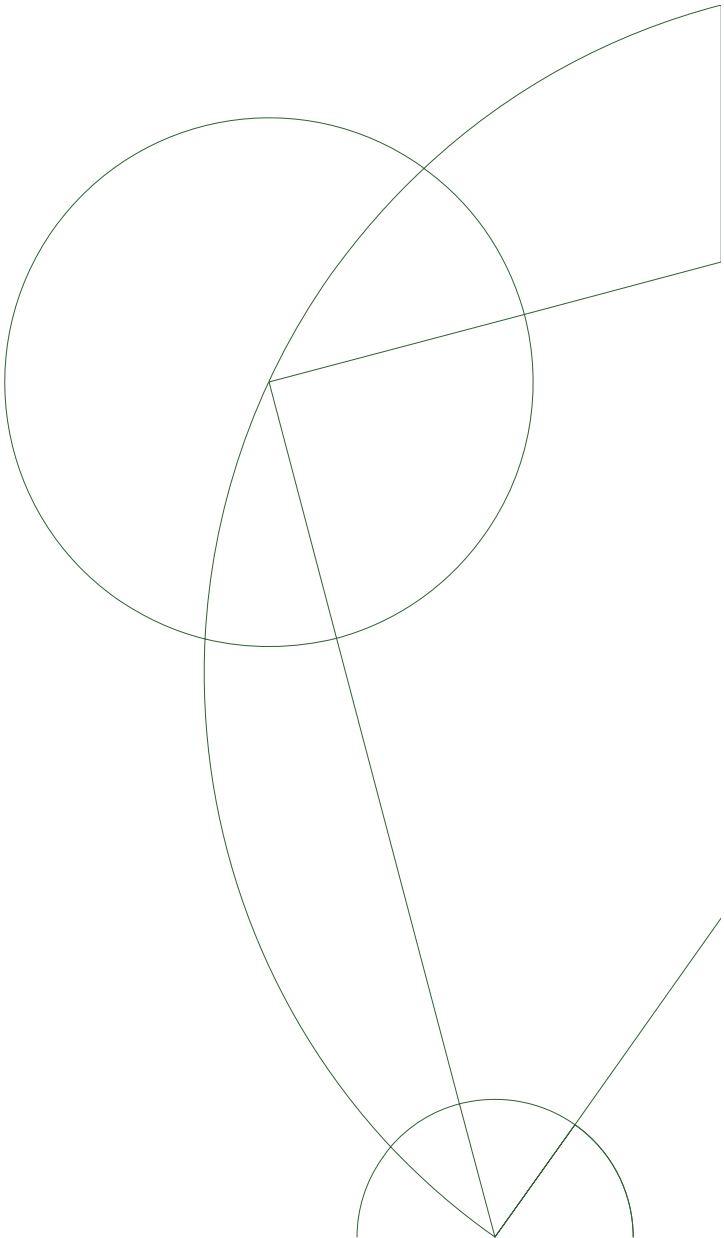
KØBENHAVNS UNIVERSITET

Bachelorprojekt

Allan Nielsen jcl187 og Christian Nielsen bnf287

Bachelorprojekt

Sammenligning af spanner-algoritmer for forskellige probleminstanser



Christian Wulff-Nilsen

13. januar 2016

Indhold

1 Abstract

2 Indledning

En spanner er en subgraf af en sammenhængende graf og indeholder typisk meget færre kanter.
De kan bruges i mange probleminstanser, hvor der er en eller anden form for samlingspunkter (knuder) der er forbundet/relateret til hinanden (af kanter). Dette kunne være den interne infrastruktur af et computer netværk i en virksomhed, omkostningerne af flyruter mellem lufthavne mm.
Om det er flyselskabet der vil spare udgifter ved at reducere antallet af ruter

** Hvad er en spanner og hvorfor er de smarte? * Sparsom til ingen praktisk sammenligning af spanner algoritmer. (se Synopse -> 3. begrundelse) * Hvad bør læser vide inden han/hun læser denne rapport?*

3 Problemformulering

Undersøg og giv et overblik over litteraturen omkring Spanner-algoritmer. Udvælg en delmængde algoritmer og implementer disse, for derefter at sammenligne implementationerne på forskellige probleminstanser og med deres teoretiske grænser.

4 Problemanalyse

Skal vi skrive noget her?

4.1 Litteraturundersøgelse

* Hvorfor er det godt med sådan en?
Hvad skal med? (emner)
Hvor meget skal den dække? (tid)

4.2 Spannere

Lad $G = (V, E)$ notere en sammenhængende graf, der er ikke-orienteret og kan være enten vægtet eller uvægtet. En spanner(*spanning subgraph*) er den mindste udspændende subgraf af G. Afstandene mellem knuderne i H er approksimativt tæt på dem i G. De fejlmarginer på afstanden i H ift. G, kan være enten *additiv* eller *multiplikativ*. måske vi kan komme med et eksempel på, hvordan dette udregnes?

En spanner indeholder alle knuderne og en delmængde af kanterne i den oprindelige graf. For en spanner H af G, vil således gælde at $V(H) = V(G)$ og $E(H) \in E(G)$.

Der findes forskellige slags spannere alt efter, hvad der tillades at input grafen har af egenskaber (uvægtet/vægtet, ikke-orienteret/orienteret). Nogen af disse er f.eks. Kruskal's MST, $(\alpha, \beta) - spanner, r - spanner$

Vores projekt betragter Ingo Althöfers Greedy-Spanner[1], samt den *pre-processing* algoritme, af Mikkel Thorup og Uri Zwicks Approximate Distance Oracles[3], der benytter sig af et sæt knuder og kanter.

Disse to har begge en øvre grænse for, hvad deres stretch bliver. Derfor er det interessant at undersøge, hvorvidt denne grænse bliver presset og hvordan andre parametre opfører sig. Dette kunne f.eks. være densitet og vægt af den fundne spanner

Nedenfor er en forklaring egenskaberne for en spanner:

4.2.1 Egenskaber

For grafer er defineret en række egenskaber. Stretch angiver er forhold mellem to grafer.

- ***Stretch*** beskriver hvor bred/lang en spanner er, i forhold til den oprindelige graf. Dette forhold udregnes ved at finde den længste vej i spanneren og den oprindelige graf.

$$\max_{(u,v \in V),(u \neq v)} \left\{ \frac{d_S(v,u)}{d_G(v,u)} \right\}.$$

  Dette kan enten defineres som det største forhold, eller den gennemsnitlige værdig. Kan dette uddybes?

- ***Densistet*** beskriver grafens fyldighed. Dvs. forholdet mellem antallet af kanter grafen har, og det maksimale antal kanter grafen kan antage (nhvis alle knuder er forbundet med alle andre knuder.) Forholdet er givet ved

$$\frac{|E|}{|E_{max}|} = \frac{|E|}{\frac{n(n-1)}{2}},$$

  hvor n er antal knuder i grafen.

- ***Højeste Grad*** ***Evt. maksimale grad*** antager antallet af kanter, knuder med flest kanter har.

$$\max_{v} |v|,$$

  hvor $|v|$ angiver antallet af kanter knuden har.

- ***Vægt*** beskriver den summen af vægten for alle kanter i grafen.

$$w_G = \sum_{e \in E} e.$$

- ***Køretid*** beskriver tiden det tager for algoritmen at konstruere spanneren.

4.3 Greedy-Spanner

Greedy-spanner, også kaldet *k-spanner*, laver en spanner på en vægtet, ikke-orienteret og sammenhængende graf $G = (V, E)$, hvor V er grafens knudesæt med størrelse $|V| = n$ og E er grafens kantsæt med størrelse $|E| = m$. Derudover benytter algoritmen en *stretch-faktor* $k > 0$.

Den teoretiske køretid for algoritmen er $O(mn^{1+1/k))}$. Og finder en spanner med stretch S, hvor $S \leq k$[1].

Algoritmen følger skridtene:

- Betragt alle grafens kanter E. Disse sorteres efter stigende vægt.
- Der laves en ny graf G', indeholdende alle knuder i G, men ingen kanter. Dette bliver den resulterende spanner.
- Der itereres over E; betragt kanten $e = [u, v]$
  - Den korteste afstand mellem kantens knudepar, $[u, v]$ bestemmes vha. en afstandsfunktion, evt. Dijkstras algoritme.
  - Hvis vægten af e ganget med stretch-faktoren k er mindre end den fundne korteste afstand, tilføjes kanten til spanneren G'.
- Spanneren er fundet, output G'.

4.4 ThorupZwick spanner

Algoritmen køres på en graf $G(V, E)$, hvor V er grafens knudesæt med størrelse $|V| = n$ og E er grafens kantsæt med størrelse $|E| = m$. Derudover tager algoritmen parametren k, et heltal ≥ 1.

Algoritmen har en teoretisk kørselstid $O(kmn^{1/k})$[3]. Den resulterende spanner har en stretch på $O(2k-1)$, og antallet af kanter i spanneren er $O(n^{1+1/k})$[3].

Algoritmen introducerer en række betegnelser:

- **Partitioner:** En delmængde knuder fra den oprindelige graf. Betegnes A_i.
- **Vidner:** Betegnes $p_i(v)$, og beskriver den knude i A_i med mindst afstand til v.

Algoritmen følger skridtene:

1. Lad A_0 indeholde V, og A_k være tom.
2. Lad A_i for $i \in [i, k-1]$, indeholde kanterne i A_{i-1} med sandsynligheden $n^{-1/k}$, sådan at $|A_0| \subseteq |A_1| \subseteq \cdots \subseteq |A_k|$.
3. For alle $i \in [k-1, 0]$:
   (a) For alle $v \in V$:

i. Beregn $\delta(A_i, v)$ - afstanden mellem knude v og partition A_i. Dette kræver at $p_i(v) \in A_i$ findes, sådan at afstandsberegningen bliver $\delta(p_i(v), v)$. Se sektion 4.4.1 for en smart måde at gøre dette på.

(b) For alle $w \in A_i - A_{i+1}$ (for alle knuder ikke i A_{i+1}):

i. Lav et korteste-vej træ fra w,
hvor $C(w) = \{v \in V_0 | \delta(w, v) < \delta(A_{i+1}, v)\}$.
Dvs. afstanden mellem en knude w i træet, og i den oprindelige graf v, skal være mindre end afstanden mellem den næste partition A_{i+1} og v. I dette skridt benyttes en modificeret version af Dijkstas algoritme for at teste denne grænse, forklaret i sektion 4.4.1.

4. Foren alle de korteste-vej træer ved at lave en ny graf $H(V_H, E_H)$ med $V_H = V_G$, og $E_H = \emptyset$. Tilføj kanterne i træerne, sådan at alle kanter der optræder i træerne, tilføjes til den nye graf H èn gang. H er en ThorupZwick spanner.

4.4.1 Bestemmelse af vidner $p_i(v)$

For at bestemme afstanden mellem en knude v og en partition A_i nemmest, introduceres en ny knude, lad denne betegnes kildeknuden s. Der tilføjes en kant mellem kildeknuden, og alle knuder i A_i med vægt 0. Afstands funktionen benyttes til at finde afstanden mellem kildeknuden s og v. Kantsættet som afstandsfunktionen returnerer, vil som knude 2 indeholde den knude i A_i, der er tættest på v.

Bemærk at afstandsfunktionen i praksis finder afstanden fra en knude til alle andre knuder i grafen, hvorfor proceduren beskrevet ovenfor er simplificeret.

Modificeret Dijkstras algoritme Dijkstras algoritme finder den korteste afstand mellem en startknude s, og enhver anden knude v. Algoritmen opretholder en minimums prioritetskø Q. Algoritmen starter ved at sætte afstanden for alle knuder $d(v) = \infty$, og $d(s) = 0$. Så længe Q ikke er tom, henter algoritmen den mindste knude u fra Q (ved første gennemkørsel af løkken, vil dette være s.) Der udføres en relax() funktion på alle us kanter (u,v), hvor $d(v)$ sættes til $\min(d(v), d(u) + l(u, v))$, hvor $l(u, v)$ er afstanden mellem u og v[2].

Modificeringen af Dijkstras algoritme går ud på kun at udføre *relax* skridtet, hvis $d(u)+l(u, v) < \delta(A_{i+1}, v)$. $\delta(A_{i+1}, v)$ er beregnet i den foregående iteration, så dette tjek tager konstant tid, men optimerer køretiden, idet at den kun skal relaxere knuden, hvis tjekket er sandt[3].

4.5 Udvikling af probleminstanser

* Ting man skal være opmærksom på når man generer grafer (kanter) tilfældigt Bestemmelse af antal knuder og kanter og forklaring af densitet ved generering

4.6 Implementering af algoritmer

Vores implementering er foretaget i Python 2.7.6, med GCC 4.8.2 compileren, på Linux Mint.

Implementeringen består af følgende dele:

- **Graf klasse.** Håndterer grafens struktur og metoder. Grafer gemmes vha. en adjacency matrix, implementeret vha. dictionaries[**psf**].

  Klassen understøtter tilføjelse samt fjernelse af knuder og kanter. Ved tilføjelse af en kant mellem to knuder, tilføjes en kant i begge retninger. Ved fjernelse af en knude, fjernes alle knudens kanter fra grafen.

  Der er også implementeret en række metoder til at bestemme grafens egenskaber. Herunder, densitet, højeste grad og vægt.

- **Grafgenererings klasse.** Grafgenereringen sker ved at definere parametrene densitet og antal kanter. Forklar lige din sorte magi :i

- **Dijkstra klasse.** Dijkstras algoritme er en implementation af Dived Eppstein fra UC Irvine[1], med modifikationerne beskrevet i sektion 4.4.1. Eppsteins implementation gør brug af et prioritets dictionary[2], også implementeret af ham, som bruger en binær hob til at gemme knuder i.

- **Greedy-Spanner klasse.** Implementerer Greedy-Spanner efter artiklen *On sparse spanners of weighted graphs*[1].

- **ThorupZwick klasse.** Implementerer preprocessing algoritmen fra artiklen *Approximate Distance Oracles*[3].

- **Eksperiment funktioner.** Ansvarlig for at køre algoritmerne med specifikke parametre. Indeholder derudover logik til at gemme data i en komma seperaret fil, og udregne køretid.

4.7 Eksperimenter

Eksperimenter opstilles ved systematisk ændring af en variabel i input grafen. Herefter kan der analyseres, hvordan tendensen er for netop den parameter og indvirkningen den betragtede variabel har. Der varieres ift:

Til opstilling og analyse af eksperimenter stilles tre-punkts skema op. Dette skema udføres for hver egenskab og variabel par der betragtes.

Estimering Et kvalificeret bud på, hvad eksperimentet vil vise.

Overensstemmelse Hvorvidt resultaten stemmer overens

Begrundelse Bud på årsagen til at dette ikke er tilfældet

[1] http://code.activestate.com/recipes/119466-dijkstras-algorithm-for-shortest-paths/
[2] http://code.activestate.com/recipes/117228/

Dataindsamling

For de genererede grafer, øges parametrene løbende. Der foretages derudover en løbende vurdering af relevante paramtetre i forhold til realistisk køretid. Dvs. overstiger køretiden hvad er realistisk muligt, er maks-grænsen fundet.

Idet TZ og de generede grafer har en tilfældighedskomponent, udfører vi vores eksperimenter flere gange på forskellige datasæt, og så findes gennemsnitsværdien. Dette gøres med henblik på at normalisere den tilfældige del af TZ og datasættet.

5 Litteraturundersøgelse

6 Eksperimenter

Der er udføres eksperimenter for hver spanner egenskab, beskrevet i *Spannere*-afsnittet, og graf varibal.

For graf variable benyttes følgende intervaller:

Densitet $d \in \{0.5, 0.6, \ldots, 1.0\}$.

Knudemængde $v \in \{25, 50, \ldots, 200\}$.

Stretch-faktor $k \in \{2, 3, \ldots, 10\}$.

7 Analyse og fremvisning af data

* grafer/tabeller for udvikling af udvalgte værdier

8 Diskussion af resultater

* Hvordan opfører det sig i forhold til de teoretiske værdier og generelt? (er der f.eks. nogen tendenser)?
Hvordan er de sat op over for hinanden?

9 Konklusion

10 Appendix

10.1 Greedy-Spanner eksempel

10.2 Thorup-Zwick eksempel

10.3 Grafer af data