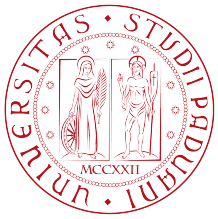


Università di Padova

Laurea Magistrale in Informatica

Corso di Methods and Models for Combinatorial Optimization

Relazione finale sugli esercizi di laboratorio

Marco Romanelli

marco.romanelli.1@studenti.unipd.it

matricola n. 1106706

4 giugno 2017

Indice

1 Introduzione

L'obbiettivo del progetto è quello di implementare il noto problema combinatorio del commesso viaggiatore (Traveler Salesman Problem), d'ora in avanti TSP, in due modalità differenti per poi valutarne le differenze. In particolare, la prima parte consiste nell'implementazione del problema utilizzando le API di CPLEX e trovare la dimensione massima del problema risolvibile entro un certo intervallo di tempo (fino a 1 secondo, fino a 10 secondi, eccetera). La seconda parte, invece, si chiede di studiare e implementare un algoritmo di ottimizzazione ad-hoc, utilizzando una qualsiasi meta-euristica vista a lezione. Fatto ciò si richiede di testare l'implementazione, presentarne i costi computazionali e compararne i risultati con il metodo utilizzato nella prima parte.

La presente relazione procederà come segue: nella prossima sezione verrà presentata la generazione delle istanze esponendo le caratteristiche di queste. La sezione 3 presenterà la prima parte del progetto, partendo dalla soluzione con CPLEX e finendo con i risultati ottenuti. La sezione 4, invece, esporrà la seconda parte del progetto. Nella sezione 5 verrà introdotto brevemente il metodo di utilizzo del codice, come ad esempio compilazione ed esecuzione. Per concludere, nella sezione 6 verranno presentate le conclusione finali.

2 Generazione istanze

È ovvio come per prima cosa sia necessario creare delle istanze per il problema che sia andrà ad analizzare. Un'istanza di un problema di foratura (come descritto nella prima parte della consegna) può essere rappresentata da un grafo dove i nodi sono i fori e gli archi con relativo peso sono gli spostamenti e i costi che la trivella deve effettuare per muoversi. Rappresentando questo tramite una matrice dei costi, il problema ora è come generare tali nodi e distanze; i fori nei pannelli perforati, infatti, non sono distribuiti casualmente sullo spazio ma seguono delle logiche intrinseche del problema che si fa a risolvere.

Per cercare, quindi, di riprodurre almeno in parte questa distribuzione si è utilizzato una collezione di istanze per TSP basate su un data-set di VLSI, fornite da Andre Roh dell'Istituto di Ricerca per la Matematica Discreta dell'Università di Bonn, consultabili gratuitamente sul sito web ufficiale [1]. Un esempio di tale istanze si può vedere in figura 1.

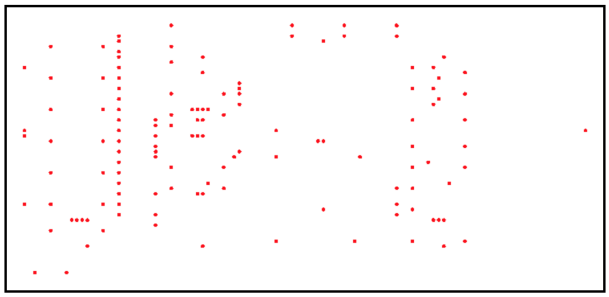


Figura 1: Esempio di problema TSP con 131 città.

Generazione istanze pseudo-casuali Queste istanze hanno però un numero di nodi molto elevato rispetto alle necessità e alle risorse di questa analisi. Inoltre, data la natura probabilistica dell'algoritmo previsto per la soluzione della parte 2 della consegna, è necessario eseguire le soluzioni proposte su molte istanze diverse.

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

Per questo motivo è stato implementato un generatore di istanze pseudo-casuali che, a partire dalle istanze TSP descritte in precedenza, generi delle sotto-istanze di dimensioni minori. Dati i punti dell'istanza originale, il generatore prima estrare casualmente un sotto-insieme di questi di dimensione variabile per poi calcolare le distanze (futuri costi) sui nuovi punti.

Da una singola istanza originale ne vengono generate 20 nuove, partendo da una dimensione $n = 5$ e iterativamente crandone una con $i*n$ nodi, fino ad arrivare a un problema con dimensione 100. Per creare poi diverse istanze con la stessa dimensione, il generatore viene eseguito su 6 istanze TSP originali, per avere alla fine del processo 120 nuove istanze pseudo-casuali.

Per l'implementazione è stato creato appositamente uno script scritto nel linguaggio Python.

3 CPLEX

Il modello è già ampiamente descritto nella consegna, per cui verrà esposto la tecnica di implementazione delle variabili e dei vincoli utilizzando le API di CPLEX ed eventuali modifiche e/o accorgimenti all'implementazione standard.

3.1 Definizione delle variabili

Nel modello sono presenti due variabili le x_{ij} e le y_{ij}. A titolo esemplificativo, prendiamo le prime:

$$x_{ij} = \text{ amount of the flow shipped from } i \text{ to } j, \forall (i,j) \in A;$$

L'implementazione corrispondente è:

```
for (unsigned int i = 0; i < N; ++i) {
  for (unsigned int j = 0; j < N; ++j) {
    if (i == j) continue;

    char htype = 'I';
    double obj = 0.0;
    double lb = 0.0;
    double ub = CPX_INFBOUND;
    snprintf(name, NAME_SIZE, "x_%d,%d", nodes[i], nodes[j]);
    char* xname = &name[0];
    CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &htype, &xname );
    xMap[i][j] = created_vars;
    created_vars++;
  }
}
```

Codice 1: Creazione delle variabili x_{ij}

Una volta create, le variabili sono memorizzate internamente al risolutore e l'unico modo per accedervi è tramite la sua posizione nell'array interno. Per semplificare questo processo, viene creata una matrice NxN che associa il nome della variabile alla sua posizione interna del risolutore. Mentre viene tenuta traccia del numero di variabili create, l'indice della variabile corrente viene memorizzato nella mappa di supporto (xMap).

Un'altra osservazione che si può fare sul codice è il fatto che non vengono create le variabili x_{ij} quando gli indici sono uguali; ovviamente non si avrebbe guadagno nel farlo, infatti se si pensa al problema reale, spostare la trivella lungo l'argo (i, i) equivale a lasciarla ferma.

3.2 Definizione dei vincoli

Sebbene le API di CPLEX permettano la definizione di più vincoli in una sola chiamata, si è scelto di definire un vincolo per chiamata; in questo modo la definizione risulta più chiara e

quindi più semplice da modificare in un futuro. Il costo computazione derivante da tale scelta è comunque limitato, in quanto la definizione del problema avviene solo in fase di inizializzazione e questa fase ha un costo molto inferiore rispetto alla fase di ottimizzazione.

```
std::vector<int> varIndex(N-1);
std::vector<double> coef(N-1);

int idx = 0;
for (unsigned int j = 0; j < N; ++j) {
  if (j == STARTING_NODE) continue;
  varIndex[idx] = xMap[STARTING_NODE][j];
  coef[idx] = 1;
  idx++;
}

char sense = 'E';
double rhs = N;
snprintf(name, NAME_SIZE, "flux");
char* cname = (char*)(&name[0]);

int matbeg = 0;
CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, varIndex.size(), &rhs, &sense, &
    matbeg, &varIndex[0], &coef[0], NULL, &cname );
}
```

Codice 2: Creazione di un vincolo

I parametri della chiamata CPLEX sono (alcuni): il numero di variabili e vincoli da creare, il numero di variabili nel vincolo dove il coefficiente è diverso da zero, la parte destra del vincolo e il verso (in questo caso definisce l'uguale "="), il vettore rmatbeg (posto a zero in quanto creo un solo vincolo [2]), l'inizio dell'array con gli indici della variabili e poi quello con gli indici dei coefficienti e infine il nome del vincolo.

3.3 Risultati

4 Algoritmo Genetico

Come da consegna, la seconda parte prevedeva l'implementazione di un algortimo ad-hoc, scegliendo fra le varie metaeuristica presenti nelle dispense del corso; fra queste, si è scelta la classe dei metodi basati su popolazione e, in particolare, gli Algoritmi Genetici, molto diffusi grazie alla loro adattabilità e semplicità implementativa.

4.1 Operatori genetici

Trattandosi di una metaeuristica, lo schema principale è generale e lascia molto spazio di manovra. Si inzia con la *codifica delle soluzioni*, per poi definire i diversi *operatori genetici*:

- la generazione di un insieme di soluzioni (popolazione iniziale);
- funzione che valuta la fitness di una soluzione;
- operatori di ricombinazione;
- operatori di passaggio generazionale.

Per la descrizione della metaeuristica si rimanda alle dispsense, mentre qui si evenzieranno solo le scelte progettuali attuate.

[2] Non è necessario utilizzare il vettore per tenere traccia delle righe, perché definendo un solo vincolo si ha una sola riga che inizierà alla posizione 0.

4.1.1 Codifica degli individui

Un individuo della popolazione è una seguenza di $N + 1$ geni, dove N è la dimensione del problema e dove ciascun gene in posizione i indica un nodo (o foro per la trivella) da visitare in posizione i nel ciclo hamiltoniano (il percorso che dovrà fare la trivella). La sequenza ha dimensione maggiore di 1 rispetto alla dimensione del problema per codificare direttamente il vincolo che la trivella ritorni al punto di partenza; l'ultimo gene, infatti, avrà sempre valore uguale al primo, cioè 0.

4.1.2 Popolazione iniziale

La popolazione iniziale viene creata generando delle soluzioni (individui) in modo pseudo-greedy, ovvero incrementalmente a partire dal nodo inziale, scegliendo il successivo casualmente fra quelli ancora da visitare, ma dando più probabilità a quelli con costo minore. Questo per permettere sia una diversificazione degli individui (parte probabilistica) mantentendo tuttavia una convergenza abbastanza veloce (scelte pesate).

La dimensione è specificata tramite un parametro ed è direttamente proporzionale alla dimensione del problema; è ragionevole pensare, infatti, che istanze maggiori abbiano un spazio delle soluzioni maggiore.

4.1.3 Funzione di fitness

La funzione di fitness serve a dare una misura quantitiva della bontà di un individuo e guida molti dei processi evolutivi dell'algoritmo. Per questo motivo, è stata scelta la funzione obbiettivo del modello, ovvero il costo del ciclo della soluzione.

4.1.4 Operatori di selezione

Gli operatori di selezione scelgono tra la popolazione corrente gli invidui che partecipano ai processi riproduttivi. La scelta dev'essere in parte pesata sugli invidui migliori, in modo da trasmettere le migliori caratteristiche alla progenie, ma anche una componente casuale che permetta di convergere più lentamente e introdurre caratteristiche diverse che potrebbero rivelarsi anch'esse migliori.

Sono stati proposti diversi metodi per combinare questi due aspetti, ma la scelta è ricaduta sul metodo chiamato *Torneo-n*. Vengono quindi prima scelti n individui e poi fra questi estratto il miglior candidato; il processo viene ripetuto una seconda volta in modo da ottenere due genitori.

Il valore degli n individui è impostato sul 20% della popolazione.

4.1.5 Operatori di ricombinazione

Gli operatori di ricombinazione, chimati in inglese di *crossover*,generano uno o più figli a partire dai genitori. Come già detto, i genitori in questo caso sono due (come avviene nella maggioranza dei casi) e viene generato un solo individuo figlio utilizzando il metodo noto come *cut-cut crossover*. Il metodo nasce dall'ipotesi che geni vicini fra loro controllino caratteristiche tra loro correlate e, quindi, affinché i figli preservino tali caratteristiche sia necessario trasmettere blocchi di geni contigui. In dettaglio, vengono definiti casualmente 2 punti di taglio (e quindi sarà un 2-cut crossover) e il figlio si ottiene copiano i blocchi dei genitori alternativamente. Per preservare l'ammissibilità dei figli (e generare quindi soluzioni ammissibili) è stato implementata una variante del metodo appena descritto, ovvero l'*order crossover*. In questa tecnica, il figlio riporta le parti esterne dal primo genitore mentre i restanti geni (blocco interno del figlio) si ottengono copiando i geni mancanti nell'ordine in cui appiaiono nel secondo genitore. Si noti che questo metodo preserva il vincolo che il primo e l'ultimo gene siano entrambi pari a 0.

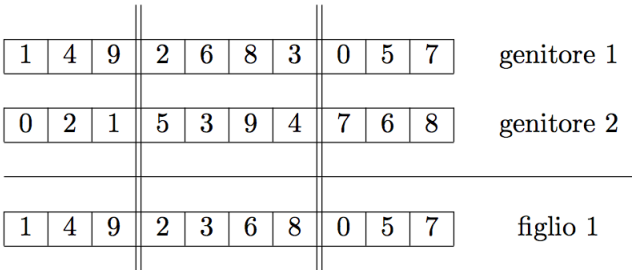


Figura 2: Esempio di 2-cut-point crossover con un figlio.

4.1.6 Mutazione

Per evitare il fenomeno chiamato assorbimento genetico (convergenza casuale di uno o più geni verso lo stesso valore), l'operatore di mutazinone modifica casualmente il valore di alcuni geni, scelti casualmente anch'essi.

Anche in questo, caso è stato implementato un metodo che preserva l'ammissibilità, chiamato *mutazione per inversione*, dove sono generati casualmente due punti della sequenza e si inverte la sottosequenza tra questi.

La probabilità di mutazione è modificabile tramite un apposito parametro ed è solitamente molto bassa.

4.1.7 Sostituzione della popolazione

A questo punto si ha un nuovo insieme di individui che va sostituito alla generazione corrente. Diverse tecniche sono state proposte e si è scelto di implementare la politica chiamata *selezione dei migliori*. Si mantengono nella popolazione corrente gli N individui migliori tra gli $N + R$ (con N dimensione della popolazione corrente e R i nuovi individui generati), effettuando la selezione utilizzando il metodo Montecarlo con probabilità proporzionale al valore di fitness.

4.1.8 Criterio di arresto

I possibili criteri per l'arresto sono diversi e sono possibili anche combinazioni fra questi. In questo caso si è scelto un metodo ibrido che prevede un numero massimo di iterazioni (evoluzioni della popolazione) e tempo massimo d'esecuzione, entrambi specificabili come parametri all'avvio dell'algoritmo.

Il tempo limite permette un confronto più semplice con il metodo CPLEX (i tempi limite sono infatti uguali) ed evita che l'algoritmo richieda troppo tempo quando la dimensione del problema è molto grande; il numero massimo di iterazioni permette invece di limitare il tempo d'esecuzione quando le istanze hanno una bassa dimensione.

4.2 Valore dei parametri

4.3 Risultati

A Note sul codice

A.1 Utilizzo

La cartella contenente il codice è strutturata in modo tenere separati i sorgenti, i binari, il codice generato, i file Header, scripts e altri file utili all'esecuzione. Per questo motivo il codice viene fornito con un Makefile che include al suo interno i comandi necessari alla compilazione e all'esecuzione dei test. Per un corretto utilizzo sono necessari alcuni passi da eseguirsi sul terminale di un computer con sistema operativo basato su Linux. Il codice viene fornito sottoforma di archivio, quindi per prima cosa è necessario estrarre i file contenuti all'interno:

```
1 tar -zxvf codice.tar.gz
```

per poi spostarsi all'interno della cartella appena creata:

```
1 cd codice/
```

Per compilare i sorgenti è sufficiente lanciare il comando Make:

```
1 make
```

Se la compilazione termina con successo, all'interno della cartella bin (per ulteriori dettagli si rimanda all'apposita sotto-sezione) troveremo l'eseguibile principale. Tuttavia si rende necessario un passo intermedio: bisogna infatti generare le istanze su cui si andrà poi a eseguire il binario. Per far ciò è sufficiente digitare

```
1 make gen-instances
```

e, una volta completato il processo, avremmo l'insieme delle istanze, generate secondo il metodo descritto nella sezione 2.

Infine, si può scegliere se eseguire gli algoritmi su tutto l'insieme di istanze:

```
1 make run
```

oppure se eseguire sono un test su una sola istanza a scelta:

```
1 make test
```

A.2 Struttura della cartella

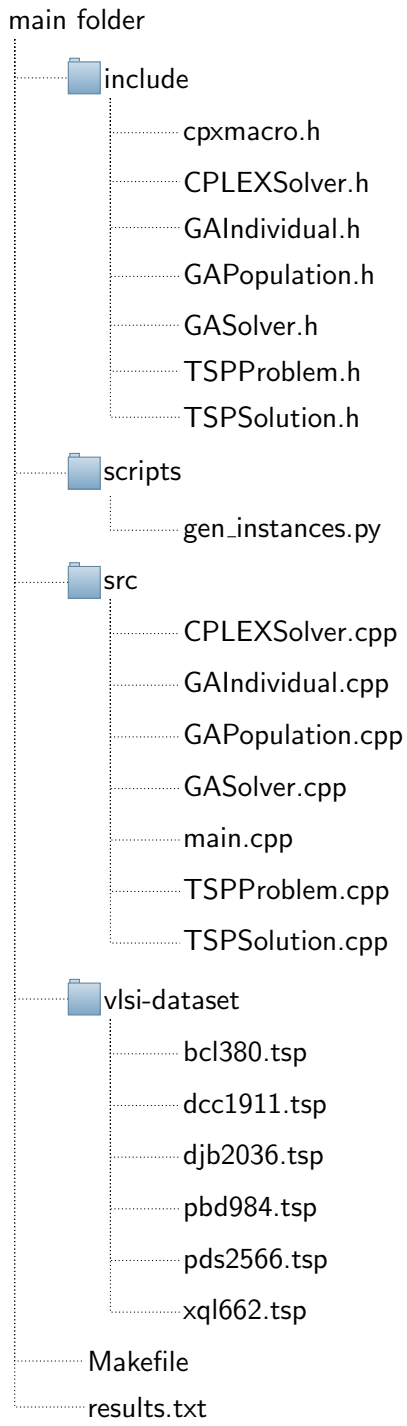
Come già accennato, la disposizione dei file all'interno della cartella è ben strutturata e definita, ma può risultare dispersiva a un estraneo. Per questo motivo, viene qui fatta una panoramica dalla disposizione delle cartelle e dei file al suo interno. Come si può vedere dalla figura A.2, all'interno della cartella pricipale troviamo:

- bin: contiene il file binario generato dalla compilazione;
- include: all'interno sono presenti tutti i file header;
- instances: una volta generate, le istanze si troveranno qui dentro;
- scripts: si trova lo script in python che si occupa della generazione delle istanze;
- src: i sorgenti per gli algoritmi risolutivi delle due parti;

- vlsi-dataset: contiente i file delle istanze originali, da cui poi verranno generate tutte le altre.

I sorgenti sono molteplici e la loro descrizione è presente all'interno nelle prime righe; Tuttavia, le parti fondamentali degli algoritmi richiesti sono contenute solamente in alcuni, che sono:

- CPLEXSover: contiene la risoluzione del problema posto utilizzando le API di CPLEX (risoluzione parte 1);
- GASolver e GAPopulation: qui si trova la risoluzione mediante l'agoritmo genetico (parte 2);
- Main: principalmente si occupa di iterare gli algoritmi su tutte le istanze, ma può essere visto come esempio su come tilizzare appropriatamente le classi e i metodi per la risoluzione di un problema (ad esempio nella funzione single_test);
- results.txt: contiene i risultati delle esecuzioni sulle istanze (presente solo dopo l'esecuzione).



A.3 Note

Utilizzando il Makefile non è necessario conoscere i nomi dei file o i loro parametri ed è sufficiente utilizzare i comandi descritti in precedenza (sotto-capitolo A.1) per compilare ed eseguire il codice. Tuttavia potrebbe rendersi necessaria una modifica al Makefile nel caso in cui il percorso di installazione di CPLEX sia diverso da quello indicato. [3] In questo caso è necessario aprire il file Makefile con un editor di testo (nano, emacs, gedit, sublime) e modificare il percorso a CPLEX, indicato nelle voci CPX_INCDIR e CPX_LIBDIR.

[3] Il Makefile presente nel codice conseganto al docente è già stato adattato per funzionare nei computer del laboratorio. La modifica è necessaria nel caso si voglia compilare su una macchina diversa.

```
1 CPX_INCDIR := <PERCORSO_CPLEX>/include/
2 CPX_LIBDIR := <PERCORSO_CPLEX>/lib/x86-64_linux/static_pic
```