Confronto tra algoritmi per TSP

di Costantino Marco

1 Introduzione

Gli algoritmi implementati sono:

- L'algoritmo di Held-Karp;
- L'euristica Nearest Neighbor;
- L'algoritmo 2-Approssimato.

Il linguaggio scelto per l'implementazione degli algoritmi è *python*.

Ho scelto python per ragioni di continuità con il precedente assignment. Per migliorare le prestazioni ho usato, al posto dell'interprete standard di python, *pypy* un compilatore just-in-time per codice python.

1.1 La struttura dati Graph

La struttura dati che ho usato per rappresentare i grafi consiste in:

- La lista dei vertici del grafo;
- La matrice di adiacenza del grafo. Ovviamente i valori salvati nella matrice corrispondono alle distanze opportunamente calcolate in base al formato del file.

1.2 L'algoritmo di Held-Karp

Ho implementato l'algoritmo di Held-Karp a partire dallo pseudo-codice visto a lezione. L'implementazione della struttura dati che usa come indici delle coppie `(vertice, insieme di vertici)` è stata fatta attraverso dei dizionari che usano come chiave delle coppie `(intero, frozenset di interi)`. *Frozenset* è una struttura dati nativa di python, versione immutabile di *set*. Questa proprietà rende la struttura dati *hashable*, e quindi utilizzabile come chiave per un dizionario. Per terminare l'esecuzione dell'algoritmo dopo un certo numero di minuti, l'algoritmo necessita un parametro: `max_time` che esprime in minuti la durata massima di esecuzione

dell'algoritmo. Questo parametro viene usato insieme a delle misurazioni del tempo trascorso per interrompere la ricorsione e collassare lo stack di chiamate ricorsive ritornando l'ultima `mindist` calcolata.

1.3 Nearest Neighbor

L'euristica Nearest Neighbor costruisce il cammino partendo da un vertice e prendendo come successivo il vertice "più vicino" (tale che il lato è di peso minimo) evitando vertici già incontrari. Quando sono esauriti i vertici del grafo, si aggiunge al termine del cammino il vertice di partenza e al peso del cammino il peso del lato tra l'ultimo vertice e quello di partenza. L'implementazione è piuttosto facile, un ciclo aggiunge al cammino il vertice più vicino (trovato guardando nella riga del vertice in esamine nella matrice di adiacenza) e lo rimuove dalla lista dei vertici considerabili (per evitare di re-incontrare vertici).

1.4 L'algoritmo 2-Approssimato

A differenza della versione vista a lezione ho usato come subroutine l'algoritmo di Kruskal al posto dell'algoritmo di Prim, questo perchè la mia implementazione dell'algoritmo di Kruskal (sviluppata per il precedente assignment) è più veloce della mia implementazione dell'algoritmo di Prim. Dopo di che, utilizzo una subroutine per costruire la matrice di adiacenza del MST trovato. Questa viene visitata con una versione di DFS iterativa (equivalente ad una visita prefissa in cui ignoro l'ordine dei figli del nodo in esame). Questa versione iterativa di DFS per funzionare ha bisogno di fare *backtracking* una volta giunta ad una foglia, si ispira a *backtracking search* vista nel corso di Intelligenza Artificiale. Terminata la visita in profondità viene aggiunta alla lista dei vertici il vertice di partenza. Infine una subroutine calcola il peso del cammino tra i vertici.

2 Domanda 1 - I risultati

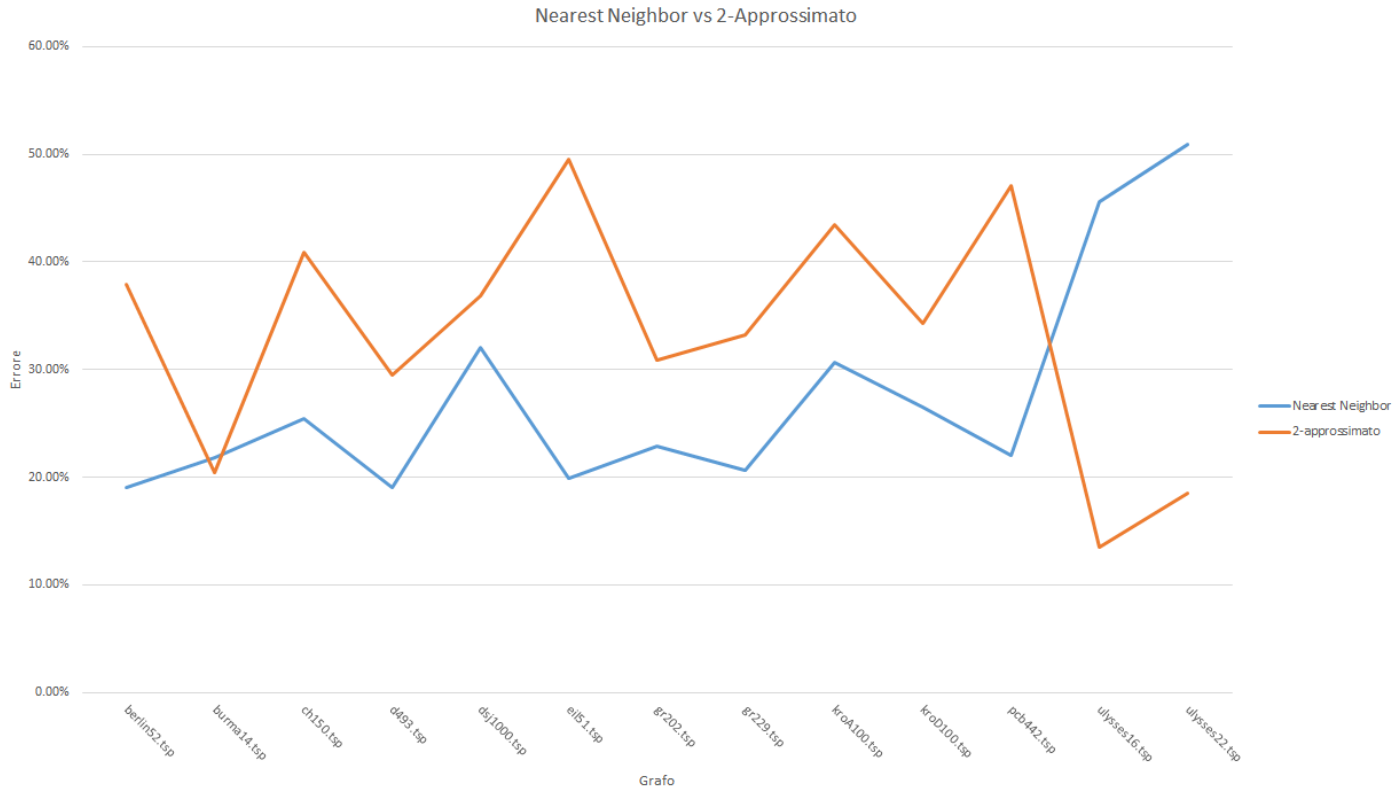
I risultati sono stati raccolti usando la funzione di python `time.time()`, prima e dopo l'esecuzione dei diversi algoritmi e sottraendo i valori ottenuti, questi sono stati salvati in dei file `.csv`, insieme ai pesi dei cammini ottenuti.

File	Held-Karp			Nearest Neighbor			2-approssimato		
	Soluzione	**Tempo (s)**	**Errore**	**Soluzione**	**Tempo (s)**	**Errore**	**Soluzione**	**Tempo (s)**	**Errore**
berlin52.tsp	17.739	240.001	135.20%	8.980	0.003	19.07%	10.402	0.045	37.92%
burma14.tsp	3.323	0.393	0.00%	4.048	0.000	21.82%	4.003	0.005	20.46%
ch150.tsp	48.006	240.000	635.39%	8.191	0.006	25.47%	9.200	0.092	40.93%
d493.tsp	111.941	240.001	219.81%	41.660	0.014	19.02%	45.327	62.909	29.50%
dsj1000.tsp	551.275.688	240.004	2854.37%	24.630.960	0.016	32.00%	25.526.005	340.748	36.80%
eil51.tsp	1.024	240.000	140.38%	511	0.000	19.95%	637	0.002	49.53%
gr202.tsp	55.127	240.000	37.27%	49.336	0.001	22.85%	52.546	1.291	30.84%
gr229.tsp	176.680	240.000	31.26%	162.430	0.001	20.67%	179.335	0.822	33.23%
kroA100.tsp	166.257	240.000	681.21%	27.807	0.000	30.66%	30.536	0.008	43.48%
kroD100.tsp	146.862	240.000	589.69%	26.947	0.000	26.55%	28.599	0.008	34.31%
pcb442.tsp	204.852	240.001	303.43%	61.979	0.001	22.06%	74.695	4.101	47.10%
ulysses16.tsp	6.859	2.218	0.00%	9.988	0.000	45.62%	7.788	0.000	13.54%
ulysses22.tsp	7.105	240.000	1.31%	10.586	0.000	50.95%	8.308	0.000	18.47%
Media	**-**	**203.278**	**433.02%**	**-**	**0.003**	**27.44%**	**-**	**31.541**	**33.55%**

I valori medi calcolati, sebbene fortemente influenzati dai valori massimi suggeriscono che Nearest Neighbor sia l'algoritmo preferibile sia dal punto di vista temporale sia dal punto di vista della soluzione ritornata. In un solo caso l'errore commesso da Nearest Neighbor supera il 50%.

Una cosa che non è evidente dalla tabella è il funzionamento di Held-Karp al variare del numero massimo di minuti di esecuzione:

Dal grafico si può notare che le misurazioni su certi grafi terminano prima di altre. La ragione di ciò è che su quei grafi dopo x minuti l'algoritmo esauriva la memoria a disposizione del processo, ritornando quindi un `MemoryError`. Ad esclusione dei grafi con 14 e 16 vertici, l'esecuzione su tutti gli altri restituisce un `MemoryError` entro gli 8 minuti dall'inizio dell'esecuzione, senza quindi ritornare il risultato ottimo. Inoltre è possibile notare come a distanza di minuti di esecuzione, la soluzione individuata non migliori di molto ed in alcuni casi, non migliori affatto. Per quanto riguarda invece Nearest Neighbor e l'algoritmo 2-approssimato, il seguente grafico riassume l'andamento dell'errore percentuale dei due algoritmi.



3 Domanda 2 – Conclusioni

Dai risultati è evidente la complessità $O(n^2\ 2^n)$ dell'algoritmo di Held-Karp. Al di fuori delle istanze molto piccole (14, 16 vertici) l'algoritmo esaurisce la memoria instanziata per il processo prima di individuare la soluzione ottima. In questo riguardo forse una versione iterativa di Held-Karp potrebbe funzionare meglio.

Dal punto di vista temporale, l'euristica Nearest Neighbor è di molto il migliore algoritmo dei 3, con un tempo massimo di 16 millisecondi sul grafo più grande del dataset (1000 vertici).

Per quanto riguarda la soluzione trovata, Held-Karp da soluzioni di molto lontane dall'ottimo quando non riesce a terminare prima di esaurire la memoria. Altrimenti il risultato è quello ottimo.

Come attesso, l'algoritmo 2-approssimato ha errori costantemente sotto il 50%.

Nearest Neighbor commette errori più grandi dell'algoritmo 2-approssimato solo in 4 casi: berlin52, burma14, ulysses16, ulysses22 ovvero sui grafi più piccoli del dataset. In un solo caso l'errore supera il 50%.