Algoritmi Avanzati: Travelling Salesman Problem

Zanatta Stefano

May 25, 2020

1 Implementazione

Ho implementato, in Python, tre algoritmi per il Travelling Salesman Problem (TSP).

- HeldKarp: l'implementazione è fedele all'algoritmo; ho dovuto aumentare il limite massimo di chiamate ricorsive (da 10^4 a 10^6);
- Random Insertion: composta da tre funzioni
  - initialize: fase di inizializzazione, crea un circuito composto dal nodo zero e il più vicino a questo;
  - random_insert: metodo principale, comprende la fase di selezione (randomica, tra i nodi non visitati);
  - insert: inserisce il nodo selezionato nel circuito;
- mst_tsp: algoritmo basato sul minimum spanning tree, in particolare con Prim; trova un minimum spanning tree, calcola la preorder list (cioè il circuito approssimato), aggiunge il nodo radice alla fine della lista, per chiudere il circuito e ritorna il suo peso.

Solo l'implementazione di Held-Karp è limitata nel tempo (180 secondi per file). Se viene raggiunto il tempo limite, vengono comunque fatte le chiamate ricorsive necessarie per avere un risultato differente da ∞.
Il grafo è implementato tramite lista di adiacenza. Solo per l'algoritmo mst_tsp, ho adattato la lista di adiacenza al codice del primo progetto, trasformandola in una istanza della classe classe Graph.
Ho parallelizzato l'esecuzione degli algoritmi in due fasi: prima Held Karp, poi i restanti. In ogni fase, ogni input esegue in parallelo con gli altri. Ho fatto questa divisione per garantire più memoria e cpu a Held Karp.

2 Risultati

L'algoritmo di Held-Karp richiede più tempo d'esecuzione rispetto alle euristiche e mst_tsp. Se la taglia dell'input è sufficentemente piccola, il risultato è corretto. In caso contrario, raggiunge il limite di 3 minuti e ritorna un pessimo valore approssimato (errore del 30% per grafi con 200 nodi, 96% sul un grafo con 1000 nodi). Per poter eseguire tutte le chiamate ricorsive, ho dovuto aumentare considerevolmente il limite di chiamate ricorsive di Python (fino a 10^6).
L' euristica da risultati approsismati, con un errore fino al 13%.
L'algoritmo basato sul mst ritorna risultati peggiori di Random Insertion, fino al 30% di errore. Il migliore tra i tre algoritmi, rispetto all'errore e la velocità, è Random Insertion.
A differenza di Held Karp, l'euristica e mst-tsp dipendono solo in parte dalla taglia dell'input: con pochi nodi i risultati sono generalmente buoni, ma con l'aumentare degli stessi, non c'è una crescita lineare dell'errore (per esempio, l'errore di ch150 è maggiore di dsj1000).

	Random			MST			Held Karp		
berlin52	8706	0.0	0.133701	10402	0.79688	0.274947	18940	181.35941	0.601795
burma14	3530	0.0	0.05864	4003	0.04687	0.169873	3323	2.18748	0.0
ch150	7362	0.0	0.113284	9123	2.0625	0.284446	48676	180.5	0.865889
d493	39061	0.21878	0.103914	45690	2.46875	0.233924	112147	180.45315	0.687892
dsj1000	20859222	1.48438	0.105447	25526005	3.60934	0.268993	553848300	180.90626	0.966309
eil51	450	0.0	0.053333	605	0.84375	0.295868	1110	180.40628	0.616216
gr202	44516	0.04691	0.097852	52615	2.5625	0.23672	57406	179.85938	0.300422
gr229	147524	0.03128	0.087593	179335	2.48439	0.249438	176922	179.67187	0.239201
kroA100	24114	0.0	0.117442	30516	1.59372	0.302595	173950	178.76565	0.877654
kroD100	23165	0.0	0.080768	28599	1.56247	0.255429	152844	179.17188	0.860681
pcb442	57356	0.37503	0.114687	75974	1.76559	0.33164	214041	178.71878	0.762765
ulysses16.tsp	7386	0.0	0.071351	7788	0.60937	0.119286	6890	177.25	0.004499
ulysses22.tsp	7341	0.0	0.044681	8308	0.32812	0.155874	8649	177.34375	0.189155

3 Considerazioni

Ho utilizzato una lista di adiacenza per rappresentare il grafo, perché più pratica rispetto ad una implementazione con una classe Python (come avevo fatto nel primo progetto). Ho riutilizzato del codice dal primo progetto (ridotto al minimo indispensabile) per calcolare il mst tramite Prim (nella cartella src.prim)