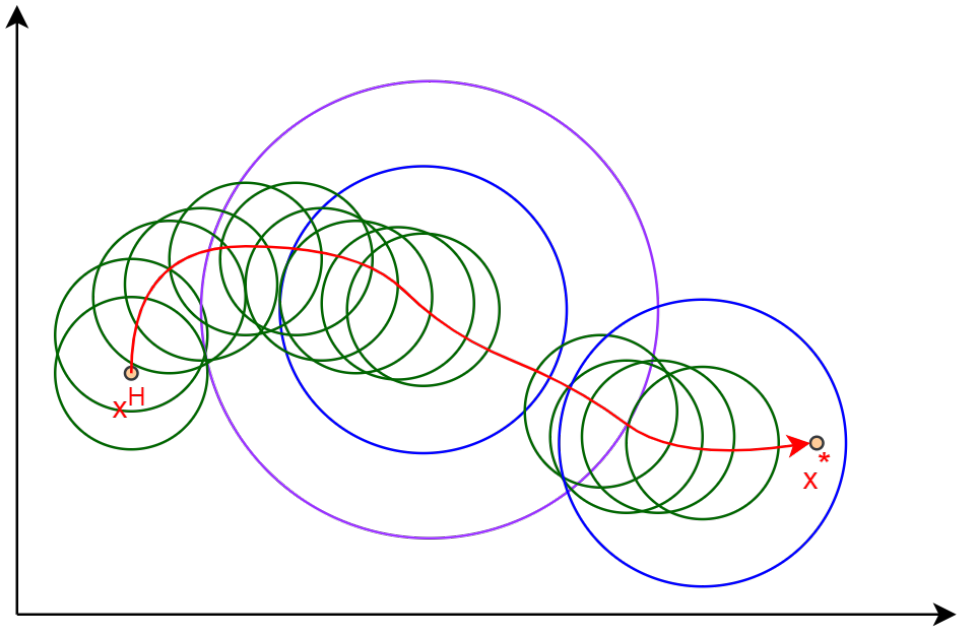


Figura 3.6: Esempio di esecuzione dell'algoritmo nello spazio delle soluzioni.

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 5.1.2).
  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 5.1.2).

- **Node limit**
  massimo numero di nodi risolti prima che l'algoritmo termini senza raggiungere l'ottimalità (vedi Sezione 5.1.2).

- **Populate limit**
  massimo numero di soluzioni MIP generate per il pool di soluzioni durante ogni chiamata della procedura di popolazione (vedi Sezione 5.1.2).

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 4.1).
L'andamento del performance profile di un algoritmo è monotono crescente. Il valore assunto per ogni ratio dagli algoritmi all'interno del grafico è la percentuale del numero di istanze che l'algoritmo risolve con quel fattore rispetto all'ottimo di quel caso.
Spesso questi grafici vengono rappresentati in scala logaritmica per notare al meglio le differenze ed avere una migliore raffigurazione, più semplice da essere analizzata visivamente.

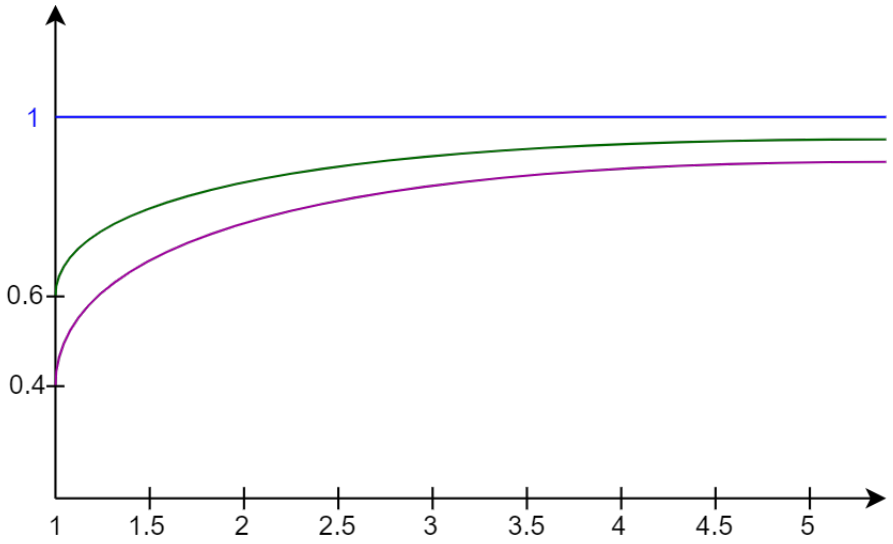


Figura 4.1: Performance profile di due algoritmi.

Per creare il performance profile degli algoritmi implementati, è stato utilizzato il programma python riportato nella Sezione 5.3.

4.4 Analisi degli algoritmi sviluppati

Appendice

In questa sezione verranno approfondite alcune funzioni di CPLEX necessarie ad implementare gli algoritmi descritti nei capitoli precedenti. Inoltre vengono analizzati tutti gli altri programmi, utilizzati nella stampa delle soluzioni e nell'analisi delle performance.

5.1 CPLEX

5.1.1 Funzioni

5.1.1.1 Costruzione 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 5.1: 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 creato. 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(istanza_problema, env , lv);
```

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

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr 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:

```
CPXnewcols(env, lp, num_colonne, costi, lower_bound,
           upper_bound, tipi_variabili, nomi_variabili);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: di tipo CPXLPptr, è un puntatore alla struttura LP precedentemente creata

num_colonne: numero di colonne da inserire

costi: vettore dei costi relativi agli archi da inserire

lower_bound: vettore contenente i lower bound dei valori assumibili dalle variabili da inserire

upper_bound: vettore contenente gli upper bound dei valori assumibili dalle variabili da inserire

tipi_variabili: vettore contenente la tipologia delle variabili da inserire

nomi_variabili: vettore di stringhe contenenti i nomi delle variabili da inserire

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:

```
CPXnewrows(env, lp, numero_righe,termini_noti,
           tipi_vincoli, range_valori, nomi_vincoli);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr alla struttura LP precedentemente creata

numero_righe: numero di righe (vincoli) da inserire

termini_noti: vettore dei termini noti dei vincoli

tipi_vincoli: 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

range_valori: vettore di range per i valori di ogni vincolo (nel nostro caso è NULL)

nomi_vincoli vettore di stringhe contenenti i nomi

delle variabili da inserire

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 $k = i$ $j = i$ 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:

```
CPXchgcoef(env, lp, i, j, new_value);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr 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

new_value: nuovo valore del coefficiente

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:

```
CPXaddrows(env, lp, num_nc, num_nr, nnz, const_term, type_costr,
           rmatbeg, rmatind, rmatval, col_name, row_name);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr alla struttura LP precedentemente creata

num_nc: numero di nuove colonne che devono essere aggiunte

num_nr: numero di nuove righe che devono essere aggiunte

nnz: numero di coefficienti non nulli nel vincolo aggiunto

const_term: vettore con i termini noti per ogni vincolo da aggiungere

type_costr: 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

col_name: vettore contenente i nomi delle nuove colonne

row_name: vettore contenente i nomi dei nuovi vincoli

5.1.1.2 Calcolo della soluzione

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

- **Risoluzione del problema di ottimizzazione**

```
CPXmipopt(env, lp);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr alla struttura LP precedentemente creata

- **Ottenimento della soluzione**

```
CPXgetx(env, lp, x, inizio, fine);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr alla struttura LP precedentemente creata

x: puntatore a un vettore di double in cui verranno salvati i valori delle variabili ottenuti dalla soluzione ottima

inizio: primo indice della variabile di cui si vuole memorizzare ed analizzare il valore

fine: indice dell'ultima variabile di cui si vuole memorizzare ed analizzare il valore

Questa funzione salva in x tutte le variabili che hanno indice $i \in [inizio, fine]$ e quindi x deve essere un vettore di almeno $fine - inizio + 1$ valori. Nel nostro programma, vengono analizzati i valori di tutte le variabili in gioco.
Per questo motivo **inizio=0** e **fine=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 toccano lo stesso nodo

5.1.1.3 Lazy constraints

Nel caso in cui si voglia sfruttare la possibilità di verificare se è stato rispettato un vincolo, solo al termine della computazione della soluzione, è neces-

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

sario inserire un "lazy constraint". Per fare ciò viene utilizzata la seguente funzione:

```
CPXaddlazyconstraints(env, lp, num_vincoli, nnz,
          termine_costante, tipo_vincolo, posizione_iniziale,
          indici, valori, nome_vincolo);
```

env: puntatore di tipo CPXENVptr alla struttura ENV precedentemente creata

lp: puntatore di tipo CPXLPptr alla struttura LP precedentemente creata

num_vincoli: numero di vincoli da inserire

nnz: vettore con il numeri di variabili per ogni vincolo

termine_costante: vettore dei termini noti dei vincoli

tipi_vincoli: 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

posizione_iniziale: vettore con le posizione iniziali dei coefficienti nei vincoli

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

valori: vettore di vettori con i coefficienti delle variabili del vincolo

nome_vincolo: vettore con i nomi dei vincoli

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 *indici* e *valori*. Come rappresentato in Figura 5.1, 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 $indici[i] = j$). Mentre l'i-esima posizione del vettore di valori contiene il corrispondente coefficiente (in questo caso c_j).

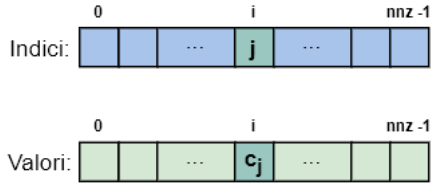


Figura 5.1: Array lazy constraints

5.1.1.4 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:

```
CPXsetlazyconstraintcallbackfunc(env, mylazycallback, tsp_in);
```

env: puntatore di tipo CPXENVptr alla struttura ENV

mylazycallback: puntatore di tipo CPXPUBLIC * alla callback chiamata

tsp_in: puntatore di tipo void* ad una struttura dati contenente le informazioni da passare alla callback

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:

```
CPXgetnumcores(env, ncores);
```

env: puntatore di tipo CPXENVptr ad una struttura ENV

ncores: puntatore alla variabile in cui viene scritto il numero di core

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 di tipo CPXENVptr ad 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

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:

```
CPXgetcallbacknodex(env, cbdata, wherefrom, x_star, start, end);
```

env: puntatore di tipo CPXENVptr 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

x_star: 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

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

```
CPXgetcallbackinfo(env, cbdata, wherefrom, which_info, result);
```

env: puntatore di tipo CPXENVptr 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

which_info: macro che specifica l'informazione che si desidera conoscere

result: puntatore di tipo void in cui verrà memorizzata l'informazione richiesta

Macro utili da utilizzare come parametro *which_info* 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:

```
CPXgetcallbacknodeobjval(env, cbdata, wherefrom, objval);
```

env: puntatore di tipo CPXENVptr 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: puntatore ad una variabile *double* in cui memorizzare il costo

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:

```
CPXcutcallbackadd(env, cbdata, wherefrom, nnz, const_term,
                  type_constraint, indices, values,
                  purgeable);
```

env: puntatore di tipo CPXENVptr 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

nnz: numero di coefficienti non nulli

const_term: valore del termine noto

type_constraint: tipologia del taglio da aggiungere, a scelta tra
'L' per vincolo $\leq$
'E' per vincolo $=$
'G' per vincolo $\geq$

indices: vettore contente gli indici dei coefficienti del taglio

values: vettore contenente i coefficienti delle variabili nel

taglio

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

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:

```
CPXcutcallbackaddlocal(env, cbdata, wherefrom, nnz, const_term,
                       type_constraint, indices, values);
```

env: puntatore di tipo CPXENVptr 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

nnz: numero di coefficienti non nulli

const_term: valore del termine noto

type_constraint: tipologia del taglio da aggiungere, a scelta tra
'L' per vincolo $\leq$
'E' per vincolo $=$
'G' per vincolo $\geq$

indices: vettore contente gli indici dei coefficienti del taglio

values: vettore contenente i coefficienti delle variabili nel taglio

5.1.1.5 Lazy Constraint Callback General

Per evitare che alcune procedure interne a CPLEX vengano disattivate non momento dell'istallazione di una callback, è possibile utilizzarla con una diversa modalità rispetto a quella descritta nella sezione precedente, detta *general*. Per quest'operazione viene invocata la seguente funzione, che si occupa anche di specificare il contesto in cui invocare la callback:

```
CPXcallbacksetfunc (env, lp, contextmask, callback, userhandle);
```

env: puntatore di tipo CPXENVptr ad una struttura ENV

lp: puntatore di tipo CPXLPptr 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

In questo caso come **contextmask** è necessario passare la macro **CPX_CALLBACKCONTEXT_CANDIDATE**, che specifica di invocare la callback nel momento in cui viene trovata una nuova soluzione intera possibile.
La callback deve avere questa dichiarazione:

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

in cui

contex: puntatore ad una struttura di contesto della callback

contextid: intero che specifica il contesto in cui invocare la callback

cbhandle: argomento passato alla callback nell'istallazione

Al suo interno, per poter accedere alla soluzione candidata e al suo costo, deve essere presente la seguente chiamata, che è specifica per questo contesto:

```
CPXcallbackgetcandidatepoint(context, x_star, begin, end, obj_p);
```

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

x_star: 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

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**.

```
CPXcallbackrejectcandidate(context, rcnt, nnz, const_term,
                           type_constraint, rmatbeg, rmatind,
                           rmatval);
```

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

rcnt: numero di vincoli che tagliano la soluzione

nnz: numero di coefficienti non nulli nel vincolo

const_term: vettore di termini noti

type_constraint: 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

5.1.1.6 Algoritmi Euristici

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:

```
CPXchgbds(env, lp, num_bouds, indices, which_bound, values);
```

env: puntatore di tipo CPXENVptr alla struttura ENV

lp: puntatore di tipo CPXLPptr alla struttura LP

num_bounds: numero totale di bound da cambiare

indices: vettore con gli indice delle colonne corrispondenti alle variabili di cui cambiare il bound

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

values: vettore con i nuovi valori

5.1.2 Parametri

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

```
CPXsetintparam(env, numero_parametro, nuovo_valore);
```

mentre se di tipo DOUBLE:

```
CPXsetdblparam(env, numero_parametro, nuovo_valore);
```

In entrambe le funzioni:

5.1.3 Costanti utili

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

env: puntatore di tipo CPXENVptr alla struttura ENV di cui si vogliono cambiare i parametri

numero_parametro: intero corrispondente al parametro da modificare (vedi Tabella 5.1)

nuovo_valore: nuovo valore (rispettivamente intero o double) del parametro

5.2 Gnuplot

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

- Collegare la libreria ed invocare le sue funzioni all'interno del nostro 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 5.2: style.txt

CPX_PARAM_EPGAP	tolleranza dell'intervalo 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.

Tabella 5.1: Parametri.

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;
posizione_in_tour 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

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

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);
```

5.3 perprof.py

Il programma utilizzato per la creazione del performance profile dei diversi algoritmo è perprof.py[4]. Di seuito vengono riportati i principali argomenti da linea di comando che possono essere utilizzati:

-D delimiter	spefica 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"
```

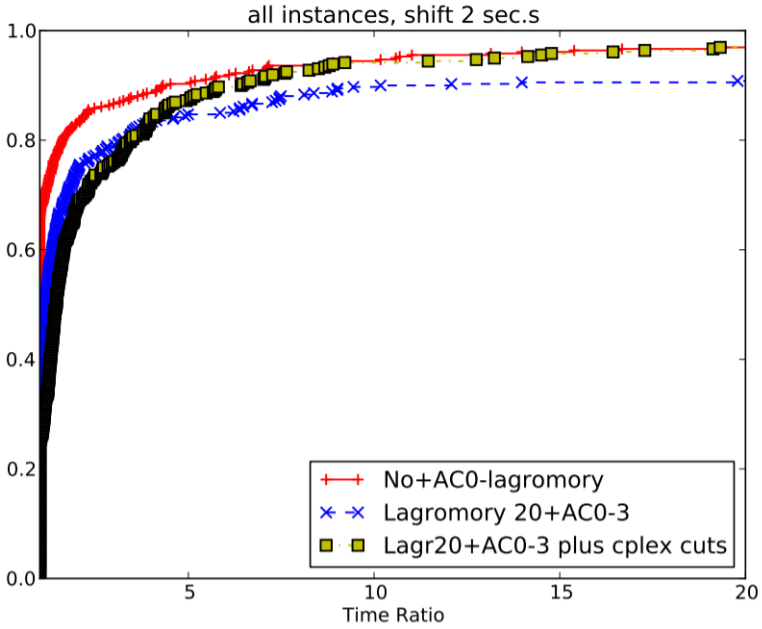
- **file di input con i dati**
  Viene riportato parte del contenuto di esempio.csv .

```
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 line di comando chiamando il programma.



Bibliografia

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

[2] http://www.math.uwaterloo.ca/tsp/.

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

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

[5] Lodi Fischetti, M. A. local branching. math. program. *Ser. B*, 98:23–47, 2003.