Relazione di laboratorio

Corso di Algoritmi Avanzati
Laurea Magistrale in Informatica
A.A. 2019-2020

Magarotto Francesco - 1236594
Muraro Enrico - 1238899
Piva Giulio - 1242455

1 Introduzione

L'esercizio di laboratorio consiste nel implementare e valutare tre algoritmi per il problema del *travel salesman problem* (TSP). Gli algoritmi sono:

- Held-Karp;
- L'euristica costruttiva Nearest Neighbor;
- 2 approssimato;

Il linguaggio di programmazione scelto dal nostro gruppo è Java.

1.1 Esecuzione del programma

Gli algoritmi sono stati sviluppati come progetto Maven. All'interno della cartella é presente la versione portable di Maven, pertanto non è necessario averlo installato. É richiesto almeno il JDK 11 installato nel sistema. Per eseguire i tre algoritmi utilizzare i seguenti comandi:
Linux:

```
./mvnw install
./mvnw exec:java
```

Windows:

```
mvnw.cmd install
mvnw.cmd exec:java
```

È richiesto un tempo di esecuzione in relazione al tempo di timeout impostato per l'algoritmo di Held-Karp. Il valore di default del timeout è di 5 minuti. L'esecuzione del main genera automaticamente dei file csv nella directory del progetto contenenti i tempi registrati per tutti e tre gli algoritmi eseguiti su tutti i file del dataset.

1.2 Strutture dati utilizzate

Per rappresentare il grafo abbiamo utilizzato una matrice di adiacenza visto che i sono grafi completi e la matrice viene quindi riempita senza sprechi di memoria.

Nell'algoritmo 2 Approssimato abbiamo fatto uso di un HashMap per registrare la visita in preordine del grafo.

La classe Heap è una nostra implementazione di MinHeap. L'albero binario è rappresentato da un array di interi, i valori all'interno dell'array corrispondono ai nodi presenti nel grafo. Il confronto tra i nodi per determinare il più piccolo è effettuato tramite un Comparator passato alla creazione dello Heap, questo per avere un'implementazione di Heap indipendente dal modo in cui viene utilizzato da uno specifico algoritmo.

1.3 Lettura di un grafo da file

Per caricare un grafo in memoria, abbiamo implementato una classe GraphReader, che si occupa della lettura del file tramite la libreria *nio* di Java. Inoltre effettua le conversioni di distanza necessarie sia per i file di tipo GEO che EUC_2D e ritorna direttamente la matrice di adiacenza del grafo.

1.4 Implementazione di Prim (da sistemare)

Prim necessita di due campi aggiuntivi "key" e "parent" per ogni nodo, key contiene l'attuale peso più piccolo per raggiungere il nodo, e parent contiene il nodo da cui si arriva. Questi campi sono stati implementati con due HashMap come per la lista di adiacenza.

L'Heap viene inizializzato con tutti i nodi del grafo, e nel costruttore viene passata un'istanza di NodeComparator che utilizza il campo key per confrontare i nodi.

Il Set Q contiene tutti i nodi che non sono ancora nel MST, ed è utile per vedere il nodo che stiamo valutando non è nel MST in tempo costante al posto di cercarlo nello Heap in tempo lineare.

Il costo totale del MST viene infine calcolato semplicemente scorrendo tutti i nodi e sommando i loro valori di key.

1.5 Implementazione di Held-Karp

L'algoritmo di Held-Karp necessita di due strutture di supporto, una per salvare le distanze minime già calcolate e una per ricordare il nodo precedente in modo da poter ricostruire il percoso trovato. Visto che entrambi sono identificati da due valori (v,S) dove 'v' è un nodo e 'S' un insieme di nodi abbiamo deciso di implementare entrambe con ArrayList di HashMap. Ogni posizione dell'ArrayList corrisponde a un nodo e contiene una mappa che ha come chiave un Set di nodi. In questo modo con due accessi costanti possiamo trovare il valore desiderato.

Il timeout di Held-Karp è controllato da un flag booleano, quando viene impostato il valore del flag a 'true' il ciclo che cerca il minimo viene interrotto e l'algoritmo ritorna la soluzione migliore trovata fino a quel momento.

Quando eseguiamo l'algoritmo facciamo quindi partire un thread usando ScheduledExecutorService, che dopo 5 minuti imposta il flag a 'true' terminando l'esecuzione.

1.6 Implementazione dell'euristica nearest-neighbor

L'Euristica nearest-neighbor consiste nel trovare il prossimo vertice, non ancora inserito nel circuito, a distanza minima dal nodo corrente. Una volta visitati tutti i nodi ripetendo questa operazione si otterá un percordo 2-approssimato per il problema del TSP. Questa euristica é di facile implementazione, ma allo stesso tempo molto efficace.

1.7 Implementazione dell'algoritmo 2-approssimato

L'algoritmo 2-approssimato é basato su una tecnica molto semplice: Si trova un MST con l'algoritmo di Prim e poi si visita l'albero in ordine prefisso. Nel nostro caso, Prim restituisce il MST sotto forma di HashMap, dove la chiave é un nodo e il value la lista dei suoi figli. A questo punto l'albero viene visitato in ordine prefisso dal metodo Preorder, restituendo un percorso approssimato per il problema del TSP.

2 Risultati ottenuti

Istanza	Held-Karp			Euristica Nearest			2 Approssimazione		
	Soluzione	Tempo (s)	Errore (%)	Soluzione	Tempo (s)	Errore (%)	Soluzione	Tempo (s)	Errore (%)
pcb442.tsp	229601.0	300.063	352.0	61979	9	22.0	73926	35	46.0
berlin52.tsp	18883.0	300.007	150.0	8980	0.0	19.0	10402	0.0	38.0
eil51.tsp	1074.0	409.244	152.0	511	0.0	20.0	605	0.0	42.0
ulysses16.tsp	6859.0	18.535	0.0	9988	0.0	46.0	7788	0.0	14.0
kroD100.tsp	148541.0	300.007	598.0	26947	0.0	27.0	28599	1	34.0
gr202.tsp	72655.0	300.009	81.0	49336	2	23.0	52615	3	31.0
ch150.tsp	48239.0	300.005	639.0	8191	1	25.0	9126	1	40.0
gr229.tsp	182845.0	300.009	36.0	162430	1	21.0	179335	2	33.0
d493.tsp	112899.0	300.01	223.0	41665	1	19.0	45623	9	30.0
dsj1000.tsp	5.52276003E8	300.011	2860.0	24630960	6	32.0	25526005	35	37.0
kroA100.tsp	172858.0	300.006	712.0	27807	0.0	31.0	30516	0.0	43.0
ulysses22.tsp	8194.0	300.013	17.0	10586	0.0	51.0	8308	0.0	18.0
burma14.tsp	3323.0	1.059	0.0	4048	0.0	22.0	4003	0.0	20.0

Come possiamo vedere dalla tabella, nell'algoritmo di Held-Karp solo poche istanze **non** sono andate in *timeout*. Infatti, i grafi con tempo di esecuzione maggiore di circa 300000 secondi, sono state interrotte. Pertanto l'83.3% delle esecuzioni dell'algoritmo di Held-Karp ritorna la soluzione calcolata fino a quel momento, con un conseguente tasso di errore molto alto, anche del 2860% come nel caso dell'istanza `dsj1000.tsp`.

2 Approssimazione				
Soluzione	**Tempo (s)**	**Errore (%)**	**Soluzione ottima**	**ρ(p)**
73926	35	46.0	50778	0,69
10402	0.0	38.0	7542	0,73
605	0.0	42.0	426	0,70
7788	0.0	14.0	6859	0,88
28599	1	34.0	21294	0,74
52615	3	31.0	40160	0,76
9126	1	40.0	6528	0,72
179335	2	33.0	134602	0,75
45623	9	30.0	35002	0,77
25526005	35	37.0	18659688	0,73
30516	0.0	43.0	21282	0,70
8308	0.0	18.0	7013	0,84
4003	0.0	20.0	3323	0,83

Questa tabella invece mostra l'approssimazione calcolata dall'algoritmo è 2 approssimata. Infatti, $\frac{Soluzione}{Solzione\ ottima} = \rho(p) \leq 2$

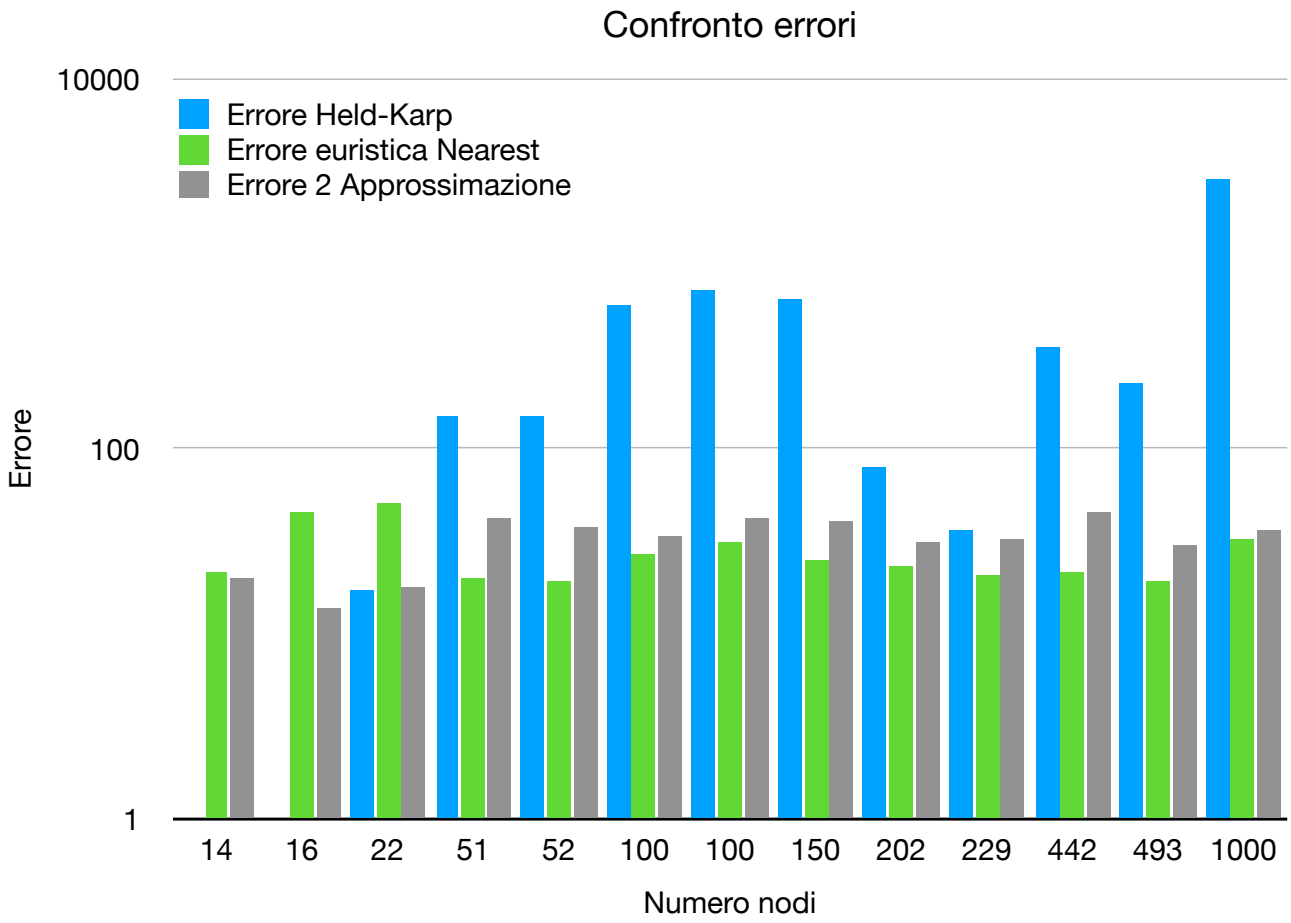

Figura 1: Istogramma che mette in relazione gli errori dei tre algoritmi utilizzando la scala logaritmica. Come era prevedibile dalla tabella sopra, l'errore maggiore è presente nell'algoritmo di Held-Karp, che in questi pochi esempi non è sempre crescente poiché il numero di istanze è esiguo. Certamente con un numero di maggiore di nodi, avremmo due possibili scenari: mantenendo lo stesso tempo massimo di esecuzione, l'errore sarebbe molto alto, mentre aumentando il tempo di esecuzione, lo stack verrebbe riempito delle chiamate ricorsive, portando ad un'eccezione runtime OutOfMemory.

2.0.1 Tempo di timeout Held-Karp

Abbiamo provato varie esecuzioni dell'algoritmo di Held-Karp per verificare come varia l'errore in relazione al raddoppio del tempo prima del timeout.

Istanza	Held-Karp			
	Soluzione	Tempo (s)	Errore (%)	$\text{Errore}_{10\text{ min}} - \text{Errore}_{5\text{ min}}$
berlin52.tsp	18866,0	600,007	150,0	0
burma14.tsp	3323,0	1,186	0,0	0
ch150.tsp	47774,0	600,001	632,0	-7
d493.tsp	112867,0	600,003	222,0	-1
dsj1000.tsp	5,52276003E8	600,011	2860,0	0
eil51.tsp	1041,0	600,014	144,0	-8
gr202.tsp	72655,0	600,015	81,0	0
gr229.tsp	182845,0	600,014	36,0	0
kroA100.tsp	171456,0	600,008	706,0	-6
kroD100.tsp	147451,0	600,002	592,0	-6
pcb442.tsp	229601,0	600,012	352,0	0
ulysses16.tsp	6859,0	15,295	0,0	0
ulysses22.tsp	8110,0	600,002	16,0	-1

Figura 2: Tabella che mostra come l'errore sia variato in relazione all'aumento del tempo di esecuzione dell'algoritmo. In particolare, aumentando il tempo prima del timeout da 5 a 10 minuti, solo i grafi di dimensione compresa tra 20 e 100 hanno avuto un beneficio. I grafi con un numero di nodi superiore a 200 **non** hanno avuto alcun beneficio dall'aumento del tempo di esecuzione.

3 Conclusioni

L'algoritmo migliore tra i tre è l'euristica Nearest che presenta un tasso di errore minore, avvicinandosi di più alla soluzione ottima. La soluzione dell'euristica Nearest Neighbor è $O(log(n))$ - approssimata a TSP, quando la disuguaglianza triangolare è rispettata. La complessità Held-Karp con table filling ha complessità $O(n^2 * 2^n)$, che è esponenziale e interropendo l'esecuzione a 5 minuti non permette .