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).

	Held Karp			Random			TSP		
	Weight	Time	Error	Weight	Time	Error	Weight	Time	Error
berlin52	18923	180.65368	0.601437	8260	0.002	0.086925	10402	0.89265	0.274947
burma14	3323	10.34522	0.0	3494	0.0	0.048941	4003	0.12818	0.169873
ch150	48987	180.15268	0.86674	6823	0.042	0.043236	9123	2.13298	0.284446
d493	112147	180.88268	0.687892	39054	0.94951	0.103754	45690	3.49595	0.233924
dsj1000	554605113	180.01668	0.966355	21111263	3.32013	0.116126	25526005	5.6858	0.268993
eil51	1119	179.89468	0.619303	472	0.00097	0.097458	605	1.13079	0.295868
gr202	57362	180.29868	0.299885	44121	0.12796	0.089776	52615	3.04677	0.23672
gr229	176922	179.82268	0.239201	149124	0.081	0.097382	179335	3.02879	0.249438
kroA100	175161	179.66268	0.8785	22345	0.01697	0.047572	30516	1.5298	0.302595
kroD100	152587	179.89468	0.860447	23162	0.01803	0.080649	28599	1.27965	0.255429
pcb442	214041	179.7077	0.762765	57679	0.70348	0.119645	75974	2.88739	0.33164
ulysses16.tsp	6890	179.40868	0.004499	7210	0.0	0.048682	7788	0.35509	0.119286
ulysses22.tsp	8649	179.62368	0.189155	7356	0.0	0.046629	8308	0.41609	0.155874

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)