7 januari 2016

Laboration 3

Grupp 7

D0012E

Jonathan Palm (9502157259)

Andreas Hansson (9506210195)

Johannes Håkansson (9310115119)

Laboration 3 - D0012E - Grupp 7

Jonathan Palm 9502157259*
Andreas Hansson 9506210195†
Johannes Håkansson 9310115119‡

7 januari 2016

1 Inledning och teori

Att kunna hitta den närmsta vägen kan vara mycket användbart. Därför har det dykt upp ett antal algoritmer för att upptäcka just den kortaste vägen till målet. Djikstras algoritm är en välkänd algorim som hittar just denna väg.

Denna laboration kommer att utvärdera Djikstras algoritm både på dess prestanda och komplexitet. Enligt engelskspråkiga Wikipedia ska Djikstras algoritm ha en tidskomplexitet på $O(|E| + |V| \log |V|)$ i värsta fall.

En viktig datastruktur för Djisktras algoritm är en så kallad *heap*. Det är en mycket optimerad typ av *priority queue*. En heap har egenskapen att de kan ta bort det minsta elementet och föra in nya element i heapen mycket billigt. Dessa operationer skulle enbart ta $O(\log_d n)$ (Wikipedia).

2 Metod

För att testa huruvida den teoretiska prestandan hos algorimen så kommer den att implementeras i Python 2.7 och därefter utvärderas mot teorin. Eftersom att djikstras algoritm kräver någon form av kö som tar hänsyn till prioritet så kommer även en *min-heap* att implementeras för att brukas i algoritmen.

*email: paljon-4@student.ltu.se
†email: nadhan-4@student.ltu.se
‡email: johhki-4@student.ltu.se

Testerna kommer att utföras på ett slumpmässigt genererat gridnät av n^2 punkter. Dessa kommer att ha $2n\,(n-1)$ kanter mellan sig. Graferna skulle kunna liknas vid ett rutnät, där varje nod har 2 (hörnen), 3 (längs kanterna), eller 4 grannar (mitten). Det kommer först att göras ett antal tester på olika storlekar på grafen, för att sedan utföras med olika storlekar på heapens största antal barn per nod.

3 Resultat

Resultaten på körningarna av algorimen illustreras av graferna längst bak i rapporten. Figur 1 visar variationen av d-värdet på heapen som används i algoritmen. Den representerar en linjär funktion (med avvikelser) som har en mycket plan lutning. Kortast tid fanns vid 4 barn per nod ($d = 4$, *4-ary heap*) på 1.47418498993 sekunder för att hitta den kortaste vägen för en $8x8$ graf med 117 kanter. Den ökade linjärt i tid och tog som allra längst tid vid 89 barn per nod. Avvikelser uppstod vid vissa mätvärden. Inget medelvärde togs för testet med variation på d.

Resultat på testet vid variation av antalet kanter finns i figur 2. Den lätt kurvade grafen är typisk för en graf på $n \log n$. Det finns avvikande värden, trots att mätvärdet har härletts från medelvärdet av körningen på 3 grafer av den angivna storleken.

4 Diskussion

Genom utförlig analys av koden så visar det sig att den teoretiska komplexiteten inte stämmer in i implementationen som vi utförde. Trots att algoritmen ska ha en teoretisk komplexitet på $O\left(|E| + |V| \log |V|\right)$ så uppvisas det inte genom att kolla på koden. I filen `lab3.py` finns det kommentarer som motiverar hur vi analyserat varje steg av algoritmen.

Den min-heap som vi använder oss av gör att vi kan föra in nya element och ta bort det minsta med en komplexitet på $\Theta\left(\log_d n\right)$ (enligt kursboken), där n är antalet element i heapen och d är hur många barn varje nod i heapen kan ha. Denna komplexitet gör att vi får komplexiteten $|V| \log_d |V|$ för att besöka de relevanta noderna och $|E| \log_d |V|$ för att besöka alla kanterna. Tillsammans med förarbete ($|V|$ tid) och en spårning av den optimala vägen ($|V|$ tid). Detta ger den sammanlagda kostnaden $|V| \log_d |V| + |E| \log_d |V| + 2|V| = O\left(|E| \log_d |V|\right)$ i värsta fall.

Det skiljer sig från teorin i och med att det fanns ”gömda” kostnader för operationerna som ej tidigare togs någon hänsyn till. Om man jämför ana-

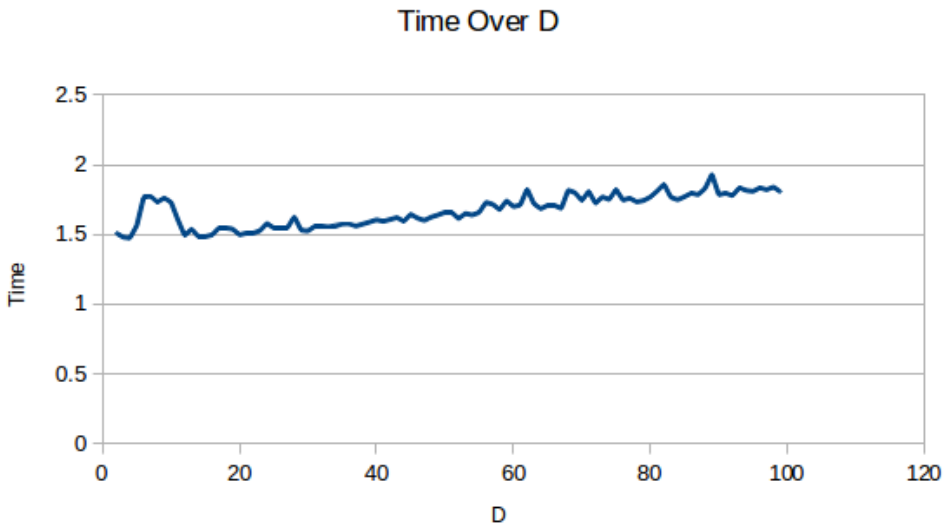
lysen av koden med resultatet på testet av körtiden för variation av antalet kanter så finns det en tydlig korrelation. Figur 2 ser mycket väl ut som en funktion som tillhör $O(|E| \log_d |V|)$.

Mätningen av algoritmens prestanda är svårt vid grafer av olika storlek. Om man inte ser till att man får exempelvis det värsta fallet varje gång så kommer slumpen att påverka mätresultatet. Detta hindrade oss dock inte från att upptäcka en tydlig trend i tillväxten på tiden det tar i förhållande till antalet kanter.

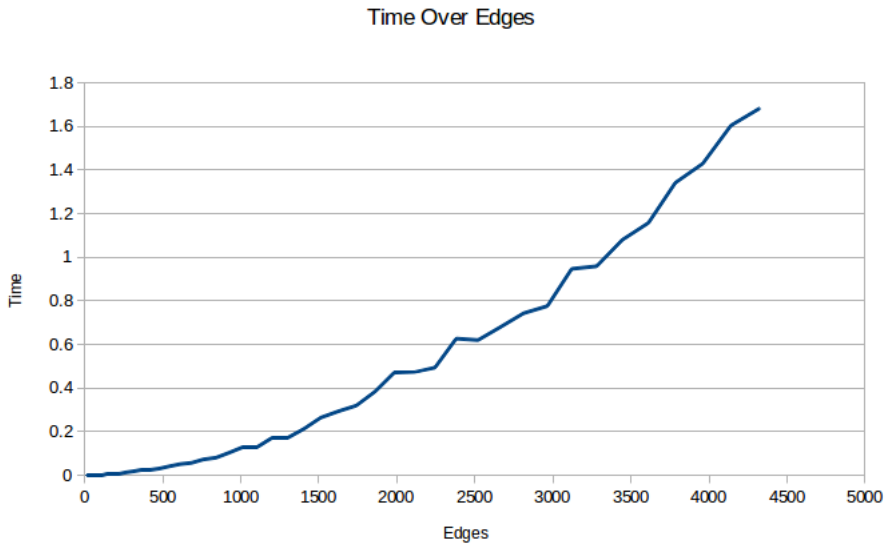
Kostnaden för att föra in saker i en heap är som listat i teoriavsnittet $O(\log_d n)$. Däremot finns det en kostnad för att arrangera om element vid borttagning av det minsta elementet. Eftersom att vi vill bibehålla heapens egenskaper måste vi göra en så kallad *sift*. Denna kostnad är en funktion av d. Så en d-ary heap med n element har en kostnad på $O(\log_d n) + \Theta(d \log_d n)$ för borttagning av det minsta elementet. Ett exempel är en *binary heap*. Där har vi $d = 2$ och då erhålls komplexiteten $O(\log_2 n) + \Theta(2 \log_2 n) = O(\log_2 n)$.

Det som ger en ökning av körtiden när d blir större är just kostnaden för dess sift-operation. När man gör en *delete-min*-operation så måste man gå igenom alla barn till det översta elementet för att avgöra vilket som är minst härnäst. Detta kommer att ta d antal kollar att göra. Sedan måste resten skjutas efter, som också har en komplexitet beroende av d. Den kommer då att köras $\log_d n$ antal gånger.

Vid låga d så går det snabbare just för att den koefficient som d ger till komplexiteten är så pass låg. $d = 4$ verkar vara optimalt för just detta fall. För $d < 4$ verkar det ökade djupet ge en sämre prestanda. Sist men inte minst så gör $d > 4$ att iterationerna på varje steg i heap-operationerna tar onödigt lång tid.



Figur 1: Tid vid variation av d på en graf med 64 noder och 117 kanter. Grafen hålls konstant mellan olika tester.



Figur 2: Tid vid variation av antalet noder och kanter. Eftersom att grafen slumpas varje gång så kan variationer förekomma. Graven är ritad på medelvärdet av 3 körningar med algoritmen.