Scuola Universitaria della Svizzera Italiana (SUPSI)

C08003 - Advanced Algorithms and Optimization

19^{th} 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 **8**

5 Risultati Ottenuti **8**
- 5.1 Ricerca di Seed . . . 8

6 Conclusioni **8**

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) forniteci 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:

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

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:

```
NAME: ch130
TYPE: TSP
COMMENT: 130 city problem (Churritz)
DIMENSION: 130
EDGE_WEIGHT_TYPE: EUC_2D
BEST_KNOWN : 6110
NODE_COORD_SECTION
1 334.5909245845 161.7809319139
2 397.6446634067 262.8165330708
(...)
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" / ILS[1]) 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
14         validateResult(tsp, r, data);
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 diseguaglianza 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 su alcuni libri e paper relativi all'argomento [1] [2] ho implementato l'algoritmo di Ant Colony System utilizzando i seguenti parametri:

[1] Intermediate Local Search

Parametro	Descrizione	Valore
α	Importanza del feromone	1
β	Importanza dell'euristica	2
ρ	Deterioramento del feromone	0.1
ε	Evaporazione del feromone	0.1
q_0	Fattore di esplorazione	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 $E\text{-}Set$. Successivamente dall'$E\text{-}Set$ viene generato un'insieme E_C nel modo seguente:

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

dove D è l'$E\text{-}Set$.

Dopo numerosi tentativi verso la strada del genetico con EAX, ho deciso di fermarmi in quanto non riuscivo ad ottenere un'implementazione valida. Purtroppo il poco tempo a mia disposizione ed una scarsa documentazione a riguardo dell'argomento mi hanno portato ad abbandonare questa strada a poche settimane dalla consegna. Sono però convinto che questa strada mi avrebbe portato ad ottimi risultati una volta implementata.

L'algoritmo genetico con un crossover base ed inefficiente è però stata implementata in `4ce199d`. Senza una funzione di crossover efficiente, questo algoritmo porta a scarsi risultati.

3.3.3 Simulated Annealing

Sconsolato dai risultati ottenuti tramite ACO / GA + EAX, ho deciso di provare a migliorare il mio algoritmo di Simulated Annealing.

Inizialmente la funzione (non deterministica) di scheduling della temperatura era definita nel modo seguente:

$$T = T_0 \cdot (1 - \frac{\text{now} - \text{start}}{\text{max\_runtime}}), \quad \text{con} \quad T_0 = 100.0 \tag{3}$$

dove *now* era un'indicazione del tempo corrente, (in ms), *start* il tempo di inizio (in ms) e *max_runtime* il tempo massimo di run (definito come 2 min e 50 secondi). Sfortunatamente questa funzione non è deterministica in quanto dipende dal numero di operazioni effettuate in un dato lasso di tempo. Usando una funzione non deterministica per la generazione del valore di temperatura ottengo automaticamente un risultato non deterministico per ogni run.

Questa problematica mi ha portato a cambiare la funzione di scheduling della temperatura nel modo seguente:

$$T_{k+1} = T_k \cdot \alpha^i, \quad \text{con} \quad T_0 = 100.0 \tag{4}$$

dove i è il numero di iterazioni, ed α un parametro impostato ad $\alpha = 0.98$.

A seguito di scarsi risultati usando anche questa formulazione, ho ritenuto più saggio utilizzare la formulazione standard e concentrarmi invece sul metodo di perturbazione utilizzato.

Ho quindi utilizzato infine la seguente formula di scheduling della temperatura:

$$T_{k+1} = T_k \cdot \alpha, \quad \text{con} \quad T_0 = [100.0, 200.0] \tag{5}$$

Perturbazione Nonostante le candidate lists (impostate ad una lunghezza di 80 per Nearest Neighbour) e l'algoritmo efficiente utilizzato, ottenevo mediamente scarsi risultati. Ho quindi implementato una versione personalizzata per la perturbazione. Inizialmente effettuavo una perturbazione random su quattro punti utilizzando la tecnica di Double Bridge. Ho migliorato questa tecnica per perturbare maggiormente il percorso inventando la perturbazione `RAND_CHOICE`.

```
switch(mode){
        case DoubleBridge:
        next = current.doubleBridge(
        getRandomOffsettedNumbers(4, length-1)
        );
        break;
        case RAND_CHOICE:
        if(random.nextBoolean()) {
                int[] ij = getRandomNumbers(2, length - 1);
                Arrays.sort(ij);
                next = current.twoOptSwap(ij[0], ij[1]);

                next = next.doubleBridge(
                getRandomOffsettedNumbers(4, length-1)
                );

            } else {
                next = current.doubleBridge(
                getRandomOffsettedNumbers(4, length-1)
                );
            }
        break;
    }
```

Casualmente, ogni qualvolta sia necessaria una perturbazione, viene scelta una variabile booleana che definisce se utilizzare Two Opt + Double Bridge oppure solamente Double Bridge.

Mediante questa soluzione ottengo risultati migliori.

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:

```
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

5.1 Ricerca di Seed

Al fine di ottenere il seed che fornisse le migliori prestazioni possibili, ho sfruttato la potenza di calcolo a mia disposizione ed ho implementato un risolutore di problemi TSP parallelo, sfruttando 30 - 35 nodi, per un totale di 162 cores.

Inoltre, per raccogliere i dati da tutte queste macchine, ho implementato un server HTTP scritto in Go che si occupa di ricevere richieste HTTP dai nodi. Le macchine (per la maggior parte) sono state avviate con una versione del sistema operativo Alpine Linux personalizzato per contenere Java 12, Maven ed alcuni tool utili al fine di ottenere controllo remoto.

Il server si connette ad un database PostgreSQL e registra i dati di tutte le istanze del TSP Solver eseguite. Al 3 Maggio 2019 il numero di risultati ottenuti ammontava a 27'776.

Questa ricerca ha portato ad i seguenti risultati:

Problem	**Seed**	α	S_T	r	**Length**	**T[s]**	**Error Percentage**
ch130	-2129375319	0.982	80.5	400	6110	0.046744592	0.00%
d198	-796751895	0.952	80	100	15780	3.466862365	0.00%
eil76	3	0.984018	100	400	538	0.028927185	0.00%
fl1577	-584050420	0.971	80	100	22466	165.491784	0.98%
kroA100	916773322	0.999	100	300	21282	0.00270346	0.00%
lin318	-357553184	0.951	100	100	42029	21.70859944	0.00%
pcb442	1694800648	0.97	100	100	50778	95.49048343	0.00%
pr439	-729289858	0.952	100	100	107217	41.53711452	0.00%
rat783	1032714840	0.956	80.5	100	8811	150.5788728	0.06%
u1060	269590158	0.978	80	100	224439	152.1944992	0.15%

Table 2: Risultati ottenuti

6 Conclusioni

Una volta testato l'algoritmo, e notato che alcuni compagni (che nel corso del semestre non hanno scritto una singola riga di codice) si sono copiati il codice a vicenda, mi sono demotivato ed ho interrotto la mia analisi. Trovate in allegato al presente documento il mio codice ed i miei risultati.

Sfortunatamente il mio impegno non ha portato ai risultati desiderati. Si riconferma nuovamente che nella Scuola Universitaria della Svizzera Italiana (SUPSI) c'è un grave problema di plagio.

References

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

[2] Marco Dorigo, Luca Maria Gambardella, IEEE Transactions On Evolutionary Computation, Vol. 1, No. 1 *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*, 1997.