Laboration 4

Simon Johansson johsim-0,
Henrik Johansson ehioja-0 ,
Harald Andersson harand-0

Implementation

Klasser

Heap
Klass för våran d-ary heap.

Vertex
Klass för noder i grafen. Har namn, nyckel som är den minsta vikten som den kan anslutas till trädet med samt pekare till noden som den ansluts till trädet via.

Edge
Klass för kanter, innehåller pekare till de två noder som den går mellan samt vikten.

Metoder

prims()
Plockar ut roten från heapen och lägger till i trädet och går sedan igenom alla kanter som är anslutna till roten

relax(vertex,edge)
Uppdaterar vikterna för noderna som kan anslutas till trädet och lägger till nya noder i heapen

Heap.insert(vertex)
Lägger till vertex sist i heapen och flyttar sedan fram om den har ett värde mindre än föräldern

Heap.parentIndex(i)
Ger index för förälder till noden på plats i.

Heap.childs(i)
Ger lista med index för barn till noden på plats i.

Heap.heapify(i)
Uppdaterar heapen från noden på plats i.

Heap.root()
Ger roten och uppdaterar heapen.

Heap.decrease(vertex)
Flyttar noden vertex till rätt plats i heapen efter att den fått ett nytt värde.

Tidskomplexitet

Prims algoritm med D-ary Heap

Då vår prim algoritm baseras på en d-ary min-heap undersökte vi hur d-ary presterar när det kommer till processen att ta bort och lägga till ett element.

När det lägst prioriterade element (a) tas bort från heapen ersätter man det med elementet som har högst prioritering (b). Detta gör man för att (b) ska användas till att balansera heapen. Balanseringen går till på så sätt att (b) jämför sitt egna prioriterings värde med varje element som finns i heapen. Detta kallas för att “klättra neråt”.

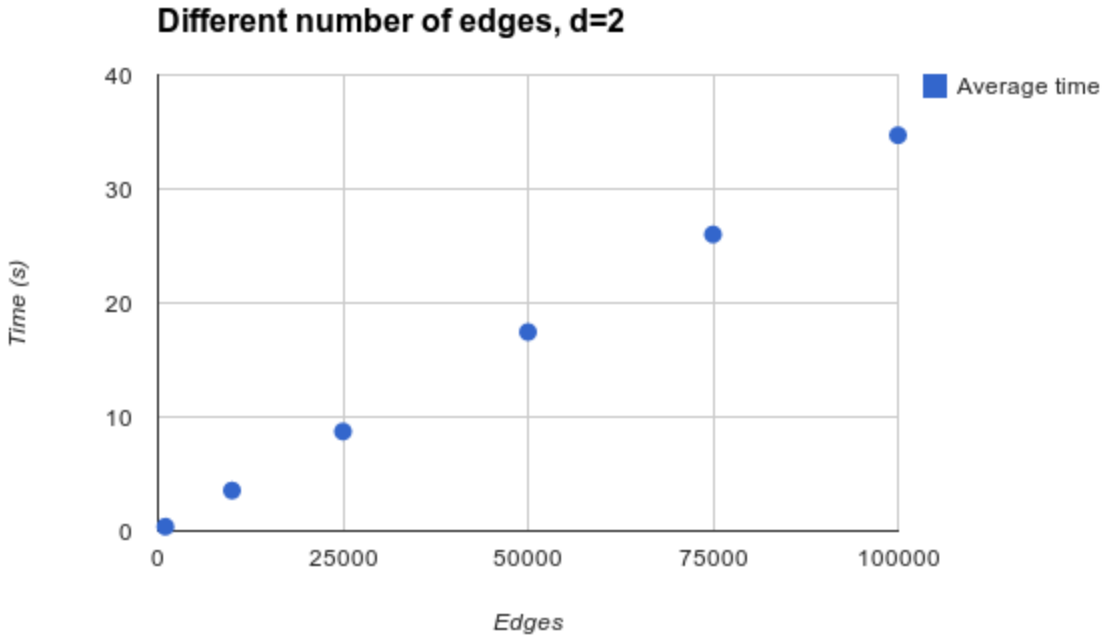
Varje gång ett element (c) ska läggas in i heapen får (c) högst prioritering för att hamna längst ner i heapen. Därefter klättrar (c) uppåt i heapen och jämför sitt prioriterings värdet med varje “förälders” värde. Så här håller det på tills (c) och övriga element hamnar på rätt plats i förhållande till deras vikt. Detta kallas för att “klättra uppåt”.

I båda fallen kan antalet “klättringar” uppgå till $log_d(n) = \frac{log(n)}{log(d)}$ stycken. Då det kan utföras “m” stycken insert procedurer (”klättra upp”, där ett element jämför sig själv med alla “föräldrar” för att hitta sin plats i trädet) så uppgår kostnaden till $O(m\frac{log(n)}{log(d)})$. Vid “n” stycken “klättra ner” måste vi istället jämföra elementet med varje nod och sub nod som finns i trädet, detta tar därför O(d) tid (per “n”). Detta leder till att $O(\frac{log(n)}{log(d)})$ utvecklas till $O(\frac{nd(log(n))}{log(d)})$.

Den totala tids konsumtionen för prims algoritmen kommer ligga på $O(m\ log_d n\ +\ nd\ log_d n)$. Där $O(m\ log_d n)$ syftar till insert operationen och $O(nd\ log_d n)$ till delete operationen för heapen.

Testning

Det första testet som gjordes var för att verifiera att körtiden varierar linjärt med m ($O(m\frac{log(n)}{log(d)})$)
Detta gjordes genom att välja d och n till konstanta värden, 2 respektive 100, och sedan kolla tiden som algoritmen tog att köra för olika värden på m. Resultatet tyder på att sambandet stämmer och kan skådas i följande graf.



Det andra testet som utfördes var att hålla m och n konstanta och variera d för att se vilket som är det optimala värdet för d. Samma listor med noder och kanter skickades in i alla testfallen. Resultaten visar på att det optimala värdet för d ligger vid m/n vilket kan skådas i följande graf.