

SCUOLA UNIVERSITARIA DELLA SVIZZERA ITALIANA (SUPSI)

C08003 - Advanced Algorithms and Optimization

19th Algorithms Cup

Solving the Travelling Salesman Problem (TSP) with Java

Denys Vitali

April 2019

Contents

1	Introduzione	2
1.1	Scopo	2
1.2	Requisiti	2
1.3	Problema del commesso viaggiatore (TSP)	2
2	Svolgimento del progetto	4
2.1	I/O	4
3	Algoritmi Utilizzati	4
3.1	Costruttivi	4
3.1.1	Nearest Neighbour	4
3.1.2	Random Nearest Neighbour	4
3.2	Ottimizzazione Locale	5
3.2.1	Two Opt	5
3.2.2	Three Opt	5
3.2.3	Double Bridge	5
3.3	Meta euristici	5
3.3.1	Ant Colony Optimization - ACS	5
3.3.2	Genetic Algorithm	6
3.3.3	Simulated Annealing	6
4	Esecuzione	7
5	Risultati Ottenuti	7
6	Conclusioni	7

1 Introduzione

Il presente documento contiene le informazioni riguardanti il mio metodo di risoluzione del problema del commesso viaggiatore (Travelling salesman problem, TSP). Questo progetto è un requisito del corso di *Algoritmi Avanzati ed Ottimizzazione (C08003)*, tenuto presso la *Scuola Universitaria Professionale della Svizzera Italiana (SUPSI)* durante il corso di laurea in ingegneria informatica.

1.1 Scopo

Lo scopo del progetto è quello di risolvere 10 problemi di TSP (originati dalla libreria TSPLIB) fornirci ad inizio corso sfruttando uno o più algoritmi sviluppati in Java. Nello specifico, i problemi da risolvere provengono dalla libreria TSPLIB, e sono i seguenti:

- ch130
- d198
- eil76
- fl1577
- kroA100
- lin318
- pcb442
- pr439
- rat783
- u1060

1.2 Requisiti

Di seguito vengono riportati i requisiti per un corretto svolgimento della coppa:

- Limite di tempo: massimo 3 minuti per problema, il tempo è inteso da quando il software inizia a quando finisce.
- 10 problemi (forniti in un file .zip ad inizio corso)
- Linguaggio di programmazione: Java
- Obbligatorio l'utilizzo di Maven, vietato l'utilizzo di librerie esterne
- Esecuzione replicabile: è necessario salvare eventuali seed e parametri da usare
- Risoluzione dei problemi: effettuata mediante tests. Deve essere possibile eseguire un test per risolvere un problema.

1.3 Problema del commesso viaggiatore (TSP)

Il problema che andremo a risolvere attraverso il nostro algoritmo è quello del commesso viaggiatore. La descrizione può essere formalizzata come segue: un commesso viaggiatore necessita di visitare n città, al massimo una volta, partendo da una città di partenza A e tornando alla stessa dopo averle visitate tutte. Si chiede di trovare qual è il percorso ottimale (ossia il percorso più breve) che visiti tutte le città.

Il problema, di tipo NP-hard, viene espresso matematicamente nel modo seguente: dato un grafo pesato e completo $G = (V, E, w)$ con n vertici, determinare il ciclo hamiltoniano con il costo minore.

Data una matrice di incidenza C (dove $c_{i,j} \in \{0, 1\}$), ed una matrice dei pesi w , la funzione obiettivo

è la seguente:

$$\min \sum_{i=0}^n \sum_{j=0}^n c_{i,j} \cdot w_{i,j} \quad (1)$$

Dato che la computazione e verifica di tutte le $n!$ possibilità richiederebbe un tempo troppo elevato (ed il tempo per la risoluzione di un problema è limitato a 3 minuti), nel nostro progetto ci accontenteremo di una "buona" soluzione, utilizzando dei metodi euristici.

2 Svolgimento del progetto

2.1 I/O

Quale step iniziale e fondamentale è stato necessario sviluppare un parser dei problemi forniti, in quanto questi venivano forniti in un formato proprietario che sfrutta la struttura seguente:

```
1  NAME: ch130
2  TYPE: TSP
3  COMMENT: 130 city problem (Churritz)
4  DIMENSION: 130
5  EDGE_WEIGHT_TYPE: EUC_2D
6  BEST_KNOWN : 6110
7  NODE_COORD_SECTION
8  1 334.5909245845 161.7809319139
9  2 397.6446634067 262.8165330708
10 (...)
11 EOF
```

Al fine di sfruttare al meglio il tempo a disposizione per i runs ho deciso di spendere un po' più del mio tempo a sviluppare un parser che fosse efficiente e non facesse uso di Regular Expressions. Questo ha portato alla creazione della classe `TSPLoader`. Una volta caricato un problema (tramite il metodo `parseFile()`) viene restituita un'istanza della classe `TSPData`. Questa classe contiene i file di problema che verranno poi utilizzati in qualsiasi algoritmo di risoluzione.

3 Algoritmi Utilizzati

3.1 Costruttivi

Gli algoritmi di tipo costruttivo sono raggruppati nel package: `TSP.ra.initial`

3.1.1 Nearest Neighbour

L'algoritmo di Nearest Neighbour, ritorna il percorso generato partendo dal nodo di partenza scelto ed utilizzando il nodo più vicino nel percorso. Questo algoritmo è utilizzato quale inizializzatore per gli algoritmi basati su Simulated Annealing.

3.1.2 Random Nearest Neighbour

L'algoritmo di Random Nearest Neighbour funziona in modo analogo a quello di Nearest Neighbour, con la sola differenza che, al momento della scelta del prossimo nodo da visitare, prende in considerazione più archi e ne sceglie uno in modo casuale.

3.2 Ottimizzazione Locale

Gli algoritmi di ottimizzazione locale sono raggruppati nel package `TSP.ra.intermediate`, insieme ad alcuni di quelli meta-euristici. L'idea della suddivisione è data dal fatto che, dato un algoritmo costruttivo ("initial"), è possibile combinare uno o più ottimizzatori locali ("intermediate" / ILS¹) con limitazioni di tempo e parametrizzazioni particolari.

È così possibile, grazie all'organizzazione delle classi, eseguire il seguente snippet in un test per combinare a piacere gli algoritmi implementati:

```
1 private void fl1577_SA(int seed) throws IOException {
2     TSPData data = getProblemData("fl1577");
3     TSP tsp = new TSP();
4     tsp.init(data);
5
6     Route r = tsp.run(data,
7         (new CompositeRoutingAlgorithm())
8             .startWith(new RandomNearestNeighbour(seed, data))
9             .add(new TwoOpt(data))
10            .add(new SimulatedAnnealing(seed))
11            .setMode(SimulatedAnnealing.Mode.DoubleBridge));
12
13     validateResult(tsp, r, data);
14 }
15
```

Figure 1: Esecuzione di fl1577 tramite RNN, 2-opt e Simulated Annealing in modalità Double Bridge

3.2.1 Two Opt

L'algoritmo di 2-opt (anche chiamato 2-exchange) è stato implementato seguendo i consigli ed il codice fornito dal CS Department of the Colorado State University. La mia prima implementazione risultava parecchio lenta, in quanto creavo un nuovo array ed un nuovo oggetto ad ogni 2-Opt swap.

Inoltre, nelle mie versioni precedenti ad ogni iterazione di 2-Opt calcolavo la lunghezza finale del percorso anziché il guadagno. Ciò comportava a numerosi calcoli inutili, che sono stati ridotti grazie al miglioramento apportato dalla regola della disuguaglianza triangolare.

3.2.2 Three Opt

Nel mio solver è anche presente un'implementazione di 3-opt, questa non viene però utilizzata in nessuna delle mie risoluzioni.

3.2.3 Double Bridge

Al fine di sfruttare una mossa di perturbazione per l'algoritmo di Simulated Annealing ho implementato la mossa di Double Bridge (una specifica mossa 4-opt che non è reversibile da delle mosse 2-opt).

3.3 Meta euristici

3.3.1 Ant Colony Optimization - ACS

Seguendo alcuni consigli e direttive date da "Ant Colony Optimization" [1] ho implementato l'algoritmo di Ant Colony System utilizzando i seguenti parametri:

¹Intermediate Local Search

α	1
β	2
ρ	0.1
ε	0.1
q_0	0.98

Table 1: Parametri per ACS

3.3.2 Genetic Algorithm

Per l'algoritmo genetico ho fatto uso di una funzione di crossover ampiamente discussa e rivisitata, ottimizzata per il TSP, chiamata EAX (Edge Assembly Crossover).

I cicli AB sono selezionati tramite una funzione `EAX.rand()` ed `EAX.heur()` creando un *E-Set*. Successivamente dall'*E-Set* viene generato un'insieme E_C nel modo seguente:

$$E_C := (E_A \cap \bar{D}) \cup (E_B \cap D) \quad (2)$$

dove D è l'*E-Set*.

Alternativamente, ed in modo equivalente, si può definire E_C nel modo seguente:

$$E_C := (E_A \setminus (E\text{-}set \cap E_A)) \cup (E\text{-}Set \cap E_B) \quad (3)$$

3.3.3 Simulated Annealing

4 Esecuzione

Per eseguire il solver è necessario utilizzare Maven ed utilizzare Java 11. Una volta clonata la repository (da <https://github.com/denysvitali/tsp-cup-2019>) è necessario eseguire il seguente comando:

```
1 mvn -Dtest=TSPRunnerTest#ch130 test
```

dove ch130 corrisponde al nome del problema da risolvere.

5 Risultati Ottenuti

La piattaforma utilizzata quale piattaforma di benchmarking è la seguente:

OS	Arch Linux
Kernel	5.1.0-rc6-mainline x86_64
CPU	Intel Core i7-6700HQ CPU @ 2.60GHz
RAM	32 GB

6 Conclusioni

References

- [1] Marco Dorigo, Thomas Stützle, *Ant Colony Optimization*, 2004.