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 [2]
William Cook [1]

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.

2.2.1 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.

```
1  set style line 1 \
2      linecolor rgb '#0000ff' \
3      linetype 1 linewidth 1 \
4      pointtype 7 pointsize 0.5
5
6  plot 'solution.dat' with linespoints linestyle 1 title "Solution"
7       ,'' using 1:2:(sprintf("%d", $3)) notitle with labels center
8       offset 1
9
10 set term png
11 set output "solution.png"
12 replot
```

Listing 2.2: 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;
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 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);
```

CPLEX

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

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 nu-

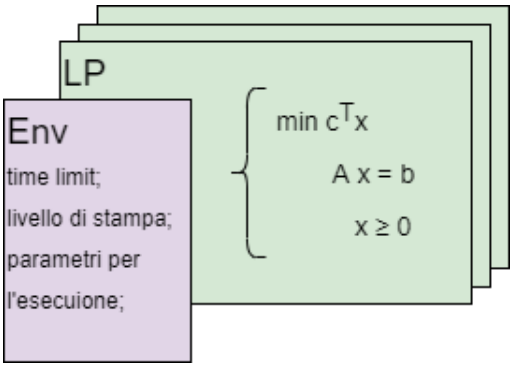

Figura 3.1: Strutture CPLEX

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

mero di vincoli per l'eliminazione dei sub-tour, vengono quindi ora descritti altri modelli che ovvino a questo problema.

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, quindi 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 rallen-

tamento 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 di Miller, Tucker e Zemlin

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 dei nodi che dal primo passasse per tutti quelli presenti nel grafo. 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 all'inizio della costruzione del modello. 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 rispetti 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 problemi, 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 di Gavish e Graves

Nella formulazione di Gavish e Graves, per impedire la formazione di subtour all'interno della soluzione ottima, è stato introdotto un nuovo vincolo per ogni ramo. 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 l'arco $x_{j,i}$, che nella nostra istanza iniziale corrispondono allo stesso arco, viene anche aggiunto il seguente vincolo:

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

Appendice

In questa sezione verranno approfondite alcune funzioni di CPLEX necessarie alle implementazioni descritte nei capitoli precedenti.

4.1 Funzioni

4.1.1 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 4.1: 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);
```

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:

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

```
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.

4.1.2 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 necessario 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 4.1, 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).

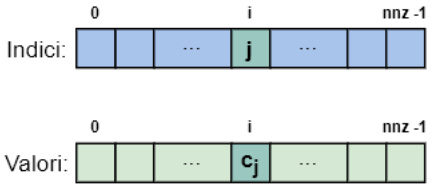

Figura 4.1: Array lazy constraints

4.1.3 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);
```

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

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

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

4.2 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:

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

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

In entrambe le funzioni

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

numero_parametro: intero corrispondente al parametro da modificare

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

4.3 Costanti utili

CPX_ON	1
CPX_OFF	0
CPX_INFBOUND	∞

Bibliografia

[1] *http://www.math.uwaterloo.ca/tsp/*

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

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