Relazione laboratori Metodi e Modelli per l'Ottimizzazione Combinatoria

Rossi Daniel

Febbraio 2020

1 Istanze

Le istanze utilizzate per il test sono nel formato a matrice.
Detta C la matrice dei costi, $\forall(i,j)$ vale $C[i,j] = C[j,i]$ in quanto la matrice C è simmetrica rispetto alla diagonale, in particolare se $i = j$ allora il valore della cella sarà $C[i][j] = 0$ in quanto rappresentante di un cappio.
Ho realizzato un semplice parser adibito alla lettura delle matrici dei costi, contenuto nella cartella *data*. Il parser prende l'indirizzo di un file *.csv* e restituisce un vector<vector<double>*>, dove il vettore interno è allocato nello heap per evitare che la copiatura del vettore non causi inutile inefficienza nell'algoritmo genetico implementato nel Laboratorio 2.
Le istanze disponibili nella cartella *data* sono:

$\#città$	$file$	obj_{opt}
5	data5.csv	19
15	data15.csv	291
17	data17.csv	2085
26	data26.csv	937
29	data29.csv	2020
42	data42.csv	699
48	data48.csv	33551
76	data76.csv	108137

La maggior parte delle istanze sono state prese da questi due siti web:

- TSP Data for the Traveling Salesperson Problem
- TSPLIB Symmetric Traveling Salesman Problem Instances.

Nel sito web l'obj_{opt} dell'istanza da 48 città viene riportata a 33523, penso sia sbagliata in quanto il modello concludendo la sua esecuzione senza raggiungere il tempo limite riporta una *obj* pari a 33551, si è deciso di prendere quest'ultimo valore come *obj* ottimo.

2 Laboratorio 1

Il Laboratorio 1 si poneva come obiettivo quello di implementare un modello di programmazione lineare con l'API Cplex C++ per il problema del TSP.

Definizioni

Il modello e la sua implementazione si basa sulle seguenti definizioni:

- n: numero di città;
- N: insieme delle città con $N = \{0, 1, .., n-1\}$;
- A: insieme degli archi tra le diverse città con $A = \{(i,j) | i \neq j, \forall i, j \in N\}$;
- O: nodo arbitrario di partenza, $O = 0$.

Per la realizzazione del Modello LP ho deciso di utilizzare alcuni costrutti visti durante i laboratori.

Struttura Map

Ho deciso di utilizzare un vector<vector<int>> allo scopo di rendere più agevole il recupero dell'indice corrispondente a ciascuna variabile. Il vettore è stato inizializzato completamente al valore -1.
Dato che non tutte le celle dovevano essere inizializzate, ad esempio quelle sulla diagonale rappresentanti i cappi, si è lasciato il valore -1.
Questo avrebbe permesso eventualmente di causare, in fase di esecuzione, un *runtime error* nel caso in cui si fosse utilizzata una cella non utilizzabile. La struttura è stata utilizzata per mappare gli indici delle viaribili x e y rispettivamente usando la struttura map_x e map_y create come spiegato precedentemente.

Variabili e constraint

Le variabili utilizzate sono state le variabili $x_{ij} \in \mathbb{R}_+ \; \forall (i,j) \in A$ rappresentanti la capacità di trasporto dell'arco $(i,j) \in A$ con $i, j \in N$.
Mentre le variabili $y_{ij} \in \{0, 1\}$ con $\forall i, j \in N$ sono state utilizzate per dire se l'arco $(i,j) \in A$ trasporta qualcosa con $i, j \in N$. Per quanto riguarda i constraint sono stati implementati, come visto nei laboratori precedenti, allocando un vettore per gli indici ed uno per i coefficienti.

Time Limit

Dato che l'esecuzione delle istanze più grandi richiedeva una grande mole di tempo, è stato inserito un time limit per arrestare l'esecuzione dopo una certa quantità di secondi, fissata a 3 secondi. È possibile definire tale parametro passandolo come argomento al momento della chiamata.

Risultati

Nella seguente tabella vengono riportati i risultati forniti dal modello senza limiti di tempo, questi sono la media di 10 esecuzioni aventi gli stessi parametri, come si può vedere ogni soluzione restituita è ottima.

n	obj_{opt}	obj_{mod}	Δ_{time}	$nvars$
5	19	19	0.004091	36
15	291	291	0.059943	406
17	2085	2085	0.372206	528
26	937	937	0.610495	1275
29	2020	2020	1.818207	1596
42	699	699	3.149250	3403
48	33551	33551	18.40613	4465
76	108137	108137	156.3707	11325

Nella tabella che segue, vengono riportati i valori medi di 10 esecuzioni indipendenti in termini di funzione obiettivo e tempo di esecuzione per ciascuna istanza, è inoltre stato imposto un tempo limite pari a 3 secondi.

n	obj_{opt}	obj_{mod}	Δ_{time}	$nvars$
5	19	19	0.004633	36
15	291	291	0.059322	406
17	2085	2085	0.361833	528
26	937	937	0.599837	1275
29	2020	2020	1.845714	1596
42	699	699	3.005812	3403
48	33551	38153	3.010204	4465
76	108137	145321	3.029727	11325

Come si può vedere le ultime due istanze hanno esaurito il tempo a loro disposizione, hanno quindi restituito la miglior soluzione trovata dall'algoritmo fino a quel momento.

3 Laboratorio 2

Il Laboratorio 2 si pone l'obiettivo di sviluppare una meta-euristica per risolvere il problema del TSP, questa deve essere in grado di trovare una soluzione molto vicina a quella ottima in tempi brevi.
Chiaramente il compromesso si basa sul fatto che per definizione una meta-euristica non restituisce sempre la soluzione ottima, bensì la miglior soluzione possibile trovata durante l'esecuzione. Come detto nella parte del Laboratorio 1 si può procedere ad imporre un tempo limite di esecuzione al modello, questo se trova delle soluzioni valide restituirà la migliore tra queste.
Utilizzare una meta-euristica risulta però essere un approccio preferibile in quanto permette di trovare una soluzione mediamente migliore ed in modo più efficiente.

Algoritmo genetico

La meta-euristica scelta è quella dell'algoritmo genetico, di cui in generale si descrivono brevemente le parti principali:

1. **Popolazione iniziale**: si crea un insieme P di individui eseguendo lo shuffle randomico del seguente vettore $< 0, 1, \ldots, n-1 >$;

2. **Local Search**: su un sottoinsieme $L \subseteq P$ di individui viene eseguita una Local Search con l'obiettivo di migliorare tale popolazione;

3. **Selezione**: durante questa operazione gli individui vengono ordinati in modo decrescente, dall'individuo peggiore al migliore in base alla funzione di fitness. Fatto questo si va a creare una distribuzione di probabilità discreta in modo da permettere anche agli individui peggiori di poter essere scelti. Viene utilizzata questa distribuzione per scegliere un sottoinsieme $S \subseteq P$ di individui con cui procedere al crossover;

4. **Crossover**: attraverso l'algoritmo di crossover $CX2$, genero due figli sempre validi che entrano a far parte dell'insieme F, a partire da ciascuna coppia di genitori, scelta casualmente dall'insieme S;

5. **Mutazione**: effettuo l'unione tra $P = S \cup F$, si procede su un sottoinsieme $M \subseteq P$ ad invertire casualmente due città in modo da mantenere la diversità;

6. Ricomincio dal punto 2 se non è già stato raggiunto il tempo limite di esecuzione o il tetto massimo di iterazioni fallimentari consecutive.

Implementazione

L'algoritmo è stato organizzato in diversi file, come segue:

- classe chromosome: rappresenta un individuo, ossia una soluzione del problema del TSP, sono implementati la Local Search e l'inversione delle città, oltre che la funzione di Fitness e di stampa. In particolare il circuito delle città è stato implementato con un vettore di interi, rappresentante la sequenza di visita delle città;
- genetic: contiene tutte le funzioni utilizzate dall'algoritmo genetico;
- utility: contiene una funzione utilizzata per fare lo shuffle con una probabilità distribuità uniforme;
- main: contiene la funzione generale dell'algoritmo genetico.

Local Search 2-Opt

La funzione *LS2opt* contenuta nel file *genetic.cpp* si occupa di lanciare su un sottoinsieme $L \subseteq P$ una Local Search per migliorare tali individui. Questa è stata implementata nella classe chromosome e si occupa di vagliare il vicinato dell'individuo. Viene generato il vettore T utilizzando la funzione contenuta in *utility.cpp*, che esegue uno shuffle randomico del vettore $< 0, 1, \ldots, n-1 >$ e ne restituisce i primi k valori contenuti.
L'algoritmo $\forall i, j \in T$ scambia le città in posizione i e j solo se l'inversione migliora la fit di tale individuo, se questo accade allora si ricomincia utilizzando un nuovo shuffle di indici. La funzione randomica basata sullo shuffle evita di estrarre nuovamente indici già estratti.

Selezione

La selezione è basata sulla **linear ranking**, la popolazione P viene ordinata in modo decrescente in base alla funzione di fitness assegnando a ciascun individuo $x_i \in P$ di posizione θ_i un peso come segue:

$$Pr(x_i) = \frac{2\theta_i}{|P|(|P|+1)}$$

Viene quindi creato un vettore di pesi avente in posizione corrispondente a quello della popolazione il peso dell'individuo. Per effettuare le estrazioni con probabilità discreta è stata utilizzata la struttura *discrete_ distribution* della libreria standard. Quando un individuo viene estratto si procede ad azzerare il suo peso in modo che non possa più essere scelto, quindi si procede a ricreare la distribuzione di probabilità discreta. Si ripete questa operazione finché non vengono estratti s elementi.

Crossover CX2

L'algoritmo di Crossover $CX2$ è stato scelto per la sua capacità di creare coppie di figli sempre valide, come spiegato nel seguente paper[1].

Funzione print

Quando viene fatta la stampa di un individuo si ottiene una cosa simile alla seguente:

291 :13 11 2 6 4 8 14 1 12 0 10 3 5 7 9 . 1

Dove viene riportata la funzione obiettivo, il circuito e dopo il punto finale una flag 0-1 che indica la validità della soluzione.

Parametri

L'algoritmo genetico implementato necessita dei seguenti parametri:

1. **n**: numero dell'istanza, viene caricato il vettore dei costi del file contenuto in *data/data**n**.csv*;
2. **p**: numero di individui da generare casualmente come popolazione iniziale;
3. **s**: numero di individui da estrarre durante la fase di selezione;
4. **l**: numero di individui su cui invocare la Local Search;
5. **k**: quantità di città su cui provare lo scambio durante la Local Search;
6. **m**: numero di individui su cui eseguire una mutazione casuale;
7. **f**: numero di iterazione fallimentari entro cui fermare l'algoritmo a partire dall'ultima esecuzione migliorativa;
8. **t**: tempo limite in secondi entro cui fermare l'algoritmo.

Insieme di combinazioni di parametri

Dato n, ossia il numero dell'istanza, si è deciso di fissare i parametri come segue:

- $p = 2n$;
- $s = n$;
- $t = 3$.

Per i parametri restanti si è proceduto con due approcci differenti:

- Grid Search;
- Random Search.

Grid Search

Dati gli intervalli $\alpha \in [a, \ldots, b]$, $\beta \in [c, \ldots, d]$, $\gamma \in [x, \ldots, y]$.
Dato un insieme di combinazioni di parametri, definite come segue:

$$\{(n, p, s, l, k, m) \mid \forall l \in \alpha,\ k \in \beta,\ m \in \gamma\}$$

Per ciascuna combinazione (n, p, s, l, k, m) si è proceduto ad eseguire 10 chiamate dell'algoritmo genetico, incrementando via via il numero di iterazioni f, come segue:

$$(n, p, s, l, k, m, f, t) \text{ con } \forall f \in [1, \ldots, 50]$$

Ci si arrestava non appena l'errore relativo medio $\epsilon_{r\%}$ calcolato, utilizzando la fit media e quella ottima, non scendeva sotto il 5%, si procedeva a scrivere in un file tale combinazione di parametri e si procedeva con quella successiva.

Random Search

Differisce dalla Grid Search solo per il fatto che gli intervalli sono generati casualmente, inoltre il numero di combinazioni viene limitato in quanto con l'approccio precedente i tempi di esecuzione risultavano proibitivi sulle istanze più grandi. Si procedeva a salvare tale combinazione di parametri solo se produceva un errore relativo $\epsilon_{r\%} \leq 5\%$.

Scelta combinazione di parametri

Dopo aver trovato l'insieme delle combinazioni di parametri con un errore relativo $\epsilon_{r\%} \leq 5\%$, si è proceduto a scegliere quello con il tempo di esecuzione medio minore. Di questo si è aumentato il numero di iterazioni fino a che non si è raggiunto un $\epsilon_{r\%} \leq 2\%$, di seguito vengono riportati i parametri così ottenuti.

n	p	s	l	k	m	f	t
5	10.0	5.0	1.0	4.0	1.0	1.0	3.0
15	30.0	15.0	6.0	14.0	6.0	10.0	3.0
17	34.0	17.0	7.0	16.0	7.0	10.0	3.0
26	52.0	26.0	12.0	25.0	12.0	10.0	3.0
29	58.0	29.0	13.0	28.0	13.0	10.0	3.0
42	84.0	42.0	20.0	41.0	20.0	10.0	3.0
48	96.0	48.0	23.0	47.0	23.0	10.0	3.0
76	152.0	76.0	31.0	75.0	31.0	10.0	3.0

I parametri così ottenuti sollevano alcune osservazioni:

- k: esegue la Local Search 2-opt completa, evidentemente è più conveniente che scambiare un numero minore di indici o aumentare le iterazione fallimentari;
- l e m: eguale numero di individui, sarà il bilanciamento migliore;
- proporzionalità tra s e k,l.

Risultati

Sono stati ottenuti i seguenti risultati dopo aver eseguito l'algoritmo 100 volte per ciascuna istanza. Si nota come in tutte le istanze i tempi di esecuzione siano molto contenuti. La obj_{mean} si avvicina molto al valore ottimo considerando l'errore relativo percentuale $\epsilon_{r\%}$, inoltre anche nel caso pessimo la obj_{worst} non si discosta molto dalla soluzione media.

- obj_{opt}: valore fit della soluzione ottima;
- obj_{best}: valore di fit della miglior soluzione trovata sulle 100 iterazioni;
- obj_{mean}: il valore medio della fit delle 100 iterazioni;
- obj_{worst}: valore di fit della peggior soluzione trovata sulle 100 iterazioni;
- SD: deviazione standard;
- Δ_{time}: tempo di esecuzione;
- ϵ_a: errore assoluto della soluzione media rispetto la soluzione ottima;
- $\epsilon_{r\%}$: errore relativo percentuale della soluzione media rispetto la soluzione ottima.

n	obj_{opt}	obj_{best}	obj_{mean}	obj_{worst}	SD	Δ_{time}	ϵ_a	$\epsilon_{r\%}$
5	19	19	19	19	0.000000	0.000198	0	0.0000
15	291	291	291	291	0.000000	0.013790	0	0.0000
17	2085	2085	2087	2095	2.449490	0.023303	2	0.0959
26	937	937	940	960	6.708204	0.081805	3	0.3201
29	2020	2020	2032	2069	12.12435	0.095173	12	0.5940
42	699	699	708	724	6.244998	0.326466	9	1.2875
48	33551	33551	34050	34208	182.0357	0.452080	499	1.4872
76	108137	108809	110198	112010	625.3702	2.080291	2061	1.9059

Si può vedere come l'errore relativo percentuale $\epsilon_{r\%}$ si attesti sempre al di sotto del 2%, questo conferma come l'euristica proposta sia capace di fornire soluzioni molto vicine all'ottimo in tempi contenuti con una accurata scelta di parametri.

4 Conclusioni

Confrontiamo ora i risultati ottenuti dal modello del Laboratorio 1 e dall'algoritmo genetico del Laboratorio 2:

n	Δ_{tmod}	Δ_{tga}
5	0.004091	0.000198
15	0.059943	0.013790
17	0.372206	0.023303
26	0.610495	0.081805
29	1.818207	0.09517
42	3.149250	0.326466
48	18.40613	0.452080
76	156.3707	2.080291

Tempi modello non limitato e euristica a confronto.

Nella versione non limitata il modello produce risultati in tempi molto elevati. Per istanze grandi il modello risulta quindi inutilizzabile per via dei suoi tempi di esecuzione. Si noti di seguito come imponendo lo stesso tempo limite di 3 secondi ad entrambi, l'euristica restituisca in tempi sempre inferiori buone soluzioni.

n	obj_{opt}	obj_{mod}	obj_{ga}	Δ_{tmod}	Δ_{tga}
5	19	19	19	0.004633	0.000198
15	291	291	291	0.059322	0.013790
17	2085	2085	2087	0.361833	0.023303
26	937	937	940	0.599837	0.081805
29	2020	2020	2032	1.845714	0.09517
42	699	699	708	3.005812	0.326466
48	33551	38153	34050	3.010204	0.452080
76	108137	145321	110198	3.029727	2.080291

Confronto tra modello e euristica con tempo limite a 3 secondi.

Si nota come i tempi di esecuzione dell'algoritmo genetico siano nettamente inferiori rispetto a quelli del modello, in particolare nelle istanze in cui quest'ultimo non sia limitato dal tempo limite dei 3 secondi.
L'algoritmo genetico riesce a fornire soluzioni molto vicine all'ottimo, in particolare nel caso delle ultime due istanze il modello raggiunge il tempo limite e le soluzioni dell'algoritmo genetico sono di gran lunga migliori.
In conclusione l'algoritmo genetico ha dimostrato di essere una buona metaeuristica, con una accurata scelta dei parametri si possono ottenere buoni risultati in tempi ridotti favorendo quindi il suo utilizzo al posto di metodi esatti limitati.

5 Compilazione esecuzione

Laboratorio 1

Per compilare il file aprire il terminale all'interno della cartella LAB1 ed eseguire il comando:

make

Una volta compilato l'eseguibile sarà denominato **main**, per eseguire basterà scrivere il seguente comando:

./main n t

Sostituendo al posto di n il numero dell'istanza e al posto di t il tempo limite in secondi.

Laboratorio 2

Per compilare il file aprire il terminale all'interno della cartella LAB2 ed eseguire il comando:

g++ -std=c++11 main.cpp

Una volta compilato sarà possibile eseguirlo scrivendo il seguente comando:

./a.out n p s l k m f t

si dovrà sostituire al posto delle lettere i parametri scelti.

6 Specifiche Hardware

Dell-XPS-13-7390
Intel Core i7-10510U CPU @ 1.80GHz 8 core

References

[1] Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, Showkat Gani *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator*. Hindawi 2017.