Delta Stepping Algorithm Raport

Morari Maxim

Facultatea de Informatica, Universitatea Alexandru Ioan Cuza, Iasi, Romania

1 Descrierea generala a alogritmului

Alogirtum Delta Stepping mentine distantele tentative de la sursa la celelalte noduri; o colectie de buckets, fiecare avand un interval egal cu delta, in care se mentin nodurile a caror distanta de la sursa inca nu s-a definitivat.

Alogritmul cauta in continuu urmatorul non-empty bucket, cauta si relaxeaza light requests pentru toate nodurile din el. Aceasta este o faza a algoritmului, inainte de care se sterg toate nodurile din bucket. Dupa relaxari, noduri pot fi reinserate in el. Aceste faze continua pana ce bucketul devine vid. Pe parcursul tuturor faze se tine minte multimea tuturor nodurilor ce au fost in bucketul curent. Cand el devine vid, se cauta si relaxeaza heavy requests pentru multimea discutata mai sus.

Toata aceasta procedura acum se repeta pana ce nu mai sunt non-empty buckets.

Un request contine nodul a carui distanta se compara cu cea tentativa, si distanta in sine. Un light request poate fi gasit de la un nod parinte pentru un copil, doara daca valoare arcul parinte-copil $\leq$ delta. Similar pentru heavy request, doar ca valoarea arcului $>$ delta.

A relaxa un request inseamna a compara noua distanta cu cea tentativa, si in caz ca noua distanta e mai mica se schimba eventual bucketul nodului respectiv.

Paralelizarea se obtine prin cautarea si relaxarea heavy/light requesturilor in paralel.

2 Detalii Implementare

La inceputul algoritmului calculez in paralel light/heavy edges pentru fiecare nod, verific daca sunt arcuri negative si calculez delta ca media tuturor arcurilor folosind paternul fork-join.

Apoi caut urmatorul non-empty bucket. Memorez bucketurile ca `ConcurrentHashMap<Integer, ConcurrentSkipListSet<Integer>>`.

Pentru a gasi urmatorul non-empty bucket mentin un currentBucketIndex si maxBucketIndex si incrementez currentBucketIndex pana ce

`buckets[currentBucketIndex]` $\neq$ null.

Nodurile din bucket acum sunt distribute in mod egal intre threaduri si se incepe o faza a algoritmului.

In fiecare faza se cauta light request si se relaxeaza.

Pentru a sincroniza paralelizarea, un singur thread poate lucra cu un nod la un moment dat. Adica doar unul poate schimba distanta tentativa a acestui nod, a-l sterge din si insera in bucketuri. Altfel, mai multe noduri se pot insera si sterge din acelasi bucket concomitent, el fiind un ConcurrentSkipListSet.

Dupa mai multe faze, bucketul devine vid. Acum Se cauta si relaxeaza heavy requests din toate nodurile memorate pe parcusurl tuturor fazelor legate de bucketul curent intr-un `HashSet<Integer>`.

Acum se reia cautarea urmatorului non-empty bucket si se repeta pana ce nu mai exista bucketuri.

Pentru menajarea threadurilor folosesc ExecutorService.

3 Analiza performantei

Pentru a calcula timpul de rulare a algoritmului am folosit `System.nanoTime()`. Pentru a oferi niste rezultate statistice, am rulat algoritmul de mai multe ori pentru acelasi input si am calculat media timpurilor.

Rezultate pentru 3 grafuri

Graph Name	**Vertices Count**	**Edges Count**	**Number of Tests**	**Average Time (ms)**
Rome	3353	8870	1000	15
New York	264346	733846	100	461
USA	1070376	2712798	10	2287

Table 1. Performance of the Algorithm on Different Graphs

Grafurile reprezinta road mapuri a oraselor New York, Rome si a USA. Fisierele le-am luat de pe https://www.diag.uniroma1.it/challenge9/.