Tietorakenteet ja algoritmit lopputyö

Martti Mourujärvi
marttimourujarvi@gmail.com

Ongelman ratkaisu

Lopputyössä tutkittiin matalimman reitin löytämistä verkosta, jossa solmut esittävät kaupunkeja ja kaaret kaupunkien välisen reitin suurinta korkeutta metreissä mitattuna. Lopputyössäni käytin Primin algoritmia minimivirittävän puun löytämiseksi. Primin lisäksi käytin pythonin standardikirjastosta löytyvää bisect-algoritmia pitämään yllä taulukkoa mahdollisista reiteistä pienimmän virittävän puun muodostamisessa. Kun pienimmän virittävän puun muodostamisessa on löydetty polku määränpäähän, lopetetaan puun muodostus ja muodostetaan puun avulla reitti alkupisteestä maaliin ja löydetään samalla reitin korkein kohta.

Ohjelman analyysi

Primin algoritmin avulla pystymme rakentamaan verkon pienimmän virittävän puun, lähtien jostain kaupungista, ohjelmassa virittävää puuta lähdetään rakentamaan aina verkon ensimmäisestä kaupungista (1) lähtien. Primin algoritmi etsii käsiteltävästä kaupungista lähtevät kaaret ja lisää ne taulukkoon, jota tutkimalla löydetään aina lyhin seuraava kaari uuteen verkon solmuun. Toteutuksessa otetaan huomioon jo löydetyt solmut, eikä niitä voida löytää enää uudestaan. Näin vältetään syklien muodostumista. Mahdollisia uusia reittejä edustavaa taulukkoa ylläpidetään bisect-algoritmilla, joka hyödyntää toteutuksessa puolitushakua, “binary-search”. Taulukon ylläpidon asymptoottinen suoritusaika bisect-algoritmilla on O(log n). Primin algoritmia suoritetaan kunnes polku määränpäähän on löydetty tai algoritmia on suoritettu kaupunkien kappalemäärän verran, tällöin reittiä ei löydetä. Primin algoritmi on varsin tehokas löytämään pienimmän virittävän puun verkosta. Ohjelmassani Primin algoritmia suoritetaan vierusmatriisille ja tällöin aikakompleksisuudeksi muodostuu O(V^2).

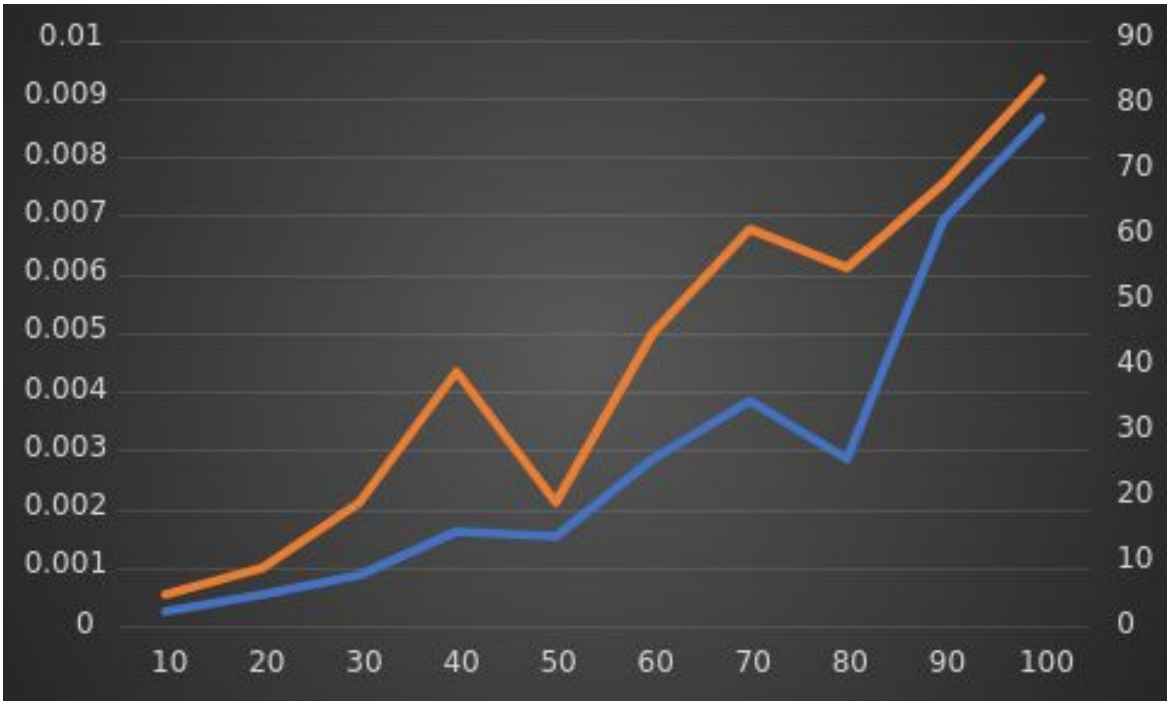
Koko ohjelman kompleksisuus teoriassa : $O(V^2)$

Aika analyysi

Tutkittaessa ohjelman suoritusta saatiin ylös seuraavat tulokset:

graph_testdata

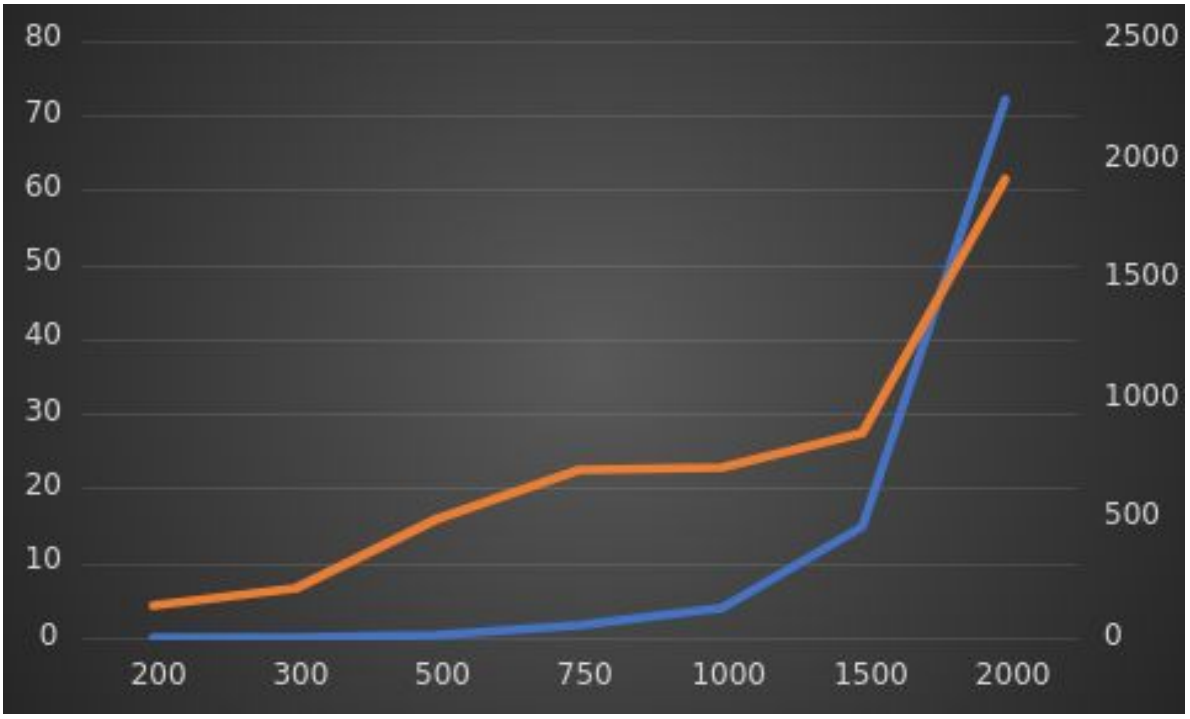
Kaupunkien lkm.	Kesto (s)	Puun solmujen lkm
10	0.000251771	5
20	0.000535011	9
30	0.000885010	19
40	0.001626253	39
50	0.001518011	19
60	0.002829075	45
70	0.003833532	61
80	0.002847672	55
90	0.006943941	68
100	0.008682491	84



Aika sekunteissa - Löydettyjen solmujen lukumäärä

graph_large_testdata

Kaupunkien lkm.	Kesto (s)	Puun solmujen lkm.
200	0.031949997	136
300	0.087661774	208
500	0.363718987	492
750	1.598939266	700
1000	3.833368595	713
1500	14.88755536	859
2000	72.02851901	1923



Aika sekunteissa - Löydettyjen solmujen lukumäärä

Matalista verkoista (< 100 kaupunkia) ohjelma suoriutuu erittäin nopeasti. Sadassa kaupungissa ei kulu sadasosasekuntia aikaa. Graafeista huomataan että ohjelman suoritukseen kuluva aika ei ole suoraan verrannollinen verkon kokoon vaan

määränpäähän mennessä löydettyjen solmujen lukumäärään. Tämä selittää miksi 50 kaupungin verkkoon kuluu vähemmän aikaa kuin 40 kaupungin verkkoon.
Isoista verkoista algoritmi suoriutuu hyvin 500 kaupunkiin asti, jonka jälkeen algoritmi alkaa hidastua huomattavasti. Graafista nähdään, että kun kaupunkien määrä kasvaa, kasvaa myös suoritusaika huomattavasti. Graafin x-akselin välitys saa kasvun vaikuttamaan eksponentiaaliselta, vaikka aika kasvaa enemmänkin neliöllisesti (x^2).

Päättelen taulukoiden perusteella, että algoritmi suoriutuu 15 sekunnissa keskimääräisessä tapauksessa ~1600 kaupunkia sisältävistä verkoista.