Relazione di Algoritmi Avanzati

Laboratorio 2 - Traveling Salesman Problem

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 di tre diversi algoritmi visti a lezione per la risoluzione del *Travelling Salesman Problem*[1], definito come segue.

1.1 Problema TSP

"Date le coordinate x, y di N punti nel piano (i vertici) ed una funzone di peso $w(u, v)$ definita per tutte le coppie di punti (gli archi), trovare il ciclo semplice di peso minimo che visita tutti gli N punti. La funzione peso $w(u, v)$ è definita come la distanza Euclidea o Geografica tra i punti u e v. La funzione peso è simmetrica e rispetta la ***disuguaglianza triangolare***[2]*."*

1.2 Tipologia di distanze

La distanza tra due punti u e v può essere:

* **Euclidea:** dati due punti `(xa; ya)` e `(xb,yb)` la distanza tra di loro è calcolata come segue, arrotondando all'intero più vicino:

```
public static Integer euclidean(double xa, double ya, double xb, double yb){
    return Integer.valueOf((int) (Math.sqrt(Math.pow(xa - xb, 2) + Math.pow(ya - yb, 2)) + 0.5));
}
```

Figura 1: Calcolo della distanza Euclidea.

* **Geografica:** se TSP è un problema da risolvere con distanze geografiche, allora i punti si trovano nella superficie terrestre, di raggio 6378.388 km. Le loro coordinate indicano la latitudine e la longitudine del corrispondente punto sul pianeta Terra: di seguito viene riportato il metodo utilizzato per calcolare questo tipo di distanza.

```
public static Integer geo(double xa, double ya, double xb, double yb){
    xa = toRad(xa);
    ya = toRad(ya);
    xb = toRad(xb);
    yb = toRad(yb);

    double RRR = 6378.388;

    double q1 = Math.cos( ya - yb );
    double q2 = Math.cos( xa - xb );
    double q3 = Math.cos( xa + xb );
    return (int) ( RRR * Math.acos( 0.5*((1.0+q1)*q2 - (1.0-q1)*q3) ) + 1.0);
}

private static Double toRad(double x) {
    double PI = 3.141592;
    int deg = (int) x;
    double min = x - deg;
    return PI * (deg + 5.0 * min/3.0)/180.0;
}
```

Figura 2: Calcolo della distanza Geografica.

[1] d'ora in poi TSP

[2] $\forall\, u, v, w \in V$ vale che $c(u, v) \leq c(u, w) + c(w, v) \Rightarrow c(< u, v >) \leq c(< u, w, v >)$

Gli algoritmi da implementare per risolvere il problema rientrano in tre categorie:

1. **Algoritmi esatti**, ovvero l'algoritmo di **Held e Karp**[3] che ritorna la soluzione esatta del problema TSP con complessità $\mathcal{O}(n^2 2^n)$;

2. **Euristiche costruttive**, cioè algoritmi che non possono dare nessun vincolo sulla soluzione ritornata. Essi arrivano alla soluzione procedendo un vertice alla volta seguendo delle regole prefissate. In particolar modo vedremo l'euristica **Cheapest Insertion**[4] che, in caso la disuguaglianza triangolare venga rispettata, si dimostra approssimare la soluzione al più di un fattore 2. Generalmente, quando la disuguaglianza triangolare non viene rispettata, è possibile dimostrare che questo algoritmo permette di trovare una soluzione $\log(n)$-approssimata a TSP.

3. **Algoritmo 2-approssimato**, cioè un algoritmo di approssimazione che può ritornare una soluzione al più 2 volte peggiore di quella ritornata da un algoritmo esatto. Dato che la disuguaglianza triangolare viene rispettata, vedremo l'algoritmo **Triangle TSP**[5] che risolve il problema TSP con un algoritmo 2-approssimato attraverso la costruzione di un MST[6].

Infine, verranno esposti ed adeguatamente discussi i risultati ottenuti.

2 Scelta del linguaggio di programmazione

Per lo svolgimento di questo *assignment*, come per il precedente, è stato scelto Java nella sua versione 8 come linguaggio di programmazione. 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. Sono infatti evitate complessità aggiuntive derivanti da inavvertite copie profonde.

3 Scelte implementative

Come specificato nel precedente paragrafo, nell'implementazione dei tre algoritmi si è cercato di creare meno oggetti possibile usando quasi esclusivamente 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 `-Xmx6500m`.
Benché il codice sia stato adeguatamente commentato[7], di seguito è riportata una *summa* delle caratteristiche di ogni classe implementata che non compaiono nella documentazione, ripartita per `package`.

[3] d'ora in poi `HeldKarp`
[4] d'ora in poi `CheapestInsertion`
[5] d'ora in poi `TriangeTSP`
[6] *Minimum Spanning Tree*
[7] 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

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, che si presenta come una normale matrice $n \times n$ simmetrica rispetto alla sua diagonale nulla[8]. Gli algoritmi `HeldKarp`, `CheapestInsertion` e `TriangleTSP`, difatti, la trattano come tale. In verità è stata implementata come matrice triangolare inferiore, contenendo `n - 1` array di lunghezza crescente: questa 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 assignment.

* `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:

  – `copyWithoutV`: funzione *utility* di `HeldKarpCore` che serve a ricreare l'insieme $S \setminus \{v\}$, necessaria all'algoritmo di Held e Karp per proseguire con le chiamate ricorsive;

  – `getResults`: funzione che ritorna $(d[0, V], T)$, cioè il valore del ciclo hamiltoniano di costo minimo disponibile dopo T minuti di computazione;

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

  TSP contiene inoltre tre campi dati, utilizzati dall'algoritmo `HeldKarp`:

  – `d`: struttura dati che rappresenta $(d[v, S], \texttt{cost})$, cioè che mappa la coppia <v, S> con il corrispondente peso del cammino minimo che parte da 0 e termina in v, visitando tutti i nodi in S. Per memorizzare il valore di $d[v, S]$, quindi, si inserisce una nuova chiave <v, S> in `d` e la si mappa con il rispettivo costo minimo calcolato. Per recuperare un valore, invece, si accede alla struttura dati con la chiave <v, S>, chiedendo quale valore sia stato ivi memorizzato. Nell'implementazione, quindi, è stato scelto di rappresentarla con il tipo `HashMap<Pair<Integer, ArrayList<Integer>>, Integer>`;

  – `pi`: struttura dati dello stesso tipo e del tutto simile a `d`, che mappa la medesima chiave con un valore intero che rappresenta il nodo predecessore di v;

[8] poiché non esistono cappi

  - `w`: variabile segnaposto della matrice di adiacenza del grafo su cui calcolare `HeldKarp`, incapsulata nella classe `TSP` al solo scopo di offrire maggior leggibilità e coerenza con lo pseudocodice visto in classe.

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 `TrangularTSP`.

* `HeldKarp`: funzione adibita all'inizializzazione e alla chiamata dell'algoritmo di Held e Karp. Si occupa di preparare l'ambiente necessario alla funzione `HeldKarpCore` e di catturare eventuali eccezioni lanciate dalla JVM[9] dovute alla mancanza di memoria (`OutOfMemoryException`);

* `HeldKarpCore`: funzione principale che implementa l'algoritmo di Held e Karp, del tutto coerente con lo pseudocodice visto a lezione. L'unica differenza è, naturalmente, la gestione degli *interrupt*: se una certa istanza di TSP dovesse metterci più del tempo a disposizione T[10], l'algoritmo interrompe immediatamente ogni ulteriore visita del grafo lasciando comunque l'opportunità di memorizzare i risultati finora calcolati. Questo comporta, a volte, un ritardo sui tempi di computazione consentiti: ciò non è visto dagli studenti come un problema poiché, di fatto, la computazione viene interrotta quindi non si è in presenza di alcun *cheat*. L'algoritmo ha complessità $\mathcal{O}(n^2 \cdot 2^n)$

* `CheapestInsertion`: l'algoritmo fa uso del metodo `getMinAdjacentVertexWeightIndex` per trovare il lato col peso minore che ha come estremo il nodo 0. Viene 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 $n + 1$ e trova per ogni nodo non visitato (e per ogni lato presente nel cammino trovato fino a quel momento) il minore `minCost`, indicando che il nodo k non visitato deve essere inserito nel cammino e in quale posizione. L'algoritmo ha complessità $\mathcal{O}(n^3)$;

* `TriangleTSP`: l'algoritmo, attraverso l'utilizzo di `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 $\mathcal{O}(m \log n + n)$.

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;

[9] Java Virtual Machine
[10] come avviene nella maggior parte dei casi

  - `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 sugli algoritmi utilizzati come *Kruskal*.

* le funzioni `printHeapInfo` e `formatSize` che sono adibite, rispettivamente, a fornire informazioni e rappresentare l'utilizzo di memoria degli algoritmi.

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. I risultati ottenuti,poi, verranno opportunamente discussi.

4.1 Domanda 1

Eseguite i tre algoritmi che avete implementato (Held-Karp, euristica costruttiva e 2-approssimato) sui 13 grafi del dataset. Mostrate i risultati che avete ottenuto in una tabella come quella sottostante. Le righe della corrispondono alle istanze del problema. Le colonne mostrano, per ogni algoritmo, il peso della soluzione trovata, il tempo di esecuzione e l'errore relativo calcolato come $(SoluzioneTrovata - SoluzioneOttima)/SoluzioneOttima$. Potete aggiungere altre informazioni alla tabella che ritenete interessanti.

	Held-Karp			Euristica costruttiva			2-approssimato		
Istanza	Soluzione	Tempo (s)	Errore	Soluzione	Tempo (s)	Errore	Soluzione	Tempo (s)	Errore
burma14.tsp	3323	0,59	0,00%						
ulysses16.tsp									
berlin52.tsp									
kroA100.tsp									
ch150.tsp									
gr202.tsp									
pcb442.tsp									

Figura 3: Esempio di tabella riportante i risultati ottenuti.

4.1.1 Svolgimento

Istanza	Held-Karp			Cheapest Insertion			TriangleTSP		
	Soluzione	Tempo (s)	Errore (%)	Soluzione	Tempo (s)	Errore (%)	Soluzione	Tempo (s)	Errore (%)
berlin52.tsp	17441	120,0174	131,25	9004	0,0096	19,38	10402	0,0074	37.92
burma14.tsp	3323	0,2029	0,0	3588	0,0003	7,97	4003	0,0002	20.46
ch150.tsp	47935	120,95	634,30	7998	0,0678	22,52	9126	0,02343	39.80
d493.tsp	111947	120,4769	219,83	39969	0,4976	14,19	45300	0,1063	29.42
dsj1000.tsp	551274242	120,0012	2854,36	22291165	7,5382	19,46	25526005	0,5358	36.80
eil51.tsp	986	120,0005	131,46	494	0,0006	15,96	614	0,0009	44.13
gr202.tsp	55127	119,9998	37,26	46480	0,0406	15,74	52615	0,0189	31.01
gr229.tsp	176212	120,0008	30,91	153896	0,0493	14,33	179335	0,0249	33.23
kroA100.tsp	164223	120,0003	671,65	24942	0,0038	17,20	30536	0,0014	43.48
kroD100.tsp	144125	120,0012	576,83	25204	0,0036	18,36	28599	0,0017	34.31
pcb442.tsp	202233	120,4061	298,27	60834	0,3954	19,80	68841	0,0901	35.57
ulysses16.tsp	6859	0,5584	0,0	7368	0,0001	7,42	7788	0,0001	13.54
ulysses22.tsp	7013	59,4115	0,0	7709	0,0001	9,92	8308	0,0001	18.47

Tabella 1: Risultati dei tre algoritmi implementati rispetto alla domanda 1

4.2 Domanda 2

Commentate i risultati che avete ottenuto: come si comportano gli algoritmi rispetto alle varie istanze? C'è un algoritmo che riesce sempre a fare meglio degli altri rispetto all'errore di approssimazione? Quale dei tre algoritmi che avete implementato è più efficiente?

4.2.1 Svolgimento

* **Come si comportano gli algoritmi rispetto alle varie istanze?**
  - `HeldKarp`: nelle istanze fino di 14 e 16 nodi l'algoritmo termina quasi immediatamente, restituendo la soluzione ottima. Nell'istanza con 22 nodi impiega poco meno di un minuto per trovarla. Per tutte le altre istanze, occorre dare un tempo limite a disposizione dell'algoritmo in quanto impiegherebbe decisamente troppo tempo per calcolare la soluzione ottima. Pur interrompendo la computazione, non è possibile fare alcuna previsione sulla qualità della soluzione ritornata calcolata fino a quel punto, che arriva anche ad essere del 2854,36% nel caso di `disj1000.tsp`.
  - `CheapestInsertion`: l'algoritmo ritorna una soluzione 2-approssimata[11] quasi immediatamente, fatta eccezione per `disj1000.tsp` che impiega circa 7,5 secondi. Ogni soluzione ritornata ha un errore che appartiene all'intervallo $[7.97 – 22.52]\%$: in media ogni soluzione ha un errore relativo del 15,56% rispetto alla sua soluzione ottima. Un notevole risultato, tenendo conto che potrebbe teoricamente ritornare una soluzione con errore relativo al più del 200%.
  - `TriangleTSP`: l'algoritmo ritorna una soluzione 2-approssimata quasi immediatamente per tutte le istanze del dataset. Ogni soluzione ritornata possiede un errore relativo che appartiene all'intervallo $[13.54 – 44.13]\%$: in media ogni soluzione ha un errore relativo del 30.74%. Un risultato quasi doppio rispetto all'euristica `CheapestInsertion`: ulteriori chiarimenti saranno forniti nella sezione §4.2.1.1

* **C'è un algoritmo che riesce sempre a fare meglio degli altri rispetto all'errore di approssimazione?**
  - `HeldKarp` riesce sempre a fare meglio di `CheapestInsertion` e `TriangleTSP` nelle istanze con meno di 23 nodi, vale a dire `burma14.tsp, ulysses16.tsp` e `ulysses22.tsp` perché riesce a trovare la soluzione ottima. Per tutte le altre istanze del dataset, `CheapestInsertion` riesce sempre a trovare una soluzione più vicina alla soluzione ottima rispetto agli altri due algoritmi implementati: il perché di questo dato di fatto non è ci è perfettamente chiaro. Abbiamo pensato che potrebbe essere perché...

* **Quale dei tre algoritmi che avete implementato è più efficiente?**
  - Considerando il termine "efficienza" ed il suo significato in campo informatico, inteso come la capacità di utilizzare meno risorse[12] possibile durante la sua esecuzione, l'algoritmo più efficiente si rivela essere `TriangleTSP` perché è corretto rispetto al vincolo di 2-approssimazione ed impiega meno risorse di `CheapestInsertion`. Quest'ultimo, tuttavia, riesce a trovare una soluzione quasi il doppio più accurata del primo, a fronte di utilizzi di risorse più che accettabili per le istanze di TSP fornite. È per questo che ci sentiamo di dire che l'algoritmo che, mediamente, riesce a combinare **efficienza** ed **efficacia** meglio di tutti sia proprio `CheapestInsertion`.

4.2.1.1 Discussione su `CheapestInsertion` e `TriangleTSP`

Entrambi gli algoritmi sono una 2-approssimazione per TSP, ma riportano risultati differenti: `CheapestInsertion` richiede più tempo per essere eseguito rispetto a `TriangleTSP`, ma ottiene un errore relativo minore.

[11] grazie al vincolo rispettato di disuguaglianza triangolare

[12] tempo di utilizzo della CPU e memoria

Per capire dunque quale dei due algoritmi sia più efficiente ed efficace, è possibile osservare gli istogrammi in Figura 15 che mostrano prima la differenza del tempo di esecuzione dei due algoritmi su scala logaritmica per i vari grafi, poi il tempo di esecuzione su una scala normale ed infine il rispettivo errore relativo. Confrontando il primo e terzo istogramma è possibile vedere come per un aggiunta di tempo di computazione relativamente basso si ottiene un notevole miglioramento per l'errore relativo, infatti `TriangleTSP` raggiunge in media un errore relativo più alto (~32%) rispetto a `CheapestInsertion` (~16%). Se si osserva il secondo istogramma si può vedere come con grafi di taglia maggiore il tempo di esecuzione per `CheapestInsertion` aumenti esponenzialmente rispetto a `TriangleTSP`, basti osservare il tempo di esecuzione per i grafi `d449.tsp` e `dsj1000.tsp`: dunque per grafi di grande taglia è consigliabile utilizzare `TriangleTSP`, altrimenti `CheapestInsertion`.

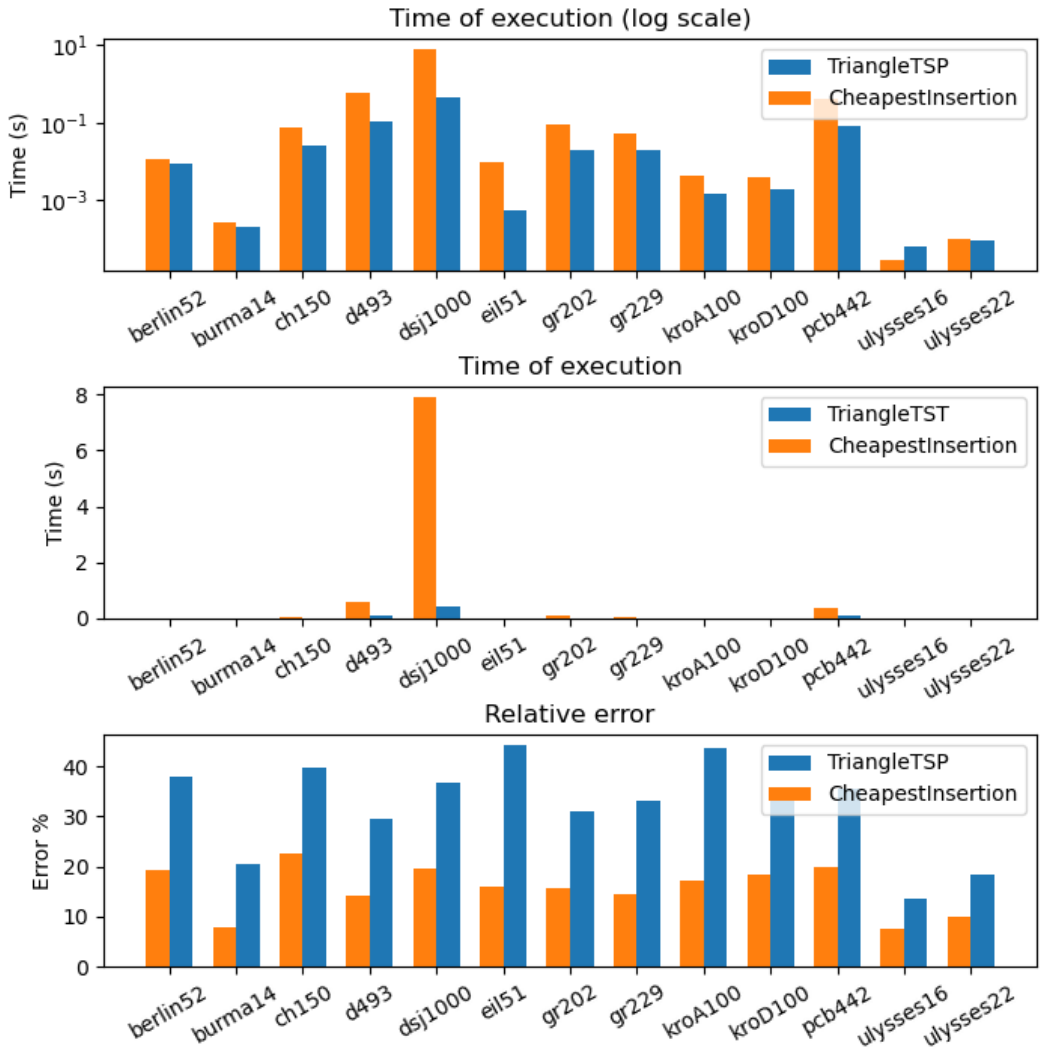

Figura 4: Confronto tra `CheapestInsertion` e `TriangularTSP`

5 Descrizione degli algoritmi

5.1 HeldKarp

Le strutture dati utilizzate dall'algoritmo `HeldKarp` sono essenzialmente 3:

* `AdjacentMatrix`, cioè la matrice di adiacenza dei nodi dell'istanza di TSP, trattata come matrice $n \times n$ simmetrica alla diagonale ma implementata come matrice diagonale superiore come specificato in §3.1;
* `Pair<Integer, ArrayList<Integer>` letteralmente la chiave $<d, S>$ che consiste nella coppia di valori di tipo `Integer` e `ArrayList<Integer>`, che rappresentano rispettivamente il nodo-destinazione e l'insieme di nodi da visitare, partendo dal nodo 0 e finendo al nodo-destinazione appena illustrato;
* `HashMap<Pair<Integer, ArrayList<Integer>>, Integer>`, che mappa a coppia $<d, S>$ di cui sopra ad un valore di tipo `Integer`. Questa struttura dati serve a rappresentare due entità fondamentali per l'algoritmo `HeldKarp`:
  - `d`, rappresentata da $d[v, S]$ nello pseudocodice e descritta in 3.1;
  - `pi`, rappresentata da $\pi[v, S]$ nello pseudocodice e descritta in 3.1.

L'algoritmo parte formando l'insieme dei nodi di partenza V, cioè tutti i nodi dell'istanza di TSP e invocando `HeldKarpCore`.

L'algoritmo `HeldKarpCore` è il vero e proprio algoritmo di Held e Karp. Dopo aver gestito i casi base in tempo $\mathcal{O}(1)$, costruisce l'insieme $S \setminus \{v\}$ indicato nel codice con il nome di `S_new` in tempo $\mathcal{O}(n-1) = \mathcal{O}(n)$.

A questo punto esaminiamo il ciclo `for`. Esso, tolte le chiamate ricorsive, itera su tutti i vertici in $S \setminus \{v\}$ e quindi impiega un tempo $\mathcal{O}(n)$. Analizzando ora le chiamate ricorsive, si osserva che terminano immediatamente quando la coppia $<v, S> \in d$, ovvero quando `d.containsKey(Pair<v,S>) = true`: questo ci fa dedurre che le chiamate ricorsive che eseguono `HeldKarpCore` casi base esclusi, è limitato superiormente dal numero di elementi indicizzati in `d`. Questo numero risulta essere $\leq |V| \cdot 2^{|V|}$, con $|V|$: numero di nodi dell'istanza di TSP.

Indicando con n la quantità $|V|$ e tenendo conto anche del tempo necessario alla costruzione dell'insieme $S \setminus \{v\}$, possiamo quindi affermare che il tempo totale impiegato da `HeldKarpCore` è pari a $\mathcal{O}(n+n) \cdot \mathcal{O}(n \cdot 2^n) = \mathcal{O}((2n) \cdot n \cdot 2^n) = \mathcal{O}(n^2 \cdot 2^n)$

```java
public void HeldKarp() {
    int size = w.size();
    ArrayList<Integer> S = new ArrayList<Integer>(size-1);
    for(int i=1; i<size; ++i) {
        S.add(i);
    }
    try {
        HeldKarpCore(0, S);
    }catch (OutOfMemoryError e) {
        printHeapInfo();
        System.out.println("Need more memory!");
        System.out.println();
    }
}

private Integer HeldKarpCore(Integer v, ArrayList<Integer> S){
    if(S.size() == 1 && S.contains(v))
        return w.get(v,0);
    Pair<Integer, ArrayList<Integer>> index = new Pair<Integer, ArrayList<Integer>>(v,S);
    if(d.containsKey(index))
        return d.get(index);
    Integer mindist = Integer.MAX_VALUE;
    Integer minprec = null;

    ArrayList<Integer> S_new = copyWithoutV(S, v);

    for(Integer u : S_new) {
        Integer dist = HeldKarpCore(u,S_new);
        if(Integer.sum(dist, w.get(u,v)) < mindist.intValue()) {
            mindist = Integer.sum(dist, w.get(u, v));
            minprec = Integer.valueOf(u);
        }
        if(Thread.currentThread().isInterrupted()) {
            break;
        }
    }
    d.put(index, mindist);
    pi.put(index, minprec);

    return mindist;
}
```

Figura 5: Snippet degli algoritmi `HeldKarp` ed `HeldKarpCore`.

5.2 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 cicli `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 1, i quali aumentano 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 22,52%, dimostrando dunque di essere molto efficace.

```
Integer nextNode = w.getMinAdjacentVertexWeightIndex(0);

path.add(0);
notVisited.remove(Integer.valueOf(0));
path.add(nextNode);
notVisited.remove(nextNode);
path.add(0);

for(int i = 0; i < w.size() - 2; i++){
    Integer tmpNode = 0;
    int minCost = Integer.MAX_VALUE;
    for(Integer nodeK : notVisited){
        for(int j = 0; j < path.size() - 1; j++){
            int tmpCost = w.get(path.get(j), nodeK)
            + w.get(nodeK, path.get(j + 1)) - w.get(path.get(j), path.get(j + 1));
            if(tmpCost < minCost){
                tmpNode = j;
                nextNode = nodeK;
                minCost = tmpCost;
            }
        }
    }
    path.add(tmpNode + 1, nextNode);
    notVisited.remove(nextNode);
}
```

Figura 6: Snippet del codice per `CheapestInsertion`.

5.3 TriangleTSP

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 `TriangleTSP`, 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%, impiegando un tempo di esecuzione di ~0.5 secondi. Dunque un algoritmo molto efficiente.

```java
public int TriangularTSP(){
    Node tree = Kruskal();//O(mlog n)
    int cost = 0;
    ArrayList<Integer> path = preorder(tree, new ArrayList<Integer>());//O(n)
    path.add(0);

    for(int i = 0; i < path.size() - 1; i++)
        cost += w.get(path.get(i), path.get(i + 1));

    return cost;
}
```

Figura 7: Snipped del codice per `TriangleTSP`.

6 Originalità introdotte

In questa sezione vengono riportate le originalità che abbiamo ritenuto significative e che abbiamo voluto introdurre nella relazione, ripartite per algoritmo.

6.1 Held-Karp

Mobilitati da curiosità abbiamo provato a variare il tempo di calcolo a disposizione dell'algoritmo di Held e Karp, per vedere se le soluzioni che impiegavano più di due minuti ritornavano una soluzione migliore, oppure se, con meno tempo a disposizione, la soluzione ritornata risultava peggiore. Le tabelle sottostanti riportano quello che è stato ottenuto.

6.1.1 Held-Karp con 1 minuto

Istanza	**Held-Karp**		
	Soluzione	**Tempo (s)**	**Errore (%)**
berlin52	17441	60,706	131,25
burma14	3323	0,1014	0,00
ch150	47935	59,999	634,29
d493	111947	60,598	219,83
dsj1000	551274242	60,001	2854,35
eil51	986	60,000	131,45
gr202	55127	60,000	37,26
gr229	176212	59,999	30,91
kroA100	165018	60,000	675,38
kroD100	146158	59,999	586,38
pcb442	202517	59,999	298,82
ulysses16	6859	0,3550	0,00
ulysses22	7013	53,181	0,00

Tabella 2: Risultati dell'algoritmo Held-Karp concedendo 1 minuto

6.1.2 Held-Karp con 5 minuti

Istanza	**Held-Karp**		
	Soluzione	**Tempo (s)**	**Errore (%)**
berlin52	17441	303,951	131,25
burma14	3323	0,15270	0,00
ch150	47935	316,682	634,29

Continua nella pagina seguente

Tabella 3 – *continuazione della pagina precedente*

Istanza	**Held-Karp**		
	Soluzione	**Tempo (s)**	**Errore (%)**
d493	111947	308,136	219,83
dsj1000	551274242	306,492	2854,35
eil51	984	300,463	130,98
gr202	55127	305,269	37,26
gr229	176212	312,290	30,91
kroA100	162606	308,605	664,05
kroD100	144125	314,625	576,83
pcb442	202044	300,972	297,89
ulysses16	6859	0,40430	0,00
ulysses22	7013	57,7751	0,00

Tabella 3: Risultati dell'algoritmo Held-Karp concedendo 5 minuti

6.1.3 Held-Karp con 10 minuti

Istanza	**Held-Karp**		
	Soluzione	**Tempo (s)**	**Errore (%)**
berlin52	17441	603,349	131,25
burma14	3323	0,15163	0,00
ch150	47885	606,479	633,53
d493	111947	603,658	219,83
dsj1000	550719964	611,369	2851,38
eil51	963	601,293	126,05
gr202	55125	605,391	37,26
gr229	175982	603,008	30,74
kroA100	161443	699,899	658,58
kroD100	143960	605,555	576,05
pcb442	202044	603,171	297,89
ulysses16	6859	0,40640	0,00
ulysses22	7013	58,3268	0,00

Tabella 4: Risultati dell'algoritmo Held-Karp concedendo 10 minuti

6.1.4 Conclusioni

La variazione del tempo concesso per il calcolo dell'algoritmo `HeldKarp` non ha avuto alcun miglioramento né peggioramento per sei istanze di TSP su tredici. Per tre di loro, precisamente `burma14.tsp, ulysses16.tsp e ulysses22.tsp`, questo accade perché la soluzione ottima viene già trovata entro un intervallo di tempo relativamente basso, quindi non considerato perché ritenuto poco interessante: la più "impegnativa" delle tre, `ulysses22.tsp`, è in grado di ritornare la soluzione ottima in poco meno di un minuto. Le altre due in meno di un secondo.

Per quanto riguarda le rimanenti sette istanze di TSP, vale a dire `ch150.tsp, dsj1000.tsp, eil51.tsp, gr202.tsp, gr229.tsp, kroA100.tsp, kroD100.tsp` invece si è registrato un aumento della qualità della soluzione ritornata, all'aumentare del tempo di computazione concesso. I grafici sottostanti riportano i dati inseriti nelle tabelle di cui sopra, mostrando più chiaramente la differenza degli errori relativi calcolati come $(SoluzioneTrovata - SoluzioneOttima)/SoluzioneOttima$, ripartito in base alle diverse tempistiche assegnate.

ch150.tsp

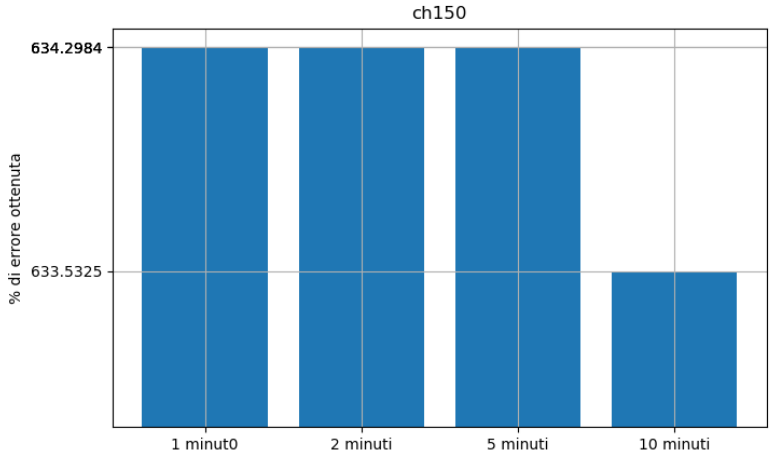

Figura 8: Variazione dell'errore relativo in ch150.

dsj1000.tsp

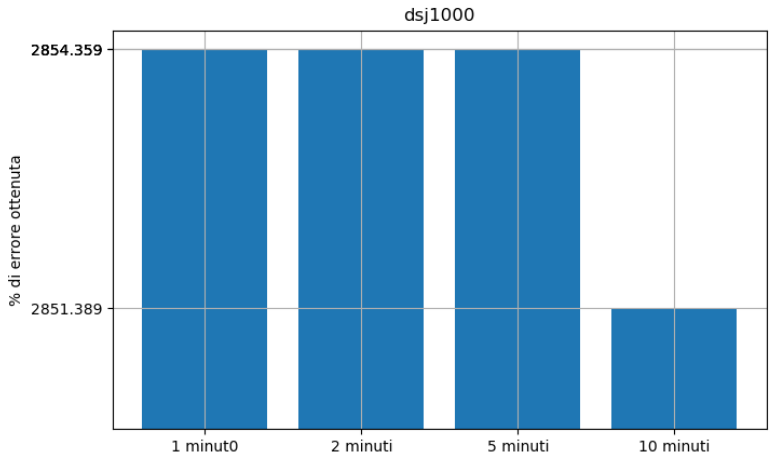

Figura 9: Variazione dell'errore relativo in dsj1000.

eil51.tsp

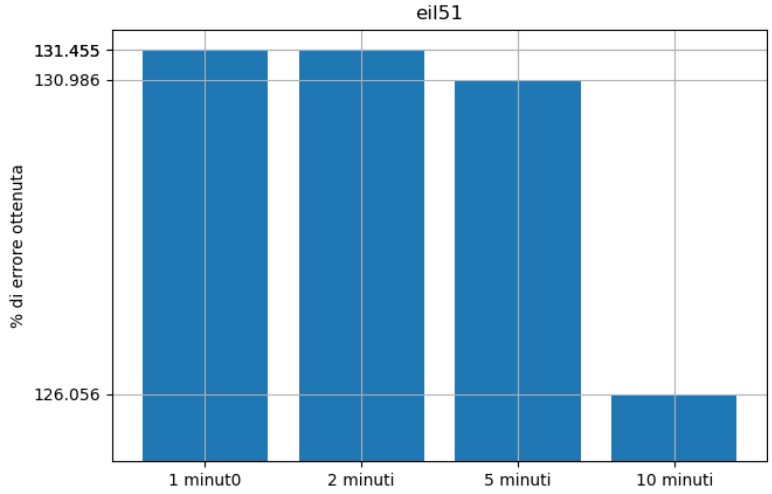

Figura 10: Variazione dell'errore relativo in eil51.

gr202.tsp

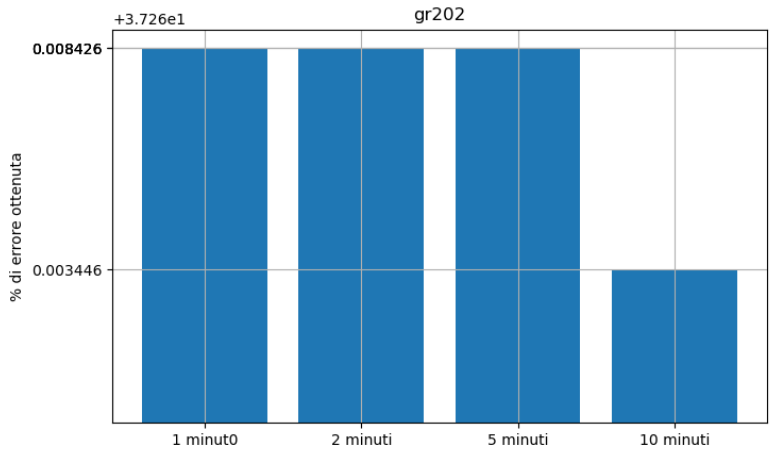

Figura 11: Variazione dell'errore relativo in gr202.

Nota: in questo grafico, essendo troppo piccola la variazione, i due valori riportati sull'asse delle ordinate vanno sommati con il numero scritto in notazione scientifica in alto a sinistra.

gr229.tsp

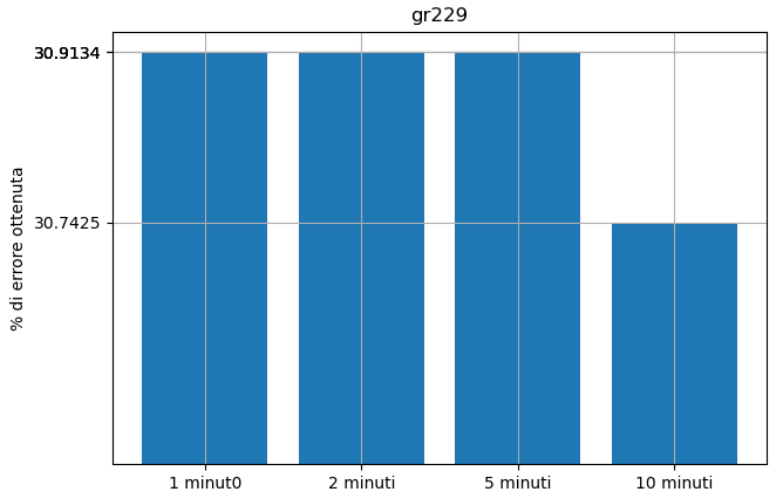

Figura 12: Variazione dell'errore relativo in gr229.

kroA100.tsp

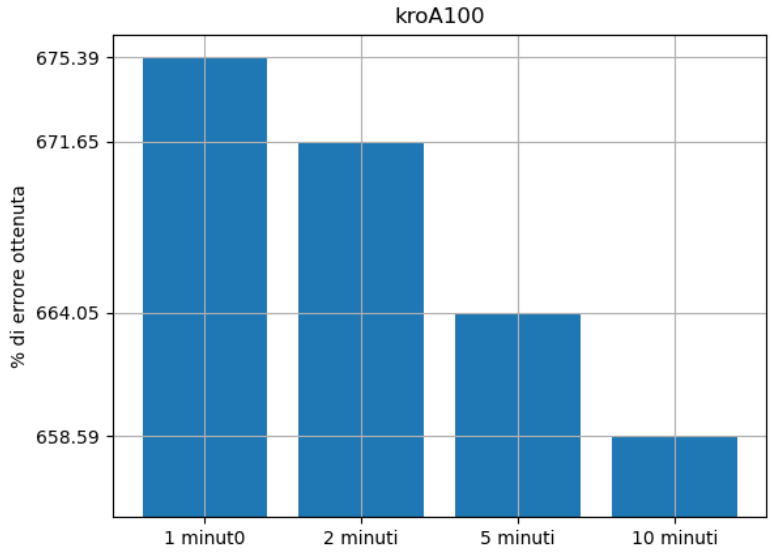

Figura 13: Variazione dell'errore relativo in kroA100.

kroD100.tsp

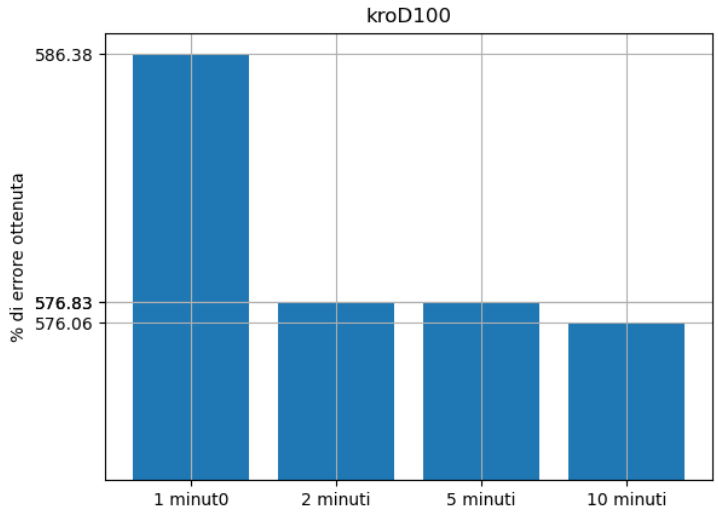

Figura 14: Variazione dell'errore relativo in kroD100.

Come ci si poteva aspettare, concedendo più tempo di computazione all'algoritmo `HeldKarp`, abbiamo registrato una probabilità non banale che l'algoritmo migliori le sue *performance*:

* Non considerando le tre istanze di TSP in cui l'algoritmo riesce a trovare la soluzione ottima sotto al minuto, si rimane quindi con dieci istanze: `berlin52.tsp`, `ch150.tsp`, `d493.tsp`, `dsj1000.tsp`, `eil51.tsp`, `gr202.tsp`, `gr229.tsp`, `kroA100.tsp`, `kroD100.tsp` e `pcb442.tsp`.
* Sette di loro hanno un miglioramento della soluzione ritornata $\Rightarrow$ **70%** delle possibilità che `HeldKarp` restituisca una soluzione più vicina alla soluzione ottima all'aumentare del tempo concesso.

Seppure ci sia chiaro che non sia corretto derivare giudizi generali di qualsiasi tipo avendo a disposizione così poche istanze di TSP, riteniamo di aver ottenuto una percentuale davvero alta, di cui si debba tenere conto, risultato sperimentale che conferma ciò che ci aspettavamo.

Crediamo che, alla luce di quanto detto finora, sia il caso di chiederci: vale la pena concedere fino a dieci minuti per trovare una soluzione migliore?

* La risposta che ci siamo dati è, non sorprendentemente, **dipende**: se il grafo ricavato dall'istanza di TSP possiede un numero di nodi n:
  - $0 < n \leq 22$ allora assegnare più tempo a `HeldKarp` **è indifferente**, in quanto l'algoritmo ritorna una soluzione ottima in meno di un minuto (quindi si ferma in ogni caso);
  - $23 \leq n < 52$ allora assegnare più tempo a `HeldKarp` **probabilmente non conviene**, perché impiegando invece l'euristica `CheapestInsertion` si ottengono risultati statisticamente migliori (vedi tabella 4.1.1). Ad ogni modo, occorre precisare che per valori di n immediatamente successivi a 22, invece, **probabilmente conviene** lasciare più tempo all'algoritmo, perché potrebbe ritornare una soluzione ottima entro 10 minuti.
  - $n \geq 52$ allora assegnare più tempo a `HeldKarp` **sicuramente non conviene**, perché abbiamo sperimentato che, per ogni istanza di TSP, risultati migliori si possono ottenere impiegando l'euristica `CheapestInsertion`.

6.2 CheapestInsertion e TriangularTSP

Entrambi gli algoritmi sono una 2-approssimazione per TSP, ma riportano risultati differenti: `CheapestInsertion` richiede più tempo per essere eseguito rispetto a `TriangularTSP`, ma ottiene un errore relativo minore.

Per capire dunque quale dei due algoritmi sia più efficiente ed efficace, è possibile osservare gli istogrammi in Figura 15 che mostrano prima la differenza del tempo di esecuzione dei due algoritmi su scala logaritmica per i vari grafi, poi il tempo di esecuzione su una scala normale ed infine il rispettivo errore relativo. Confrontando il primo e terzo istogramma è possibile vedere come per un aggiunta di tempo di computazione relativamente basso si ottiene un notevole miglioramento per l'errore relativo, infatti `TriangularTSP` raggiunge in media un errore relativo più alto (~32%) rispetto a `CheapestInsertion` (~16%). Se si osserva il secondo istogramma si può vedere come con grafi di taglia maggiore il tempo di esecuzione per `CheapestInsertion` aumenti esponenzialmente rispetto a `TriangularTSP`, basti osservare il tempo di esecuzione per i grafi *d449* e *dsj1000*.

Dunque per grafi di grande taglia è consigliabile utilizzare `TriangularTSP`, altrimenti `CheapestInsertion`.

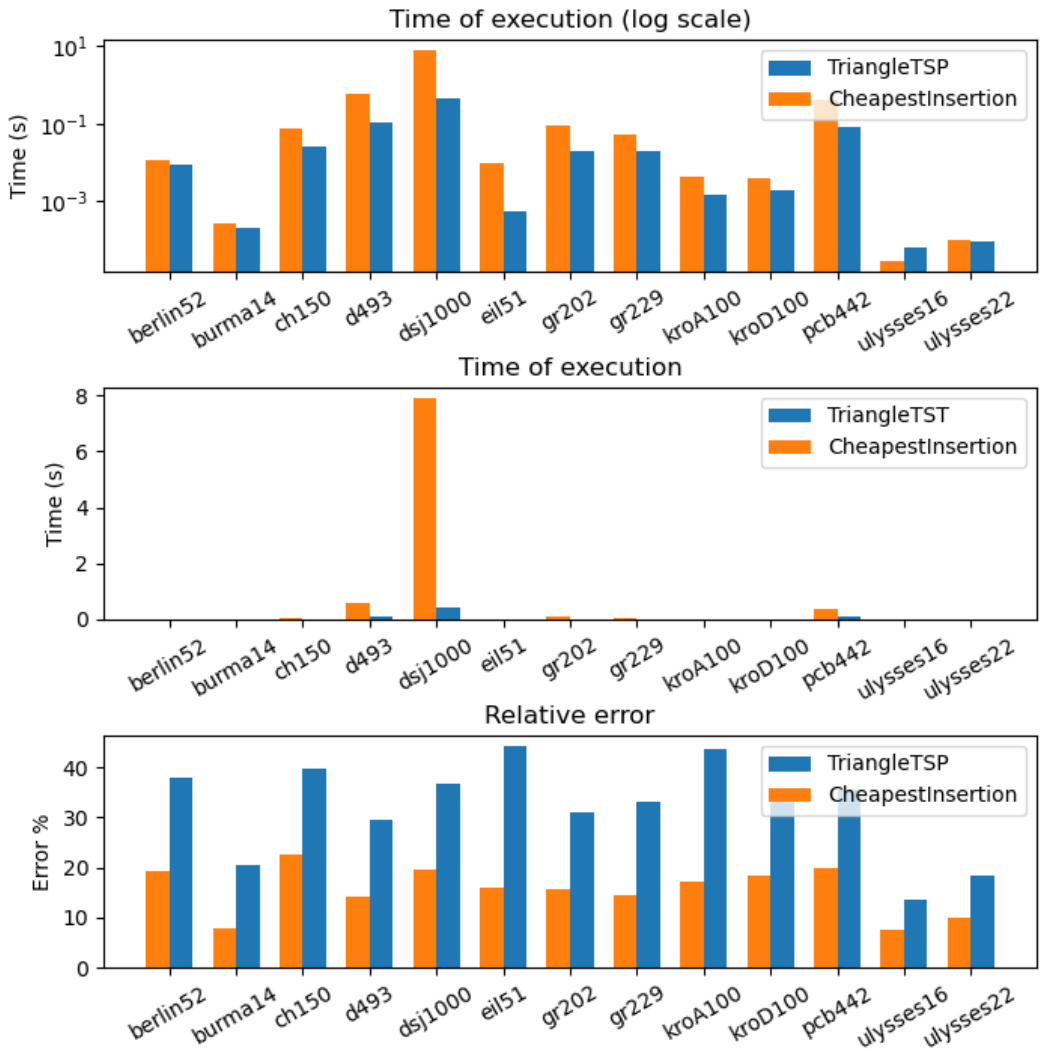

Figura 15: Confronto tra `CheapestInsertion` e `TriangularTSP`