Relazione di Algoritmi Avanzati

Nicola Carlesso - matricola 1237782
Federico Brian - matricola 1243422

A.A. 2019/2020

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Il presente documento descrive le scelte architetturali ed implementative del secondo elaborato di laboratorio del corso di Algoritmi Avanzati. Di seguito, verrà offerta una panoramica sul lavoro svolto dagli studenti Nicola Carlesso e Federico Brian, riguardante lo studio ed il confronto dei tre diversi algoritmi visti a lezione per la risoluzione del problema *Travelling Salesman Problem*[1]:

* l'algoritmo di Held e Karp (in seguito: `HeldKarp`) che ritorna la soluzione esatta del problema TSP con complessità $\mathcal{O}(n^2 2^n)$;
* un algoritmo ottenuto da un' "euristica costruttiva", in particolar modo `CheapestInsertion`, ottenendo un algoritmo 2-approssimato per TSP;
* l'algoritmo `TriangeTSP` che risolve il problema TSP con un algoritmo 2-approssimato attraverso la costruzione di un MST con l'assunzione che il grafo rispetti la disuguaglianza triangolare.

Infine, verranno esposti ed adeguatamente discussi i risultati ottenuti.

[1] d'ora in poi TSP

2 Scelta del linguaggio di programmazione

Per lo svolgimento di questo *assignment* è stato scelto, come linguaggio di programmazione Java nella sua versione 8. La scelta è derivata, principalmente, da due fattori:

* è stato sia studiato durante il percorso di laurea triennale, sia approfondito autonomamente da entrambi;
* in Java, è possibile utilizzare riferimenti ad oggetti piuttosto che oggetti stessi. Questo ha permesso un'implementazione degli algoritmi che si potrebbe definire "accademica", perché coerente con la complessità dichiarata e semanticamente vicina allo pseudocodice visto a lezione.

Questo ultimo punto ha bisogno di essere sviluppato ulteriormente per risultare chiaro. In una prima implementazione degli algoritmi , utilizzando l'approccio *object-oriented* senza l'utilizzo di riferimenti, ci siamo accorti che il codice aggiungeva complessità, anche abbastanza pesanti, rispetto allo pseudocodice illustrato a lezione. Questo accadeva perché inizialmente sono stati utilizzati costruttori di copia profonda che, oltre a raddoppiare l'utilizzo di memoria, aggiungevano una complessità d'ordine rispetto al numero dei lati, al numero dei nodi oppure rispetto ad entrambe.

Ad esempio, in una prima implementazione dell'algoritmo `NaiveKruskal`, ad ogni iterazione del ciclo principale, veniva creato un nuovo grafo, copiando il grafo ottenuto aggiungendo iterativamente un lato alla volta. Il costruttore di copia profonda provvedeva a creare due nuove liste: una di nodi ed una di lati, entrambi aventi le medesime caratteristiche delle liste del grafo da cui sono stati copiati.

Questo ci ha portato a riflettere sul significato dello pseudocodice dei tre diversi algoritmi e ci ha guidato verso uno sviluppo di un codice che:

* mantenesse la caratteristica di facile leggibilità propria della programmazione ad oggetti;
* fosse coerente con le complessità dichiarate a lezione.

Questi obiettivi sono stati raggiunti agendo su riferimenti di oggetti piuttosto che su oggetti stessi.

3 Scelte implementative

Come specificato nel precedente paragrafo, nell'implementazione dei tre algoritmi si è cercato di creare meno oggetti possibile usando per lo più riferimenti. Questo ha permesso non solo un risparmio in termini di memoria ma anche di prestazioni: nell'implementazione abbiamo infatti cercato di creare strutture dati che risparmiassero quanta più memoria possibile, dato che l'algoritmo *HeldKarp*, oltre ad essere computazionalmente oneroso, richiede anche l'utilizzo di molta memoria. Difatti, nell'eseguire l'esecuzione del `main`, è necessario richiedere l'utilizzo di più memoria RAM per il corretto funzionamento dell'algoritmo attraverso il flag `-Xmx8192m`.
Benché il codice sia stato adeguatamente commentato[2], di seguito è riportata una *summa* delle caratteristiche di ogni classe implementata che non compaiono nella documentazione, ripartita per `package`.

3.1 Modello

Le componenti del modello, vale a dire le classi presenti all'interno del `package` chiamato `lab2.model`, comprendono tutte le strutture dati utilizzate nella risoluzione dei tre problemi assegnati.

* `AdjacentMatrix`: matrice di adiacenza usata per rappresentare i grafi. Tale matrice si presenta come una matrice triangolare inferiore, contenendo `n - 1` array di lunghezza crescente. Tale scelta è stata fatta per risparmiare memoria evitando di costruire una matrice quadrata. Tale classe presenta i metodi standard `get(u,v)` e `set(u,v)`, per ottenere ed impostare il peso del lato (u,v), ed un motodo `getMinAdjacentVertexWeightIndex(v)` per ottenere il lato con peso minore che ha per estremo il vertice v;

* `Graph`: contiene esclusivamente una matrice di adiacenza, dato che tutte le informazioni necessarie dei nodi (i quali iniziano ad essere contati da 0) e dei lati possono essere ottenute analizzando la matrice di adiacenza;

* `Node`: classe non utilizzata direttamente da *Graph*, ma usata per costruire il MST (Minimum Spanning Tree) per l'algoritmo di 2-approssimazione richiesto. Essa dunque contiene solo l'*ID*, un riferimento al nodo padre e una lista di riferimenti ai nodi figli;

* `Edge`: come la classe *Node*, anche *Edge* viene utilizzata solo per costruire l'MST attraverso l'algoritmo *Kruskal* implementato nello scorso assignement.

* `DisjointSet`: questa struttura dati gestisce partizioni di oggetti, rappresentati con un numero intero che li identifica. Ogni oggetto può stare in una sola delle partizioni degli insiemi disgiunti presenti. La struttura dati è utilizzata all'interno dell'algoritmo `Kruskal`.

* `TSP`: contiene tutti gli algoritmi richiesti dagli assignment, più le funzioni ausiliarie per il corretto funzionamento di questi ultimi, in particolar modo è importante menzionare

  - `deepCopyWithoutV`:
  - `getResults`:
  - `preorder`: metodo utilizzato per l'algoritmo di 2-approssimazione *Tree_TSP*, per ottenere una lista pre-ordinata dei nodi del MST ottenuto dal metodo `Kruskal`.

  TSP contiene inoltre tre campi dati per il corretto funzionamento di *HeldKarp*

[2] come si può vedere dal `Javadoc`, automaticamente generato e accessibile all'interno della cartella `JavaLab2/doc/`, aprendo il file `index.html` con il browser preferito

- `d`:
- `pi`:
- `w`:

3.2 Algoritmi

Il package `lab2.algorithm` contiene un'unica classe, `TSP`, che permette di risolvere il problema TSP utilizzando gli algoritmi `HeldKarp`, `CheapestInsertion` e `Tree_TSP`.

* `HeldKarp`: in combinazione col metodo `HeldKarpCore` non presenta grandi differenze rispetto allo pseudocodice fornito a lezione, con la sola distinzione di un `if` necessario per la gestione dei *Thread* nel caso in cui la computazione dovesse richiedere più di due minuti;

* `CheapestInsertion`: l'algoritmo fa uso del metodo `getMinAdjacentVertexWeightIndex` per trovare il lato col peso minore che ha come estremo il nodo 0. Viene dunque creato un array di nodi che rappresenta il cammino per TSP ed un array coi nodi ancora non visitati. L'algoritmo dunque esegue un ciclo fino a quando il cammino non raggiunge lunghezza $n + 1$ ed trova per ogni nodo non visitato e per ogni lato presente nel cammino trovato fino a quel momento, il minore `minCost`, indicando che nodo k non visitato deve essere inserito nel cammino e in che posizione. L'algoritmo ha complessità $\mathcal{O}(n^3)$;

* `Tree_TSP`: l'algoritmo, attraverso l'algoritmo *Kruskal*, ottiene prima il MST sotto forma di *Node*, un nodo che possiede la lista di puntatori ai nodi figli, dopodiché ottiene la lista pre-oridinata dei nodi dell'albero ottenuto. L'algoritmo ha complessità $\mathcal{O}(m \lg n + n)$.

3.3 Main

Il package `lab1.main` contiene la classe `Main`, responsabile dell'esecuzione degli algoritmi.
All'interno vi sono tre funzioni:

* la funzione `Main` che fa partire il calcolo oppure il test del costo della soluzione per TSP secondo l'algoritmo desiderato;

* la funzione `compute` che si occupa di calcolare il costo della soluzione per TSP di ogni grafo presente nel dataset, di salvarlo in un file di testo e di proseguire al test dello stesso.
  Per utilizzare/testare i tre diversi algoritmi, inserire all'interno della funzione `compute` una tra le seguenti stringhe:
  - `HeldKarp` per l'algoritmo di Held e Karp;
  - `Heuristic` per l'algoritmo di 2-approssimazione che sfrutta l'euristica strutturale della disuguaglianza triangolare;
  - `2Approx` l'algoritmo che attraverso il calcolo del MST calcola una soluzione per TSP con 2-approssimazione.

* la funzione `test`, che esegue alcuni, banali, test sulle strutture dati o algoritmi utilizzati come *AdjacentMatrix* e *Kruskal*.

3.4 Test

Il package `lab2.test` contiene due classi:

* `TestTSP` il cui scopo è di testare la bontà delle soluzioni ritornate con i tre algoritmi sviluppati e di calcolarne l'eventuale errore relativo;

* `TestKruskal` che è servita per testare il funzionamento dell'algoritmo di *Kruskal*.

3.5 Documentazione

Il codice, opportunamente commentato, possiede una documentazione auto-generata con la funzionalità `javadoc`: la si può consultare accedendo alla directory `JavaProject/doc` ed aprendo il file `index.html` con il proprio browser preferito.

4 Risultati degli algoritmi

Questa sezione risponderà alla Domanda 1: verranno riportati sotto forma di tabella i risultati dei costi per il problema TSP dei grafi richiesti, il tempo di esecuzione di ogni algoritmo e l'errore relativo rispetto alla soluzione esatta.

4.1 Specifiche Hardware dei calcolatori utilizzati

Dato il considerevole divario di prestazioni ottenute dalle due diverse macchine, gli studenti hanno ritenuto opportuno riportare le differenti tempistiche impiegate per il calcolo dei costi degli MST.

Caratteristica	**PC di Nicola**	**PC di Federico**
Architettura	64 bit	64 bit
Nome processore	Intel i5-7300HQ	Intel i7-8750H
Numero core	4	6
Numero thread	4	12
Range velocità di clock [GHz]	2.50 - 3.50	2.20 - 4.10
Dimensione cache L1 [KiB]	256	384
Dimensione cache L2 [MiB]	1	1.5
Dimensione cache L3 [MiB]	6	9
Dimensione RAM [GiB]	8	31.2

Tabella 1: Specifiche dei calcolatori utilizzati
(Fonte: `http://intel.com`).

4.2 HeldKarp

L'algoritmo non presenta variazioni nell'implementazione rispetto all'algoritmo mostrato a lezione, dunque possiede una complessità di $\mathcal{O}(n^2 2^n)$.

4.3 CheapestInsertion

L'algoritmo opera con l'assunzione che il grafo dato rispetti la disuguaglianza triangolare, inserendo iterativamente un nodo k all'interno del circuito parziale per la soluzione a TSP per $n-2, n = |V|$ volte con V l'insieme dei nodi del grafo. In particolare, dato $k \notin C \subseteq V$, con C l'insieme dei nodi nel circuito parziale, e i nodi $u, v \in C$ t.c. $(u, v) \in P$, con P l'insieme dei lati nel circuito parziale, k viene scelto ed inserito nel circuito parziale tra i nodi u e v minimizzando: $w(u, k) + w(k, v) - w(u, v)$.
La complessità dell'algoritmo si può calcolare analizzando i tre ciclo `for` nell'algoritmo. Il ciclo più esterno viene eseguito $n-2$ volte. I due cicli più interni scorrono tutti i nodi $k \notin C$ e i lati $(u, v) \in P$. É possibile vedere come, ad ogni iterazione del ciclo `for` più esterno il numero di iterazioni dei due cicli più in interni, rispettivamente diminuiscano ed aumentino; in particolare il numero totale di iterazioni compiute dai tre cicli `for` è: $\sum_{i=2}^{n-1}(n-i)i = \frac{1}{6}(n^3 - 7n + 6)$. Dunque l'algoritmo `CheapestInsertion` possiede una complessità di $\mathcal{O}(n^3)$. Questo risultato risulta evidente analizzando i tempi di risoluzione dei vari grafi in Tabella **??**, il quale aumenta esponenzialmente all'aumentare della taglia del grafo, passando dai 0.004 s per un grafo da 100 nodi, a 7.5 s per un grafo da 1000 nodi.
L'algoritmo è 2-approssimato per TSP, anche se nei grafi forniti l'errore relativo massimo è del 22%, dimostrando dunque di essere molto efficacie.

N.	**Name Graph**	**TSP cost**	**Time (s)**	**Error (%)**
1	berlin52	9004	0.0109721	19.38
2	burma14	3588	2.739E-4	7.97
3	ch150	7998	0.0745751	22.52
4	d493	39969	0.5705098	14.19
5	dsj1000	22291165	7.8855598	19.46
6	eil51	494	0.009735	15.96
7	gr202	46480	0.0874466	15.74
8	gr229	153896	0.0529914	14.33
9	kroA100	24942	0.0040815	17.20
10	kroD100	25204	0.0040534	18.36
11	pcb442	60834	0.3931202	19.80
12	ulysses16	7368	2.83E-5	7.42
13	ulysses22	7709	9.53E-5	9.92

Tabella 2: Risultati dell'algoritmo `CheapestInsertion`

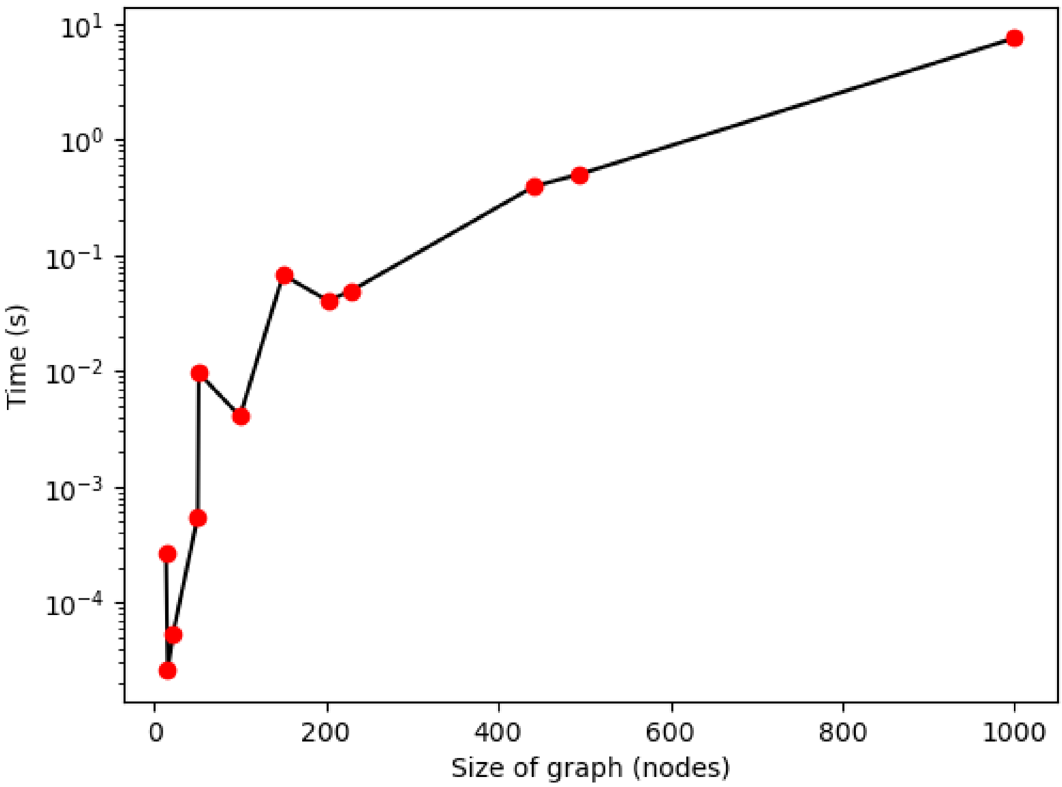

Figura 1: Performance dell'algoritmo `CheapestInsertion`.

4.4 Tree_TSP

L'algoritmo crea il circuito per TSP attraverso la lista preordinata dei nodi del MST ottenuto dall'algoritmo `Kruskal`. Tale algoritmo costruisce un albero la cui radice è un'istanza della classe `Node`, che non ha niente a che fare con la classe `Graph` ed è utilizzata esclusivamente dall'algoritmo `Kruskal`. Un discorso analogo vale per la classe `Edge` usata solamente dall'algoritmo `Kruskal` per ottenere una lista ordinata per peso dei lati del grafo.
Dato $n = |V|$ e $m = |E|$, l'algoritmo `Tree_TSP`, dato utilizza `Kruskal`, ha una complessità $\mathcal{O}(m \log n)$.
L'algoritmo è 2-approssimazione per TSP, e per i grafi forniti ottiene risultati con un errore relativo al massimo del 45%.

N.	**Name Graph**	**TSP cost**	**Time (s)**	**Error (%)**
1	berlin52	10402	0.0083738	37.92
2	burma14	4003	2.059E-4	20.46
3	ch150	9126	0.0253737	39.80
4	d493	45300	0.1079314	29.42
5	dsj1000	25526005	0.4340513	36.80
6	eil51	614	5.553E-4	44.13
7	gr202	52615	0.0187077	31.01
8	gr229	179335	0.0203114	33.23
9	kroA100	30536	0.0014703	43.48
10	kroD100	28599	0.001964	34.31
11	pcb442	68841	0.0833932	35.57
12	ulysses16	7788	6.06E-5	13.54
13	ulysses22	8308	9.25E-5	18.47

Tabella 3: Risultati dell'algoritmo `Tree_TSP`

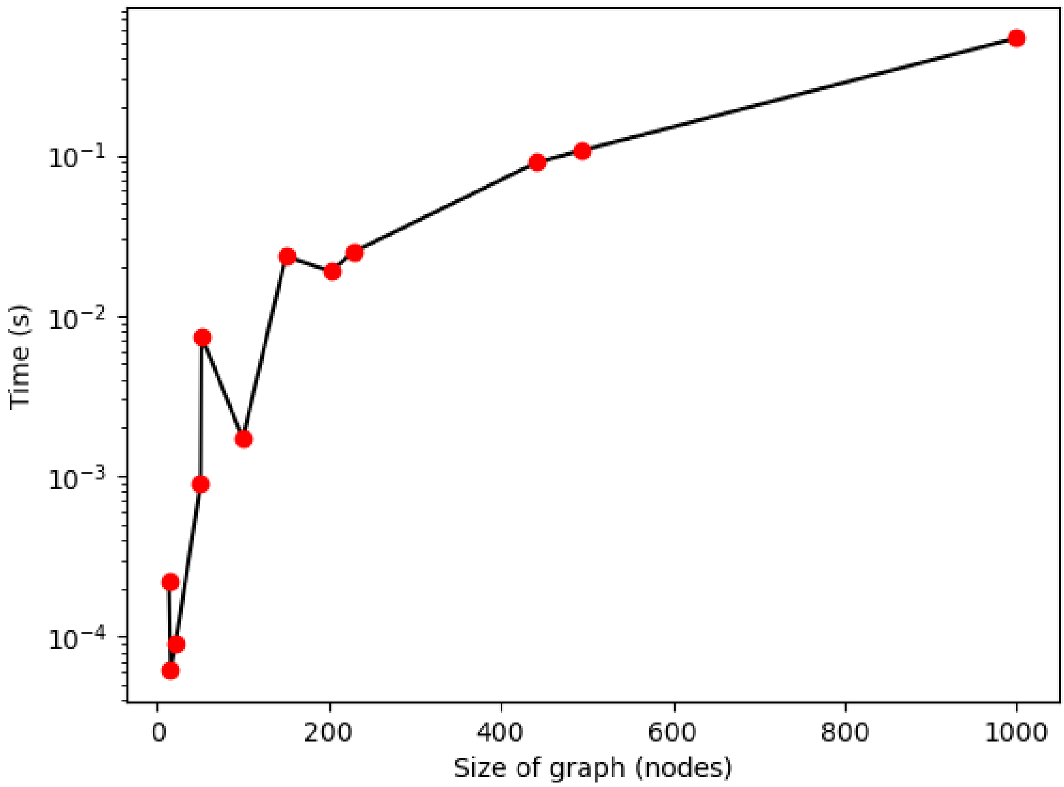

Figura 2: Performance dell'algoritmo `Tree_TSP`.

5 Conclusioni

In questa sezione sono analizzate le performance dell'algoritmo `HeldKarp` e viene effettuato un confronto delle performance tra `Tree_TSP` e `CheapestInsertion`.

5.1 HeldKarp

5.2 CheapestInsertion e Tree_TSP

Entrambi gli algoritmi sono una 2-approssimazione per TSP, ma riportano risultati differenti: `CheapestInsertion` richiede più tempo per essere eseguito rispetto a `Tree_TSP`, ma ottene un errore relativo minore. Per capire dunque quale dei due algoritmi sia più efficiente ed efficacie, è possibile osservare la Tabella 4 che mostra il prodotto tra l'errore relativo e il tempo di esecuzione. Facendo una media dei valori è possibile vedere come l'utilizzo di `Tree_TSP` sia più vantaggioso, infatti il prodotto ottenuto per `CheapestInsertion` è 13, mentre per `Tree_TSP` è 2. Questo perché, `Tree_TSP` ottiene in media un errore relativo più alto (32%) rispetto a `CheapestInsertion` (16%), ma possiede un tempo di esecuzione molto più basso, dunque per grafi di grande taglia è consigliato utilizzare `Tree_TSP`.

N.	**Name Graph**	**CheapestInsertion ratio**	**Tree_TSP ratio**
1	berlin52	0.2126	0.3175
2	burma14	0.0022	0.0042
3	ch150	1.6794	1.0099
4	d493	8.0955	3.1753
5	dsj1000	153.453	15.9731
6	eil51	0.1554	0.0245
7	gr202	1.3764	0.5801
8	gr229	0.7594	0.6749
9	kroA100	0.0702	0.0639
10	kroD100	0.0744	0.0674
11	pcb442	7.7838	2.9663
12	ulysses16	0.0002	0.0008
13	ulysses22	0.0009	0.0017
	Average	13.36	1.91

Tabella 4: Rapporto Errore-(Tempo di computazione) tra CheapestInsertion e Tree_TSP