Scuola Universitaria Professionale
della Svizzera Italiana

Dipartimento
Tecnologie
Innovative

Traveling Salesman Problem
18ª Coppa di Algoritmi

Studente: Brian Dalle Pezze

Data: 4 maggio 2018

Problema

Il problema del commesso viaggiatore consiste nella ricerca del ciclo hamiltoniano più breve all'interno di un grafo pesato completo. Questo tipo di problema appartiene alla classe dei problemi *NP-Completi*.

Lo scopo del progetto è quello di riuscire a generare una soluzione che diverga il meno possibile da quella ottimale, con delle limitazioni:

1. L'applicazione può essere eseguita solo in single thread
2. L'applicazione ha un tempo massimo di esecuzione di 3 minuti

Tra le istanze conosciute di TSP, per il nostro progetto ne sono state fornite 10:

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

Soluzioni implementate

Algoritmo costruttivo

Come algoritmo costruttivo avevo implementato un semplice *Random* nelle prime fasi di sviluppo, ma ho poi scelto di non usarlo dal momento che peggiorava i miei risultati. Nelle fasi più avanzate del progetto mi sono quindi limitato ad usare l'ordine di lettura delle città come soluzione iniziale.

Algoritmi di ottimizzazione locale

Come ottimizzazione locale ho inizialmente implementato, un *2-opt Best Improvement*. Su suggerimento di Axel Kuhn (e poi anche del prof. Gambardella) ho apportato una lieve modifica per velocizzarne l'esecuzione, trasformandolo in un *"Best-First" Improvement*.

Algoritmi meta-euristici

Come meta-euristica ho utilizzato un *Simulated Annealing*. Per sfruttare appieno il limite dei 3 minuti di tempo, quando il sistema si è raffreddato alzo di nuovo la temperatura al massimo per far ricominciare l'algoritmo ed esplorare più soluzioni (tenendo comunque traccia della migliore).

In diverse applicazioni che ho potuto esaminare, prima di abbassare le temperatura si aspettava una situazione di "equilibrio" in modo da poter mantenere il grado di accettazione delle soluzioni più alto (soprattutto nelle prima fasi). Nel mio caso mi sono limitato ad abbassare la temperatura solo una volta ogni 50 cicli (valore arbitrario determinato durante le procedure di seeding).

Per perturbare la soluzione ho eseguito una mossa di *4-opt* (double bridge), come suggerito da Marco Cinus. Inizialmente mi limitavo a swappare tra di loro la posizione di 4 città nel tour.

Esecuzione programma

Piattaforma

La piattaforma utilizzata è stato il PC Linux (Ubuntu) fornitoci da Marco Cinus per effettuare i test su una piattaforma comune per caratteristiche.

Compilazione

Per compilare il main in modo da generare il file **.class*, basta semplicemente usare:

```
javac nome_classe.java
```

Per generare il file **.jar*, invece, il comando è il seguente:

```
jar cfm TSPSolver.jar ./META-INF/MANIFEST.MF TSPSolver.class ./Structure ./IO ./Algorithm
```

I parametri dopo "cfm" sono i seguenti:

1. Nome file jar che si vuole creare
2. Manifest che si vuole utilizzare
3. Files con cui comporre il jar (inserendo delle directory, i file al loro interno vengono aggiunti ricorsivamente)

Esecuzione

Nella struttura del progetto sono presente due tipologie di file in aggiunta a quelle richieste nella consegna:

- **.claim*: questo tipo di file contiene solo la lunghezza del risultato ottenuto dal mio algoritmo per un dato problema (che da il nome al file). Serve come supporto per lo script di controllo.
- **.config*: questo tipo di file contiene i parametri con cui configurare l'algoritmo per ogni problema, quindi il seed, la temperatura di partenza e il tasso di raffreddamento.

Per eseguire il programma è possibile digitare, nella cartella appropriata, due tipi di comando:

```
java TSPSolver -c nome_problema
```

Oppure:

```
java TSPSolver -s nome_problema temperatura cooling_rate seed
```

Dove "-c" esegue l'algoritmo con la configurazione associata al problema, mentre "-s" è il normale solver che richiede i parametri inseriti manualmente. Temperatura e Cooling Rate sono double, mentre il seed è un long. Cooling Rate dev'essere un valore tra 0 e 1 non compresi.

Al termine dell'esecuzione, verrano creati i file **.opt.tour* e **.claim*. Il file **.config* verrà aggiornato solo se il risultato rappresenta un tour migliore.

Esiste infine un comando per applicare l'algoritmo a tutti i problemi utilizzando parametri generati casualmente:

```
java TSPSolver -seed
```

Scripts

Per comodità ho deciso di implementare due script in Bash per automatizzare alcuni processi:

- *create_jar.sh*: questo script si limita, a partire dal progetto e dal manifest creato da IntelliJ, a generare il file **.jar*
- *solve_and_check_all.sh*: questo script automatizza il processo di test dell'algoritmo. Utilizzando il file **.jar*, applica l'algoritmo a tutti i problemi contenuti nella cartella *./Problems* usando i rispettivi file di configurazione (quindi con il comando "-c"). Una volta terminata questa fase, sfrutta il file **.py* che ci è stato fornito per controllare che tutte le soluzioni generate siano valide.

Ovviamente questi script funzionano solo ammettendo che la struttura del progetto nel filesystem rimanga inavriata da quella attuale.

Risultati

Nella Tabella 1 sono mostrati i risultati migliori per ogni problema. Nella cartella *./Solutions* sono riportati i relativi file **.opt.tour*. Nella Tabella 2 sono riportati i parametri associati a ogni problema.

Tabella 1: Soluzioni migliori per ogni problema

Problema	Optimum	Risultato	Errore
ch130	6110	6110	0.00%
d198	15780	15780	0.00%
eil76	538	538	0.00%
fl1577	22249	22765	2.32%
kroA100	21282	21282	0.00%
lin318	42029	42029	0.00%
pcb442	50788	50986	0.41%
pr439	107217	107252	0.03%
rat783	8806	9009	2.31%
u1060	224094	226629	1.13%
Media			0.62%

Tabella 2: Parametri per ogni problema

Problema	Temperatura	Cooling Rate	Seed
ch130	1454.0	0.7082785944635276	-7716356462269080079
d198	874.5580118634485	0.991700753955645	4999408061570499896
eil76	1930.0	0.9678235713785959	-6134614828855466962
fl1577	1412.1779138746506	0.5872129996448873	-6721988644085780744
kroA100	1663.0	0.9681210301406746	9089990836345110089
lin318	771.0	0.7763535923349907	4916838395621579886
pcb442	545.0	0.9850814764934567	5992200681231382488
pr439	1376.7010878552994	0.9828754281175798	-5230393556779708681
rat783	1495.32775227876	0.9288405456613162	7843041926870010925
u1060	797.4352524966511	0.7735954764352162	-5598247544688303158

Conclusioni

Con un errore medio inferiore al 1%, si può dire che il progetto sia stato concluso con successo. Migliorie sono sicuramente applicabili, per esempio una gestione più oculata nelle strutture dati (nell'attuale implementazione il double bridge è eseguito con dei for loop invece di usare funzioni come copyOfRange sull'array).