

Traveling Salesman Problem

20^a Coppa di Algoritmi



Studente: Nicolò Rubattu

Data: 15 maggio 2020

Problema

Il problema del commesso viaggiatore (in inglese Traveling salesman problem, da cui la sigla TSP) è un caso di studio tipico dell'informatica teorica e della teoria della complessità computazionale.

Dato un'insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza.

TSP è un problema NP-completo. Non esistono algoritmi esatti che risolvano problemi di TSP in modo efficiente. Metodo esatto di risoluzione è rappresentato dall'enumerazione totale, ovvero nell'elaborazione di tutti i possibili cammini sul grafo per la successiva scelta di quello migliore. La complessità di tale procedimento lo rende impraticabile per problemi reali, dove il numero di città/nodi del grafo è elevato ($n!$ possibili tour).

Il TSP rappresenta inoltre un problema di programmazione lineare intera dove però risulta inattuabile ottenere valutazioni tramite la tecnica del rilassamento continuo poiché il numero di vincoli crescerebbe in modo esponenziale al crescere dei nodi. In questo ambito, algoritmi esatti come il branch and bound si sono negli ultimi anni rivelati efficaci per problemi contenuti fino a poche migliaia di città.

Entrano dunque in gioco soluzioni euristiche, che hanno “alta” probabilità di produrre una “buona” soluzione “velocemente”.

Vincoli secondari del problema

- TSP simmetrico: per ogni coppia di città, la distanza tra una città e l'altra è simmetrica. Ovvero, la distanza tra A e B è uguale alla distanza tra B ed A.
- TSP euclideo: si utilizzano distanze euclidee tra le città. Città e distanze si trovano dunque sullo stesso piano, così da essere soddisfatta la disuguaglianza triangolare.
- Distanze intere
- 3 minuti per la ricerca del tour migliore
- Il programma deve poter lavorare su 10 problemi diversi ch130, d198, eil76, fl1577, kroA100, lin318, pcb442, pr439, rat783 e u10160.

Soluzioni implementate

- Algoritmo costruttivo

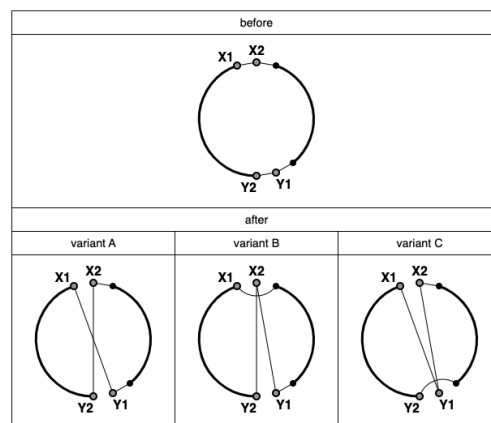
Con lo scopo di generare una prima soluzione ammissibile è stato implementato l'algoritmo **Nearest Neighbor**. L'algoritmo, partendo da un nodo iniziale, costruisce un tour scegliendo passo passo ad ogni successivo il più vicino al nodo corrente che non sia già stato visitato in precedenza ritornando poi alla città iniziale.

Peculiarità della mia implementazione

L'algoritmo riceve in ingresso il nodo di partenza e da quello costruisce il tour.

- Algoritmo di ottimizzazione locale

Sono stati implementati l'algoritmo 2-opt e **2h-opt**. Solamente quest'ultimo si trova nella versione finale del progetto. Partendo da un tour iniziale (risultato del NN), questi algoritmi verificano tutti gli scambi fra tutte le coppie di archi del grafo ed eseguono coloro che ne riducono il costo (funzione obbiettivo!). L'algoritmo 2h-opt, per ogni coppia di archi, verifica le seguenti varianti mentre il semplice 2-opt solamente la variante A.



Peculiarità della mia implementazione (2h-opt)

L'algoritmo confronta tutte le coppie possibili di archi eseguendo i dovuti miglioramenti e ripetendo ciò fintanto che nessun miglioramento è stato più apportato. Nel calcolare i miglioramenti e quindi controllare le 3 varianti (A, B e C) descritte dall'immagine sopra ho adottato la seguente procedura. Le tre varianti sono ordinate con priorità, prima viene controllata la variante A, poi la B, poi la C. Quest'ordine è stato mantenuto sempre lo stesso per tutta l'esecuzione. Non appena una variante restituisce un miglioramento lo scambio viene subito eseguito (always exchange) per passar direttamente alla prossima coppia di archi. Questo approccio si è rivelato efficace (dopo aver testato anche la best exchange) ma soprattutto mi ha permesso di avere un guadagno in termini di tempo e numero di iterazioni compiute nel successivo algoritmo meta-euristico.

- Algoritmo meta-euristico

È stato implementato l'algoritmo meta-euristico **Simulated Annealing Ibrido** con l'inserimento

della ricerca locale risultando in due fasi: la fase di diversificazione e la fase di intensificazione. L'algoritmo esegue fino allo scadere del tempo concessogli e restituisce la miglior soluzione trovata.

Peculiarità della mia implementazione

La temperatura iniziale è stata resa dipendente dal problema secondo la formula

$$T_0 = \frac{1.5 \cdot \text{costo della soluzione iniziale}}{\sqrt{n}}, \quad \text{con } n = \text{numero di nodi del problema}$$

La costante di raffreddamento è 0.95 e comune a tutti i problemi.

Ogni "livello di temperatura" è mantenuto per un certo numero di iterazioni dell'algoritmo ed è stato reso anch'esso dipendente dal problema, parametrizzato in relazione al numero di nodi del problema, con l'intento di terminare ogni esecuzione di ogni problema con un valore di temperatura finale vicino a 10^{-3} (alla fine dei 3 minuti). Questo perché il tempo di esecuzione di una local search è dipendente dalla dimensione del problema (complessità quadratica). Con questo accorgimento della temperatura i risultati sono risultati migliori. Nel dettaglio, indicando con n il numero di città del problema, 150 per $n < 400$, 70 per $n > 400$, 20 per $n > 700$, 10 per $n > 1000$ e 3 per $n > 1500$.

La fase di diversificazione è stata attuata implementando il Double Bridge. Riguardo ciò i quattro indici necessari alla mossa vengono generati casualmente ed ordinati attraverso una sorting network per ragioni di performance.

La fase di intensificazione è eseguita dalla ricerca locale del 2h-opt descritto precedentemente.

Per quanto riguarda la decisione di mantenere o scartare una soluzione peggiore di quella corrente non sono stati apportati cambiamenti alla classica formula probabilistica.

L'esecuzione termina qualora venisse raggiunto l'ottimo (ricevuto in input, nello dal file del problema) per risparmiare del tempo e non correre per niente.

Per ogni problema, il tour iniziale con cui l'algoritmo ibrido meta-euristico inizia (SimulatedAnnealing) è il migliore tour tra tutte le combinazioni di Nearest Neighbor + 2h-opt ottenute variando il nodo di partenza da 1 a N (numero di nodi del problema).

La seguente tabella racchiude questi risultati.

Problema	Città di partenza	Errore	Tempo di calcolo
<i>ch130</i>	18	1.47 %	~ 300 ms
<i>d198</i>	31	0.71 %	~ 600 ms
<i>eil76</i>	1	0 %	~ 20 ms
<i>fl1577</i>	461	2.57 %	~ 10 min
<i>kroA100</i>	17	0.18 %	~ 200 ms
<i>lin318</i>	97	1.69 %	~ 2 s
<i>pcb442</i>	221	1.22 %	~ 6 s
<i>pr439</i>	301	1.77 %	~ 6 s
<i>rat783</i>	51	3.83 %	~ 40 s
<i>u1060</i>	12	3.78 %	~ 2 min

Esecuzione del programma

PIATTAFORMA

Tutti i test e l'esecuzione finale si sono tenuti sul mio portatile personale, un Macbook Pro Mid-2012 con:

Sistema Operativo	MacOS Catalina 10.15.4
Kernel	Darwin 19.4.0
Processore	2.6 GHz Intel Core i7-3720QM quad-core
RAM	8 GB 1600 MHz DDR3

Tenuta considerazione della datazione del mio hardware non ho apportato modifiche ai 3 minuti stabiliti. Tutti i run di ricerca seed sono stati condotti con tempo pari a 2 minuti e 50 secondi, batteria al 100%, presa di corrente e single-process con focus sulla finestra Terminale (per ovviare allo scheduling con priorità del sistema operativo).

COMPILAZIONE ED ESECUZIONE

Posizionarsi nella root del progetto (dove è presente il file pom.xml).

Con MAVEN:

- Per compilare:
`mvn clean install`
- Per compilare saltando i test
`mvn clean install -DskipTests`
- Per eseguire tutti i test / problemi di TSP
`mvn test`
- Per eseguire un singolo test / problema TSP specifico (es: eil76)
`mvn -Dtest=SolverTest#eil76 test`

N.B. Non si necessita di nient'altro, tutti i parametri risultano già impostati singolarmente per ogni problema all'intero di ogni suo specifico test. Anche i file di input dei problemi sono già inseriti nelle resources del progetto. Per ogni problema risolto viene generato un file .opt.tour con il best tour trovato ed alcuni dettagli (es. il costo) nella root del progetto.

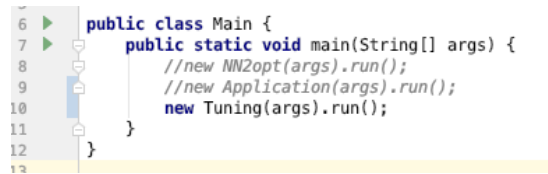
Framework per run automatizzate

I seguenti problemi di TSP non hanno richiesto particolare ricerca di *seed* in quanto il seed di default (0) li ha risolti all'ottimo con un singolo run: *ch130*, *d198*, *eil76*, *kroA100* e *lin318*.

Mentre per i rimanenti cinque problemi (*fl1577*, *pcb442*, *pr439*, *rat783*, *u1060*), i quali numeri di nodi risultano maggiori si è ricorso a tentare più run cambiando di volta di volta il seed e aggiornando man mano la soluzione best trovata assieme appunto al seed in questione che la replicasse.

Per fare ciò non è stato sviluppato nessun framework particolare ma semplicemente è stato generato un artefatto (*CoppaAlgoritmi-1.0.jar*). Questo è stato eseguito per ogni problema (specificando i parametri problem-specific da linea di comando) da un processo Terminale che aggiornava i risultati su un file di output.

L'artefatto sopracitato è stato generato eseguendo il comando package di Maven avendo nel Main riga 7 ed 8 commentate e non 9. Per capirci:



```
6 public class Main {  
7     public static void main(String[] args) {  
8         //new NN2opt(args).run();  
9         //new Application(args).run();  
10        new Tuning(args).run();  
11    }  
12 }  
13
```

La classe *Tuning* del package *core*, se visionata all'interno, altro non fa che ciò che descritto sopra.

I valori di seed di volta in volta derivano dalla chiamata `System.currentTimeMillis()`.

pcb442 e *pr439* hanno raggiunto l'ottimo dopo poche ore. Mentre per i rimanenti tre, *fl1577*, *rat783* e *u1060* sono state dedicate alcune giornate in background.

Risultati

Nella tabella seguente sono mostrati i risultati migliori per ogni problema. Nella cartella /tour sono riportati i relativi file *.opt.tour*.

<i>Problema</i>	Ottimo	Risultato	Errore	Seed
<i>ch130</i>	6'110	6'110	0 %	0
<i>d198</i>	15'780	15'780	0 %	0
<i>eil76</i>	538	538	0 %	0
<i>fl1577</i>	22'249	22'340	0.409 %	1587216352705
<i>kroA100</i>	21'282	21'282	0 %	0
<i>lin318</i>	42'029	42'029	0 %	0
<i>pcb442</i>	50'788	50'778	0 %	1586979372812
<i>pr439</i>	107'217	107'217	0 %	1586023885384
<i>rat783</i>	8'806	8'833	0.307 %	1586927476425
<i>u1060</i>	224'094	225'408	0.586 %	1587257994270
			<u>0.130 %</u>	

Errori calcolati come:

$$Errore \% = \frac{Risultato - Ottimo}{Ottimo} \cdot 100$$

Errore medio calcolato come la media dei singoli errori.

Conclusioni

Si è cercato di utilizzare meno memoria possibile e le copie di array in memoria avvengono tramite la chiamata di sistema `System.arraycopy (...)` per ragioni di performance.

Tutti i tour finali sono stati validati tramite lo script `tspTourChecker.py` fornito dal docente.

```
[tspliNico]$ python tspTourChecker.py src/main/resources/files/ch130.tsp tours/ch130.opt.tour 6110
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/d198.tsp tours/d198.opt.tour 15780
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/eil76.tsp tours/eil76.opt.tour 538
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/fl1577.tsp tours/fl1577.opt.tour 22340
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/kroA100.tsp tours/kroA100.opt.tour 21282
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/lin318.tsp tours/lin318.opt.tour 42029
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/pcb442.tsp tours/pcb442.opt.tour 50778
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/pr439.tsp tours/pr439.opt.tour 107217
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/rat783.tsp tours/rat783.opt.tour 8833
Tour is correct
[tspliNico]$ python tspTourChecker.py src/main/resources/files/u1060.tsp tours/u1060.opt.tour 225408
Tour is correct
[tspliNico]$
```

Di seguito è mostrato l'output dell'esecuzione del comando `mvn test` sulla mia macchina.

```
-----
T E S T S
-----
Running ch.supsi.rubattu.SolverTest
TSP Tuning started...
[ fl1577]   Start: 461   Seed: 1587216352705   Error: 0,409%   Best at: 594   Time: 180596
[ lin318]   Start: 97   Seed: 0   Error: 0,000%   Best at: 19272   Time: 91811
[ pcb442]   Start: 221   Seed: 1586979372812   Error: 0,000%   Best at: 13524   Time: 138924
[ rat783]   Start: 51   Seed: 1586927476425   Error: 0,307%   Best at: 4789   Time: 180334
[ kroA100]   Start: 17   Seed: 0   Error: 0,000%   Best at: 9   Time: 6
[ d198]     Start: 31   Seed: 0   Error: 0,000%   Best at: 16940   Time: 32317
[ ch130]    Start: 18   Seed: 0   Error: 0,000%   Best at: 11330   Time: 7786
[ eil76]    Start: 1   Seed: 0   Error: 0,000%   Best at: 0   Time: 1
[ pr439]    Start: 301   Seed: 1586023885384   Error: 0,000%   Best at: 10605   Time: 110638
[ u1060]    Start: 12   Seed: 1587257994270   Error: 0,586%   Best at: 2034   Time: 180318
-----
0,130%
Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 922.837 sec
```

N.B. L'output della seguente schermata ha una colonna titulata 'Best at'... si riferisce al numero dell'iterazione dell'algoritmo meta-euristico nella quale la best solution generata è stata trovata. Questo indicatore può tornare utile durante esecuzioni su macchine diverse. Nel caso, qualora l'errore risultasse inferiore rispetto ai miei risultati, e dunque anche il 'best at', significa che l'algoritmo ha corso più iterazioni nei 3 minuti trovando un risultato migliore. Qualora invece l'indicatore risultasse inferiore al valore stampato sulla schermata, significa che l'esecuzione ha bisogno di più tempo per pareggiare il numero di iterazioni svolte, almeno a pareggiare il mio valore di 'best at'.

Confrontandomi con il docente ed i compagni di corso ritengo molto buoni i risultati raggiunti.

L'intero codice e sviluppo del progetto è in allegato a questo documento ma è anche stato versionato ed è consultabile su github.com/nicorbtt/tsp.