Corso di Algoritmi Avanzati

Laboratorio 2: Traveling Salesman Problem

10 luglio 2020

Busin Lorenzo	1237580
Tartaggia Nicolò	1237784
Voinea Ciprian	1237294

Indice

1 Introduzione

Questo elaborato ha lo scopo di illustrare il lavoro svolto per il secondo homework del corso *Algoritmi Avanzati*.
L'homework ha come obiettivo quello di confrontare tra loro gli algoritmi per il problema intrattabile chiamato "Problema del Commesso Viaggiatore" (*Traveling Salesman Problem - TSP*) definito come segue: date le coordinate x,y di N punti nel piano (i vertici), e una funzione 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 (ciclo hamiltoniano). La funzione peso $w(u, v)$ è definita come la distanza Euclidea o Geografica tra i punti u e v. La soluzione ottimale consiste nell'andare a calcolare il cammino di lunghezza minima.
L'homework ha come obiettivo quello di confrontare tra loro gli algoritmi esatti e con algoritmi di approssimazione:

1. *Algoritmo Held-Karp* (Sez. 3)
2. *Euristica costruttiva* (Sez. 4)
3. *Algoritmo 2-approssimato* (Sez. 5)

Il linguaggio in cui sono stati implementati questi algoritmi è `Python`. Abbiamo deciso di utilizzare questo al contrario di altri linguaggi come `C++` o `Java` in quanto questa scelta ci ha permesso di utilizzare i *Jupyter Notebook* e di programmare utilizzando l'IDE *PyCharm* oppure con *Google Colab*.

Nella Sez. 6 sono presenti i risultati, in forma tabellare, che confrontano le performance dei tre algoritmi, calcolati sul dataset dato, contenente 13 grafi di esempio, di dimensione compresa tra 14 e 1000 vertici e descritti in file `.tsp`. Sono inoltre illustrati i grafici delle performance dei vari algoritmi.

Il lavoro è stato suddiviso equamente tra i membri del gruppo nel seguente modo:

- Algoritmo Held-Karp: Lorenzo Busin
- Euristica costruttiva: Nicolò Tartaggia
- Algoritmo 2-approssimato: Ciprian Voinea

Nonostante questa suddivisione tutti i membri hanno collaborato all'implementazione degli algoritmi ed effettuato una verifica finale tramite *peer review* cercando di seguire una linea di sviluppo comune.

2 Funzioni comuni

Per questo homework sono state utilizzate delle funzioni comuni per le lettura e la gestione dell'input e per il calcolo della distanza tra due vertici.

- `import_dataset(path)`: metodo che permette di leggere il contenuto dell'input, utilizzando il percorso specificato dal parametro `path`. Esso memorizza e ritorna il numero totale di vertici, il formato dei vertici e la lista di vertici. I valori ritornati vengono salvati in variabili globali, in modo da essere accessibili in qualunque momento e da ridurre il numero di parametri passati alle varie funzioni.
  Tale metodo ritorna:
  - `dim`: numero di vertici all'interno del dataset;
  - `wt`: tipo delle coordinate (`EUC_2D`, `GEO`);
  - `V`: vettore di coppie in formato `(i, [x, y])`.
- `weight(u, v)`: metodo che permette di calcolare la distanza tra due vertici in base al loro formato. Se questo è di tipo *EUC-2D* viene calcolata la distanza euclidea mentre se è di tipo *GEO* le coordinate vengono prima convertite in radianti e, successivamente, viene calcolata la distanza geografica.

3 Algoritmi esatti

```
1  HELD_KARP (v,S)
2      if S = {v} then
3          return w[v,0]
4      else if d[v,S] 6= NULL then
5        return d[v,S]
6      else
7        mindist = infinity
8        minprec = NULL
9        for all u in S \ {v} do
10           dist = HK–VISIT(u,S \ {v})
11           if dist + w[u, v] < mindist then
12               mindist = dist + w[u, v]
13               minprec = u
14       d[v,S] = mindist
15       phi[v,S] = minprec
16       return mindist
```

L'algoritmo di Held e Karp è un algoritmo di programmazione dinamica per risolvere il problema del TSP proposto nel 1962. La programmazione dinamica è una tecnica che permette di risolvere problemi di ottimizzazione combinando le soluzioni di sottoproblemi più semplici: l'algoritmo si occupa di risolvere ognuno dei sottoproblemi una volta sola, salvando poi il risultato in una tabella per poterlo riutilizzare in seguito senza doverlo risolvere nuovamente. L'algoritmo si basa sul fatto che ogni sottocammino di un cammino minimo è a sua volta un cammino minimo e questa proprietà permette di risolvere il problema del `TSP` in modo ricorsivo.

3.1 Strutture dati

L'algoritmo si basa sulle seguenti strutture dati:

- `V`: vettore di coppie in formato `(i, [x, y])` dove `i` rappresenta l'indice progressivo nell'insieme `{0, .., n - 1}` e `[x, y]` corrisponde alle coordinate del punto;
- `subsets`: dizionario usato per enumerare i sottoinsiemi di `V`;
- `d`: dizionario che usa come chiavi le coppie `(v, S)` e come valore il peso del cammino minimo che parte da 0 e termina in `v`, visitando tutti i nodi in `S`;
- `phi`: dizionario che usa come chiavi le coppie `(v, S)` e come valore il predecessore di `v` nel cammino minimo.

3.2 Implementazione

La soluzione del problema è stata implementata nel seguente modo:

- Viene ricavato l'indice corrispondente al sottoinsieme `S` presente all'interno di `subsets`;

- Se il tempo trascorso dall'inizio dell'esecuzione dell'algoritmo è superiore a `max_time` la funzione ritorna il valore `None` come segnale per interrompere la computazione, altrimenti passa all'istruzione successiva;
- Se `S` ha un solo elemento ed esso coincide con `v` il valore ritornato sarà il peso dell'arco che ha come estremi 0 e `v`, altrimenti passa all'istruzione successiva;
- Se la soluzione al sottoproblema `[v, S]` è già presente in `d` il suo valore viene ritornato dall'algoritmo, altrimenti passa all'istruzione successiva;
- Nel caso ricorsivo inizialmente vengono inizializzati i valori di `mindist` e `minprec`, poi viene costruito il nuovo sottoinsieme `S \ {v}` e se non è già presente in `subsets`, la sua enumerazione viene aggiunta incrementando il contatore. Per rendere univoco ogni sottoinsieme esso viene trasformato, tramite la funzione `encode`, ottenendo così una stringa composta dagli indici dei vertici contenuti al suo interno separati da uno spazio (ad esempio l'insieme `{1, 2, 3}` verrà codificato nella stringa "1 2 3"). Per ogni vertice `u` presente nell'insieme `S \ {v}` viene effettuata la chiamata ricorsiva a `held_karp(u, S \ {v}, max_time)`: se il risultato di tale chiamata è `None` significa che il tempo limite è stato superato e quindi il ciclo verrà interrotto restituendo la soluzione migliore finora trovata, altrimenti l'algoritmo cercherà la soluzione migliore al sottoproblema che, dopo aver aggiornato `v` e `phi`, sarà ritornata in output.

3.3 Complessità

La complessità dell'algoritmo è $O(n^2 \cdot 2^n)$, dove `n` indica il numero di vertici:

- In ognuno dei casi base l'algoritmo ha complessità $O(1)$;
- Nel caso ricorsivo il ciclo `for` itera su tutti i vertici nell'insieme `S \ {v}` e quindi impiega tempo $O(n)$. Il numero di chiamate ricorsive è limitato superiormente dal numero di elementi presenti in `d[v, S]` che è al più $n \cdot 2^n$, con `v` vertice del grafo e `S` un sottoinsieme di vertici.

4 Euristiche costruttive

Il termine euristiche costruttive identifica un'ampia famiglia di euristiche che costruiscono la soluzione finale aggiungendo un vertice alla volta seguendo uno schema prefissato. Tra le euristiche disponibili, l'algoritmo implementato è `Nearest Neighbor`. Questo tipo di algoritmo ha una natura *greedy*, in quanto va ad aggiungere il vertice più vicino al vertice attualmente in coda al cammino parziale.

```
Nearest_Neighbor (V) // V = list of vertexes
    1) \\ Initialization
       Select v_0 from V as starting vertex of the partial path
    2) \\ Selection
       Given (v_0 ... v_k) as partial path, the next vertex v_{k+1} not added yet
       to the partial path is the nearest one to v_k.
    3) \\ Insertion
       Insert v_{k+1} right after v_k in the partial path
    4) Repeat the process from the step (2) until all the vertexes have
       been inserted
    5) Close the path inserting the starting vertex v_0 at the end of the
       partial path (v_0 ... v_n)
```

La soluzione ritornata dall'euristica è $log(n)$-approssimata a TSP.

4.1 Funzioni

L'implementazione del problema non ha richiesto particolari strutture dati. Di conseguenza sono state create delle apposite funzioni per la costruzione dell'algoritmo:

- `closest_vertex(p, V)`: metodo che permette di calcolare la distanza minima tra il vertice `p`, il quale si trova in coda alla lista di vertici del cammino, e la lista di vertici `V` non ancora inseriti. Una volta individuato il vertice più vicino, questo viene ritornato insieme alla distanza calcolata;
- `nearest_neighbor(V)`: metodo che permette di costruire il circuito hamiltoniano seguendo le regole dell'euristica `Nearest Neighbor`.

4.2 Implementazione

La soluzione finale è la seguente:

- Viene eseguita una copia profonda `C` della lista `V`. La copia ottenuta è utilizzata per effettuare le operazioni di costruzione del cammino, mentre la lista originale rimane invariata;
- Inizializzazione: viene inizializzato il cammino iniziale contenente il vertice di partenza, il quale viene rimosso da `C`;

- Viene dichiarata una variabile locale `current_distance` per memorizzare la distanza del cammino corrente;
- A questo punto inizia il ciclo interno: il processo di iterazione continua finché tutti i vertici di `C` sono stati visitati;
    1. Selezione: avviene la chiamata alla funzione `closest_vertex(p, V)`;
    2. Inserimento: il vertice ritornato nella fase precedente viene inserito in coda al cammino parziale e rimosso da `C`;
- Una volta completato il ciclo `for` interno, viene chiuso il ciclo hamiltoniano inserendo nuovamente il vertice di partenza;
- La distanza totale calcolata viene ritornata come risultato dell'algoritmo.

4.3 Complessità

L'algoritmo `Nearest Neighbor` può essere implementato in tempo proporzionale a n^2, dove n è il numero di vertici del grafo:

- La ricerca del vertice a distanza minima da un dato vertice ha complessità $O(n)$;
- La ricerca precedente avviene per tutti gli n vertici. Complessità finale $O(n^2)$.

5 Algoritmo 2-approssimato

```
1 APPROX-TSP-TOUR(G, c)
2     select a vertex r ∈ G.V to be a root vertex
3     compute a MST T for G from root r using MST-PRIM(G, c, r)
4     let H be a list of vertices, ordered according to when they are first ↩
          visited in a preordered tree walk of T
5     return the hamiltonian cycle H
```

L'algoritmo di 2-approssimazione utilizza come sottoprocedura l'algoritmo di Prim, che fornisce un MST il cui peso dà un lower bound sulla lunghezza del ciclo della soluzione ottima del TSP sul grafo dato.
Questo algoritmo sfrutta la disuguaglianza triangolare, ovvero dati tre nodi A, B, C, la distanza tra A e C può al massimo la distanza tra A e B sommata alla distanza tra B e C.
Fintanto che la funzione del costo soddisfa tale regola, viene utilizzato il minimum spanning tree per creare un ciclo con un costo che è massimo due volte il costo della soluzione ottima, per questo si dice 2-approssimato.

5.1 Strutture dati

Per implementare l'algoritmo di 2-approssimazione è stato necessario l'utilizzo dell'algoritmo di Prim, che richiede la struttura dati `Heap`. Questa è stata presa dalla soluzione del primo homework e adattata in maniera da poter essere utilizzata con gli indici presenti nel dataset e con grafi completi.
Sono state riutilizzate anche le strutture dati `Graph` e `Node`, adattando la prima per la costruzione di grafi completi.

5.2 Funzioni

Le funzioni ausiliarie utilizzate per calcolare la soluzione dell'algoritmo di 2-approssimazione per TSP sono le seguenti:

- `MSTPrim`: metodo che ritorna l'MST di un grafo in input partendo dal nodo dato;
- `preorder`: presi in input la mappa dei successori e il vertice di partenza, ritorna una lista della visita in profondità dell'albero a partire da tale nodo;
- `mst_to_tree`: data in input la mappa dei predecessori ciascun nodo, la converte in un albero;
- `ham_circ_weight`: metodo che, dato in input un ciclo hamiltoniano, ne calcola il peso;
- `two_approximation`: metodo principale che implementa le operazioni principali dell'algoritmo di 2-approssimazione.

5.3 Implementazione

La soluzione del problema è stata implementata nel seguente modo:

- viene costruito il grafo `g` a partire dai lista di vertici letta dal file in input;
- si utilizza l'algoritmo di Prim per calcolare l'MST del grafo appena creato;
- tale MST viene convertito, tramite l'uso della funzione `mst_to_tree`, in una lista che rappresenta i padri di ciascun nodo;
- andando a riordinare questa lista tramite la funzione `preorder`, si ottiene un ciclo hamiltoniano al quale viene appeso in coda il primo nodo, con indice 0, per chiuderlo;
- viene quindi calcolata la soluzione finale andando a sommare i pesi degli archi che collegano i nodi del ciclo.

5.4 Complessità

Per calcolare la complessità di questo algoritmo bisogna tenere in considerazione anche le operazioni che vengono svolte da Prim e, di conseguenza, quelle svolte sullo heap.
La complessità totale è quindi $O(|V|^2)$, dove V indica la lista di vertici del grafo.

6 Risultati

6.1 Tabella dei risultati

	Held-Karp			Nearest Neighbor			2-Approssimato		
Istanza	**Soluzione**	**Tempo (s)**	**Errore (%)**	**Soluzione**	**Tempo (s)**	**Errore (%)**	**Soluzione**	**Tempo (s)**	**Errore (%)**
burma14.tsp	3323	1.78971	0.00	4048	0.00031	21.82	4003	0.00146	20.46
ulysses16.tsp	6859	9.34773	0.00	9988	0.00034	45.62	7788	0.00197	13.54
ulysses22.tsp	8110	90.00008	15.64	10586	0.00051	50.95	8308	0.00332	18.47
eil51.tsp	1082	90.00043	153.99	511	0.00150	19.95	605	0.01534	42.02
berlin52.tsp	18923	90.00042	150.90	8980	0.00150	19.07	10402	0.01549	37.92
kroD100.tsp	150055	90.00158	604.68	26947	0.00509	26.55	28599	0.05820	34.31
kroA100.tsp	167968	90.00131	689.25	27807	0.00477	30.66	30516	0.05387	43.39
ch150.tsp	48636	90.00295	645.04	8191	0.01123	25.47	9126	0.36979	39.80
gr202.tsp	56530	90.00515	40.76	49336	0.03299	22.85	52615	0.47693	31.01
gr229.tsp	176922	90.00675	31.44	162430	0.04380	20.67	179335	0.55846	33.23
pcb442.tsp	206008	90.02463	305.70	61979	0.09068	22.06	72853	1.49114	43.47
d493.tsp	111941	90.03063	219.81	41660	0.11230	19.02	45595	1.84300	30.26
dsj1000.tsp	552601511	90.12495	2861.47	24630960	0.47197	32.00	25526005	6.86773	36.80

Figura 1: Risultati del TSP calcolati

6.2 Grafico di confronto delle performance

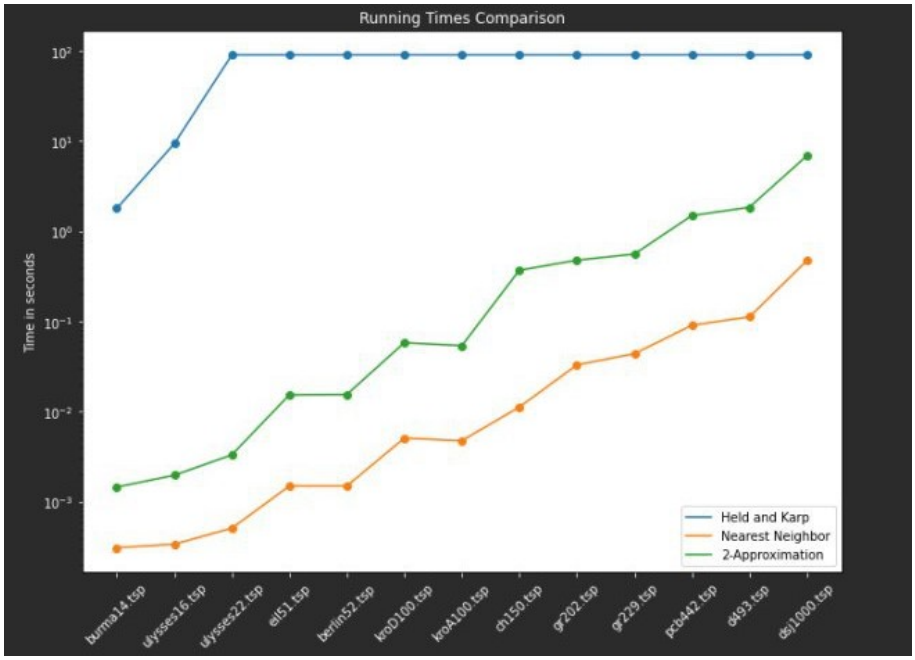

Figura 2: Performance degli algoritmi

6.3 Grafico di confronto degli errori

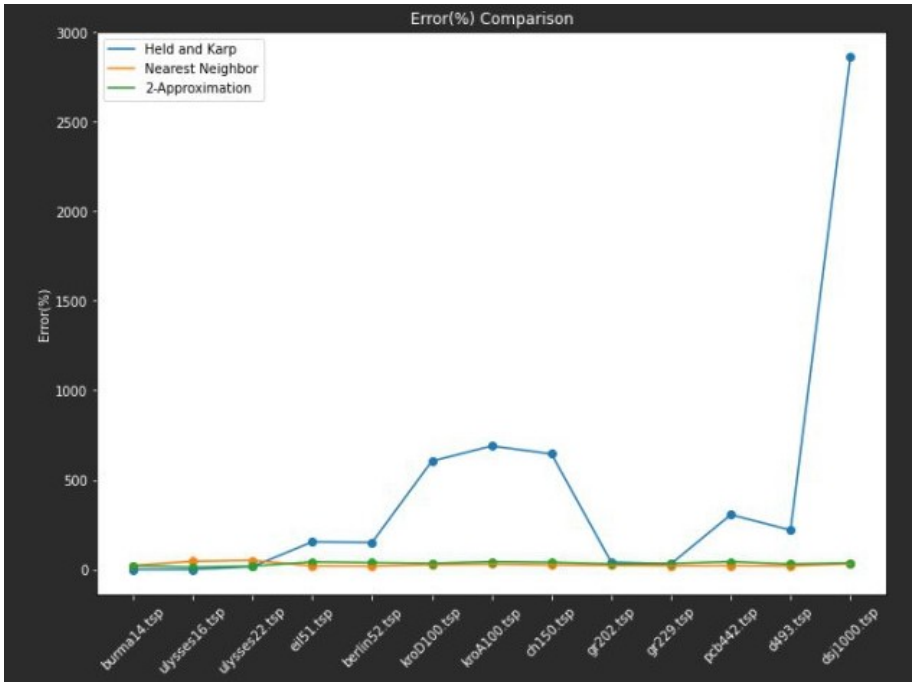

Figura 3: Errori degli algoritmi

6.4 Grafico di confronto degli errori con scala logaritmica

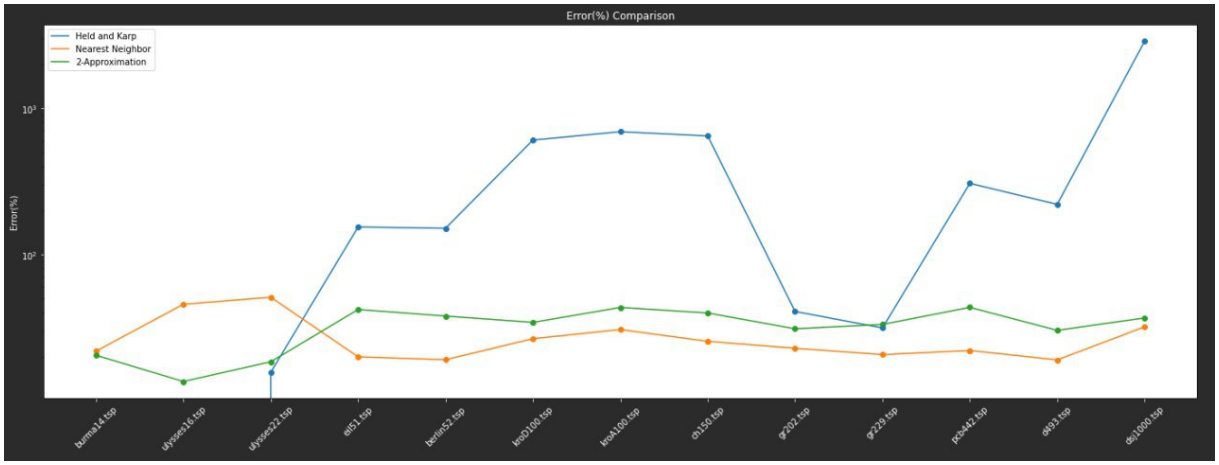

Figura 4: Logaritmo degli errori degli algoritmi

7 Conclusione

Per quanto riguarda la prima e la seconda istanza, l'algoritmo di Held-Karp riesce a terminare entro il limite dei 90 secondi, trovando la soluzione ottima. Per quanto riguarda le altre istanze, sicuramente Held-Karp troverebbe la soluzione ottima se avessimo a disposizione un tempo infinito. Per le altre istanze, Nearest Neighbor risulta migliore: Held-Karp, allo scadere dei 90 secondi, ha una soluzione ottima solamente per una parte dei sottoproblemi, proseguendo poi in maniera non più ottimale.
Non c'è dunque un algoritmo che fa meglio degli altri in tutte le istanze dei problemi proposti, dato un limite di tempo all'algoritmo di Held-Karp.
Tuttavia, Nearest Neighbor è quello che presenta mediamente tempi di esecuzione ed errori inferiori rispetto agli altri algoritmi.
È stato scelto di impiegare l'algoritmo di Prim in quanto presenta una complessità minore, $O(|V|^2)$, rispetto a quello di Kruskal. Anche l'euristica Nearest Neighbor ha complessità $O(|V|^2)$, ma dai tempi rilevati si può dire che quest'ultima ha delle costanti asintotiche più basse.
Naturalmente l'algoritmo meno efficiente è Held-Karp che ha complessità $O(2^{|V|} * |V|^2)$, che però garantisce una soluzione ottima quando termina, ovvero avendo un tempo sufficiente.