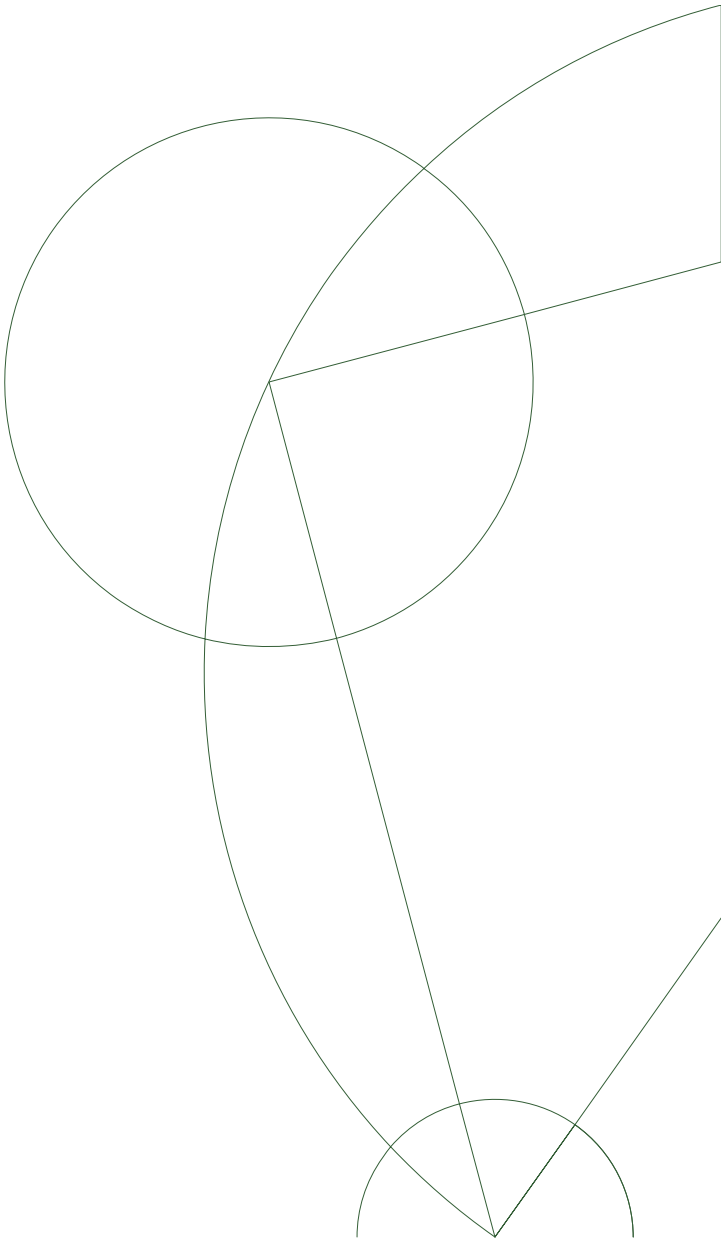
KØBENHAVNS UNIVERSITET

Bachelorprojekt

Allan Nielsen jcl187 og Christian Nielsen bnf287

Bachelorprojekt

Sammenligning af spanner-algoritmer for forskellige probleminstanser



Christian Wulff-Nilsen

26. januar 2016

Indhold

1 Abstract

This paper seeks to compare the spanner algorithms Greedy-Spanner of Ingo Althöfer and the pro-processing algorithm from Mikkel Thorup and Uri Zwick's Approximate Distance Oracle algorithm.

They get compared against their theoretical bounds, as well as to each other.

It is found that Greedy-Spanner creates light spanners, at the cost of a higher runtime, while the spanner of Thorup and Zwick constructs heavier spanners, but in faster time.

2 Indledning

En spanner er en delgraf af en sammenhængende graf. Den indeholder alle knuder i den oprindelig graf, og typisk mange færre kanter. En spanner vurderes typisk på dens stretch. Dvs. den længste *omvej* i forhold til den oprindelige graf. En graf med lav stretch kaldes også for en *sparse spanner*.

Idet spannere er grafer i sig selv, har de samme egenskaber som grafer. Dette inkluderer densitet, som er forholdet mellem det egentlige antal kanter og det maksimale antal kanter. Højeste grad, som er den mængde kanter den knude med flest kanter har, samt vægt, som er summen af alle kanters vægt i grafen.

En spanner bruges i problemer med samlingspunkter (knuder), der er forbundet/relateret til hinanden (kanter). Dette kan eksempelvis være en intern infrastruktur i et computer netværk, omkostninger af flyruter mellem lufthavne, og lignende. Ønskes sådanne strukturer optimeret, således at man stadigvæk har samme dækning (samme knuder), men fjerner dyre veje (tunge kanter), er det oplagt at benytte konstruere en spanner.

Spannere er et aktivt forskningsfelt, og der findes således adskillelige algoritmer til konstruktion af disse. Denne rapport foretager en sammenligning af Ingo Althöfers Greedy-Spanner algoritme (Greedy), og Mikkel Thorup Uri Zwicks *pre-processing* algoritme (ThorupZwick), fra deres Approximate Distance Oracles algoritme.

Begge algoritmer har nogle interessant egenskaber i forhold til hinanden. Greedy har en højere øvre grænse for dens køretid, men leverer altid den spanner, der har færrest kanter. ThorupZwick har modsat en lavere øvre grænse for køretid, men leverer en mere fyldig spanner. Greedy har også en nedre grænse for, hvor let en spanner den kan levere, hvor ThorupZwick ikke har nogen øvre eller nedre grænse.

Begge algoritmer danner det kan kalder en $t - spanner$. Dette betyder at stretchen i spanneren er på netop t. Endvidere så danner ThorupZwick mere nøjagtigt en $(2k-1) - spanner$ og ved at give Greedy samme $(2k-1)$-faktor, kan vi opnå samme stretch grænse. Herved at algoritmerne sammenlignlige.

Grænser	Stretch	Størrelse	Køretid	vægt
Greedy	$O(2k-1)$	$O(n^{1+1/k})$	$O(mn^{1+1/k})$	$O(MST(G)(n/2t)$
ThorupZwick	$O(2k-1)$	$O(kn^{1+1/k})$	$O(kmn^{1/k})$	-

Figur 1: Greedy og ThorupZwicks teoretiske grænser.

Rapporten har til formål at sammenligne disse algoritmer. Der undersøges hvordan algoritmerne lever op til deres grænser, er de tæt på, langt fra? Og er der tilfælde hvor den ene algoritme er favorabel at benytte i forhold til den anden?

Til at opnå dette køres algoritmerne på tilfældige grafer, med fastsatte paramtetre *densitet*, *antal knuder* og for stretch-faktorer k.

Rapporten opnår konklusionerne at Greedy laver markant lettere spannere end ThorupZwick. Dog konstruerer ThorupZwick sin spanner hurtigere end Greedy, som har en lineær stigning som funktion af mængden af kanter i grafen. Der findes endvidere at graden for begge grafer ligger mere eller mindre fast, uanset mængden af kanter.

Dette belyser, at ønsket er at opnå den letteste spanner, skal der konstrueres en Greedy spanner. Ligeledes hvis der ønskes en spanner konstrueret hurtigt, skal ThorupZwick benyttes.

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

4.1 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.

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 $E(H) \subseteq E(G)$.

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](Greedy), samt den *pre-processing* algoritme, af Mikkel Thorup og Uri Zwicks Approximate Distance Oracles[4](TZ), der benytter sig af et sæt knuder og kanter.

Greedy og TZ 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:

4.1.1 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.

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

- ***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

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

- ***Højeste Grad****(grad)* antager antallet af kanter, knuder med flest kanter har.

$$\max_{v\in V} deg(v)$$

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

$$w_G = \sum_{e\in E} \omega(e).$$

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

4.2 Greedy-Spanner

Greedy-spanner algoritmen laver en *k-spanner* af 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* $k > 0$.

Algoritmen har grænsen $O(n^{1+1/k})$ for antallet af kanter i den resulterende spanner, med en stretch på $O(2k-1)$, og kørselstid $O(n^{1+\frac{1}{k}})$.

Denne spanner har også en optimal lightness egenskab. Dette udspringer af, at den er en generalisering af Kruskal's Minimum-Spanning-Tree(MST), hvilket medfører at den letteste spanner er netop MST(G).
Denne egenskab ses i 3.b) nedenfor. Hvis stretch-faktoren får mod ∞, vil en kant $e(u, v)$ aldrig blive tilføjet, hvis der allerede findes en vej mellem u og v i spanneren.[1]

Algoritmen følger skridtene:

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

Se appendix 8.1

4.3 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 $O(kmn^{1/k})$[4]. Den resulterende spanner har en stretch på $O(2k-1)$, og antallet af kanter i spanneren er $O(kn^{1+1/k})$[4].

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 knuderne 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 4.3.1 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 4.3.2.

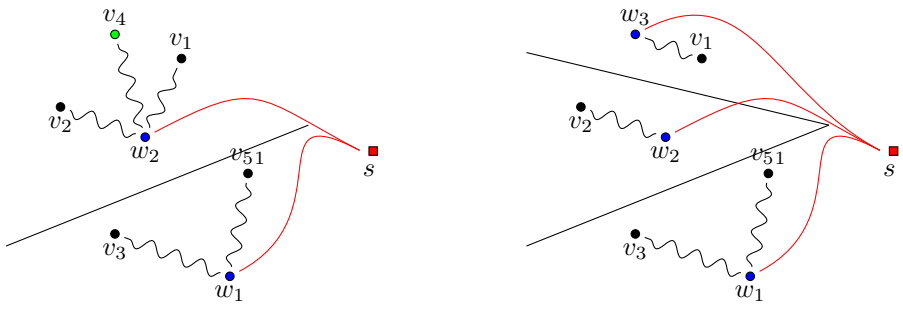
4. Foren alle de korteste-vej træer ved at lave en ny graf $H(V_H, E_H)$ med $V_H = V_G$, og $E_H = \emptyset$. Tilføj kanterne i træerne, sådan at alle kanter der optræder i træerne, tilføjes til den nye graf H èn gang. H er en ThorupZwick spanner.

4.3.1 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, $p_i \in A_i$, 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.

Betragt Figur 1. Den firkantede knude er den nye kildeknude. De cirkelformede knuder er de oprindelige knuder i grafen, hvor deres farve angiver hvilken opdeling A_i de tilhører. Den røde linie mellem kildeknuden s og knuderne $w \in A_i$, angiver den nye kant mellem disse, med vægt 0.

Idet man finder afstanden mellem s og alle $v \in V$, følger det at den første besøgte knude, må være en af knuderne i A_i, idet afstanden er 0. Knuder der besøges på denne måde, betegnes som vidner. Som det fremgår af **??** og **??**, der illustrerer to iterationer af 3.b) skridtet, fremgår det at $A_{i+1} \subseteq A_{i]}$



(a) A_{i+1}'te iteration

(b) A_i'te iteration

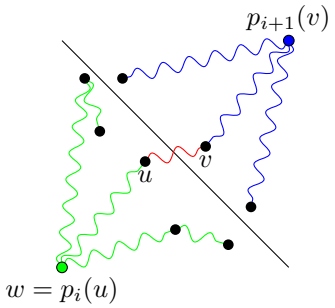
Figur 2: Tilføjelse af kildekunde for at bestemme vidner.

4.3.2 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[3].

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 køre relaxe() på knuden, hvis tjekket er sandt[4].

Den modificerede Dijkstras algoritme benyttes til at konstruere kortestevej træer. Figur 2 illustrer formålet med modificeringen. Hvis $d(u) + l(u, v) < \delta(A_{i+1}, v)$ ikke er sandt, findes der en kortere ved til v, hvorfor (u, v) ikke skal tilføjes til træet.



Figur 3: Hvis $d(u) + l(u, v) < \delta(A_{i+1}, v)$, findes der en bedre vej til v. Den sorte streg angiver skillelinien for de punkter det korteste-vej træ $C(w)$ skal indeholde.

4.4 Generering af probleminstanser

For at sikre at én algoritme ikke bliver favoriseret til fordel for en anden, genereres grafer vha. tilfældighed.

Dette gøres ved, at for alle mulige kanter i grafen, beregnes en sandsynlighed for at denne skal med i grafen.

Der følger heraf, at den resulterende densitet kan variere fra den ønskede. Dette vil udjævne sig når antallet af kanter stiger, idet der er flere muligheder for at opnå den ønskede densitet.
Inden en kant bliver tilføjet til grafen, får den tildelt en tilfældig vægt i form af et heltal mellem $[0; 999]$.
Grafen sikres til slut for at være sammenhængende. Dette gøres ved at køre Dijkstras algoritme fra en vilkårlig knude i grafen, til alle andre knuder i grafen. Er alle afstandende mindre end uendelig, er grafen sammenhængende. Hvis ikke, startes genereringen forfra.

4.5 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[2].

  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. Der genereres det angivne antal kanter og en ny graf bliver instansieret med disse. For en beskrivelse af denne process og dens problemstillinger, se sektion 4.4.

  For at generere grafen gennemløbes alle mulige kanter i grafen, og bliver tilføjet med en given sandsynlighed. Sandsynligheden for at en kant skal tilføjes til grafen, er en funktion af produktet af den ønskede densitet og systemets højeste heltal. Der genereres et tilfældigt tal mellem 0 og systemets højest heltal. Er dette tal mindre end eller lig med sandsynligheden, tilføjes kanten.

- **Dijkstra klasse.** Dijkstras algoritme er en implementation af Dived Eppstein fra UC Irvine[1], med modifikationerne beskrevet i sektion 4.3.2. 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*[4].

- **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.

Implementeringen er vedlagt som .zip fil.

4.6 Eksperimenter

Eksperimenter opstilles ved at betragte en egenskab og systematisk ændre endten en af variablerne i input grafen eller k-faktoren. Herefter kan der analyseres, hvordan en egenskab opfører sig og den indvirkning den betragtede variabel har. Variablerne er densiteten, k-værdien og antal knuder.

Det formodes, at den øvre grænse for størrelsen af TZ, holder for alle $k1$[4]. Det kunne være interessant at se om denne formodning er holdbar eller om nogle af værdier, udover de beviste, kan afkræftes.

For at eksperimenterne kan være så repræsentable som muligt, køres algoritmer adskillige gange på forskellige genererede grafer. Data for både spannere og grafer gemmes.
Ydermere plottes de øvre grænser, hvor det er relevant.

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

4.7 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 Eksperimenter

For variable benyttes følgende intervaller:

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

Knudemængde $v \in \{10, 15, \ldots, 50\}$.

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

Vi har udvalgt følgende eksperimenter et estimat på opførslen beskrives ved eksperimentet.

Stretch(densitet) Der forventes at denne vil stige for både Greedy og TZ.
Greedy vil stige, fordi der generelt vil være flere kanter i grafen, og derved.

Stretch(k) Greedy vil stige grundet den stigende stretch-faktor.
TZ vil have en finere indeling af de sub-optimale træer, men dette vil muligvis ikke have nogen indflydelse på stretch-faktoren.

Vægt(densitet) Der forventes at vægten af spanneren vil falde, idet antallet af kanter i grafen stiger. Dette følger af at der er flere kanter for algoritmerne at vælge, hvorfor der vil være større chance for at finde en kort vej.

Vægt(knuder) I takt med at antallet af knuder stiger, vil der automatisk være flere kanter i spanneren, hvorfor vægten vil stige ligeledes.

Køretid(knuder) Der forventes at begge køretider stiger, men at Greedy har en langt større stigning end TZ.

Grad(k) Der forventes at denne falde for begge. Greedy, fordi den vil blive mindre dense, ved at tillade en større stretch. TZ fordi antallet af partitioner vil stige. Det medfører, at antallet af sub-optimale træer også vil stige.

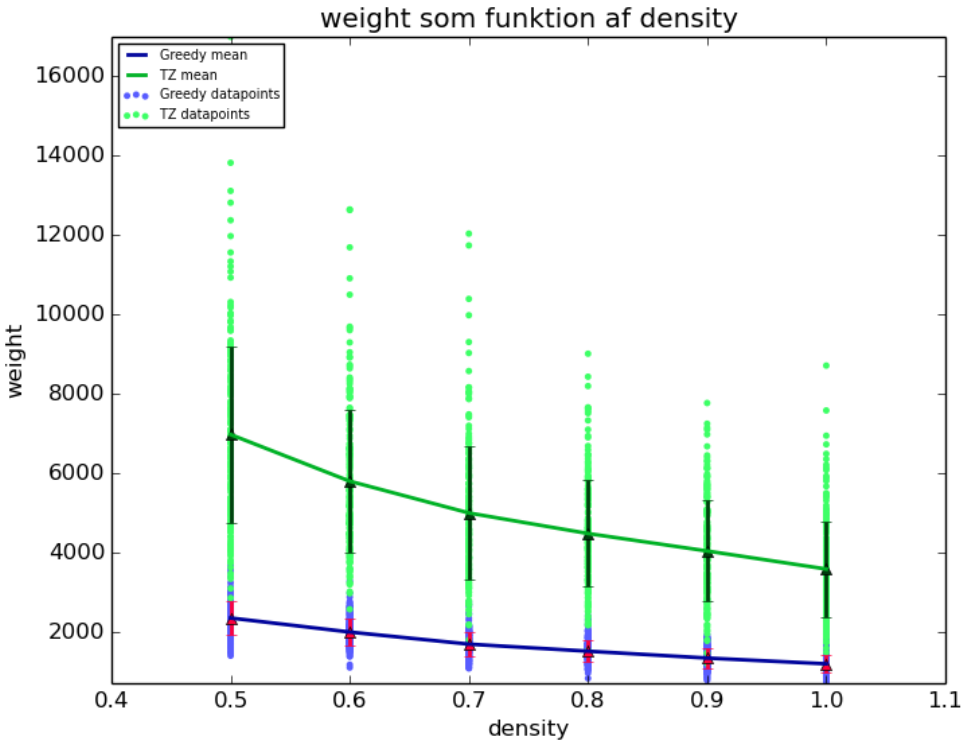
Densitet(k) Der følger samme årsag til estimeringen her for både Greedy og TZ, dog med en forventning om at den vil være aftagende for TZ, da der formodes at når antallet af sub-optimale træer stiger, så falder vægten. Herved kan det være interessant at se på, om det også kan ses på densiteten.

6 Fremvisning af data, Analyse og diskussion

I denne sektion vil der blive vist og forklaret adskillige grafer. Disse er valgt tilfældigt, da de viser en generel tendens på tværs af de faste parametre. Eksempelvis, hvis der vises, hvordan degree afhænger af densiteten, med k og antal knuder som faste parametre, så vil tendens være den samme ved andre faste k og/eller antal knuder.

Der er i graferne også anført error-bars, for at kunne identificere hvor konsistent algoritmerne performer. For at mindske den statistiske usikkerhed, er der indsamlet 200 datapunkter for hver data instans. For flere grafer, med forskellige faste parametre, se appendix 8.2.

Vægt



Figur 4: Vægt som funktion af densitet. $k = 4$, $|V| = 35$, antal kørsler per densitet: 200.

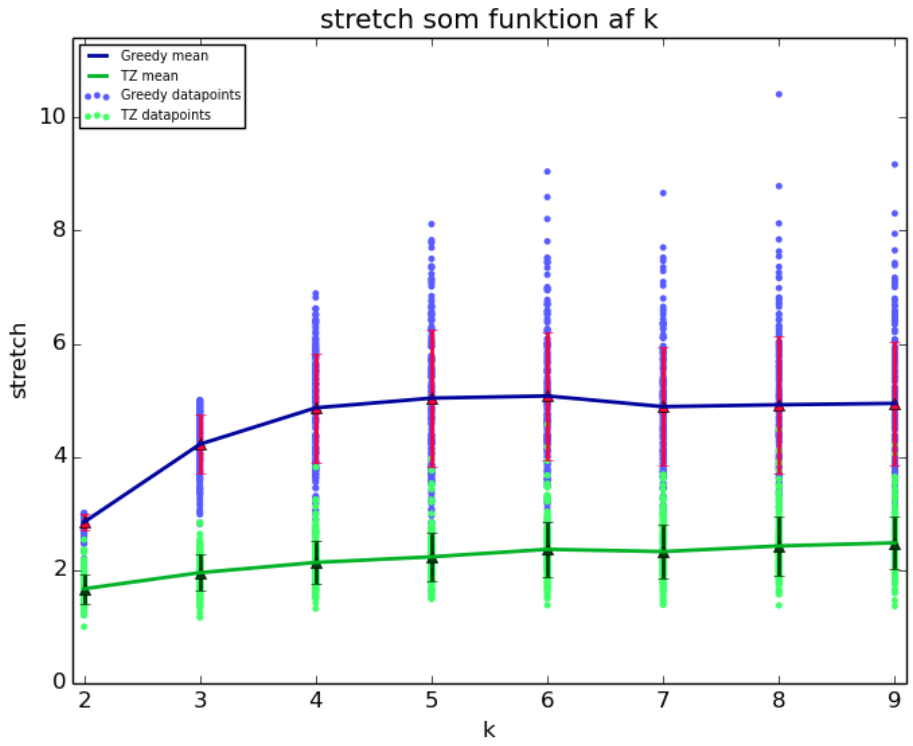
Figur 4 viser den resulterende vægt som funktion af densitet. Figuren viser kørsler for $k = 4$ og $|V| = 35$. Kørsler med andre værdier viser lignende tendenser,

som kan ses i appendix 8.2.

Der ses at vægten falder i takt med at densiteten stiger. Når densiteten stiger, stiger mængden af kanter i grafen (jvf. 4.1.1 (densitet)). Forventningen om at vægten vil falde idet der er flere kanter at vælge imellem, holder stik.

For alle variable (k, densitet og antal knuder) har ThorupZwick spannere en gennemsnitlig højere vægt end Greedy spannere. ThorupZwick danner kortestevej træer, og ovenstående resultat er en indikation på, at der er mange kanter i disse træer som kunne have været undværet, hvis ønsket var en lettere spanner.

Stretch

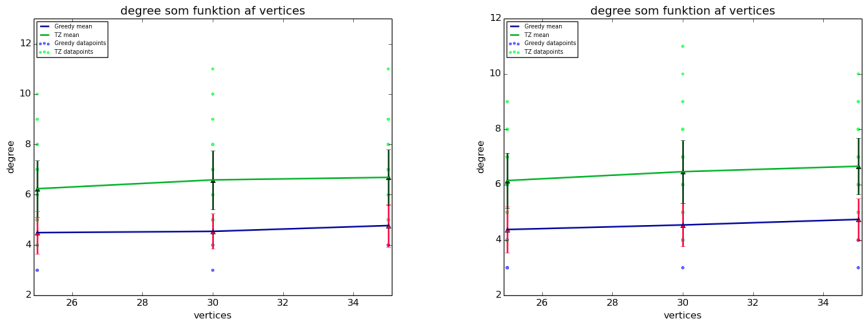


Figur 5: Stretch som funktion af k. $d = 0.8$, $|V| = 40$.

ThorupZwicks stretch er ligeledes stigende, dog i mindre grad sammenlignet med Greedy. ThorupZwicks stigning af mere konsistent, og med lav varians. I de første iterationer af ThorupZwick, hvor partitionerne er små, danner den træer med mange kanter. Under foreningen af disse, tilføjes alle kanter til spanneren, hvorfor ThorupZwick spanneren har betydeligt flere kanter, sammenlignet med

en Greedy spanner. Idet der er flere kanter, bliver sandsynligheden for at finde en god vej bedre, hvorfor stretch også er mindre.

Grad



(a) $k = 9$, $d = 0.5$ (b) $k = 9$, $d = 1.0$

Figur 6: Grad som funktioner af antal knuder.

Ved konstant k og densiteter $d = 0.5$, $d = 1.0$ observeres der en nogenlunde ens stigning. Mængden af kanter har således ingen påvirkning på graden af spanneren.

Denne tendens gør sig gældende for alle kombinationer af parametre. Se appendix 8.2 for illustrationer af dette.

Variansen af graden forbliver ens idet parametren stiger. Dette betyder at der ikke opnås en mere præcis grad, ved at øge datamængden (fx antal kanter).

Køretid

I figur 7a ses hvordan køretiden for Greedy stiger, idet antallet af kanter stiger. Mens den for ThorupZwick forbliver mere eller mindre konstant.

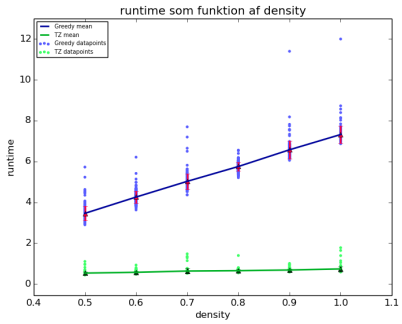
I figur 7b ses samme tendens, idet antallet af knuder stiger.

I figur 7c ses at k ikke har nogen videre påvirkning på køretiden. Den stiger marginalt for ThorupZwick, mens den for Greedy har en høj varians.

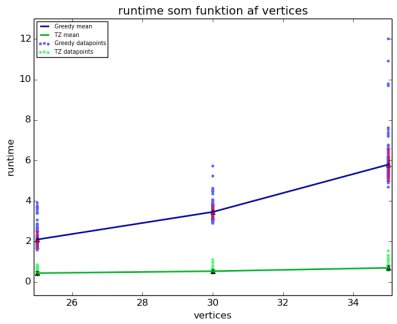
Køretiden for Greedy og ThorupZwick er givet ved hhv. $O(mn^1 + 1/k)$, $O(kmn^{1/k}$. Den direkte indvirkning antallet af knuder og kanter har på den teoretiske køretid, illustreres af resultaterne fra vores eksperimenter.

I figur 7b antydes der en polynomiel udvikling, hvilket følger af køretidens form ax^b, når antallet af kanter og eksponenten holdes fast og antal knuder er varierende.

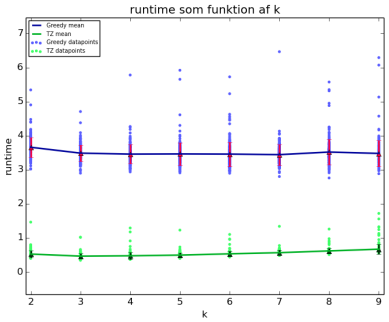
En indlysende, men interessant detalje er, hvis man lader $k \to n$, så vil de to øvre grænser være ens.



(a) $k = 6$, $|V| = 30$



(b) $k = 6$, $d = 0.5$



(c) $|V| = 30$, $d = 0.5$

Figur 7: Kørselstid for Greedy og ThorupZwick.

6.1 Densitet

Der ses i figur 8 at densiteten af de resulterende spannere falder idet k stiger. Endvidere ses der at Greedy lavere mere sparse grafer end ThorupZwick. Greedys styrke er netop af finde sparse spannere[1, pp.3].

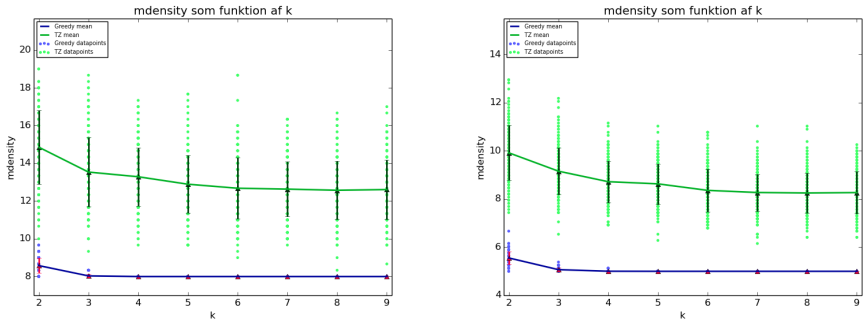
Greedy når den nedre grænse for lightness $|E_{min}| = n - 1$. Dette udregnes ved at indsætte E_{min} i udtrykket for densiteten:

$$\frac{2E_{min}}{n(n+1)} = \frac{2(n-1)}{n(n+1)} = \frac{2}{n}.$$

Lad $n = |V| = 25$ fra figuren, og indsæt dette i udtrykket ovenfor:

$$\frac{2}{n} = \frac{2}{25} = 0.08.$$

Dette svarer til de 8% som Greedy netop ligger sig op ad.



(a) $d = 0.5$, $|V| = 25$ (b) $d = 0.5$, $|V| = 40$

Figur 8: Spanner densitet som funktion af k.

7 Konklusion

Resultaterne af eksperimenterne belyser at Greedy laver lettere spannere end ThorupZwick. ThorupZwick har fordelen med en hurtigere køretid, og den ekstra mængde kanter i ThorupZwick spanneren, bevirker at den stretch er lavere.

Graden af begge grafer er mere eller mindre konstant.

Der kan derfor opstillings følgende fordele/ulemper:

Greedy: Konstruerer den letteste spanner, på bekostning af kørselstid og stretch.

ThorupZwick: Konstruerer en tung spanner hurtigt, og med lav stretch.

Referencer

[1] Ingo Althöfer m.fl. „On sparse spanners of weighted graphs“. I: *Discrete Computational Geometry* 9.1 (1993), s. 81–100. DOI: 10.1007/BF02189308. URL: http://dx.doi.org/10.1007/BF02189308.

[2] Python Software Foundation. *Python Patterns - Implementing Graphs*. 1998. URL: https://www.python.org/doc/essays/graphs/ (sidst set 13.01.2016).

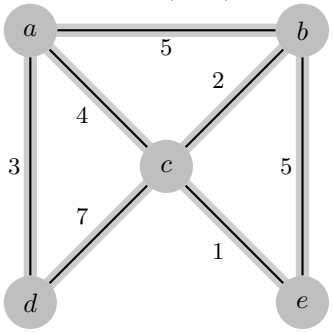
[3] Ronald L. Rivest Clifford Stein Thomas H. Comrne Charles E. Leiserson. *Introduction to Algorithms*. 3. udg. The MIT Press, 2009.

[4] Mikkel Thorup og Uri Zwick. „Approximate Distance Oracles“. I: *J. ACM* 52.1 (2005), s. 1–24.

8 Appendix

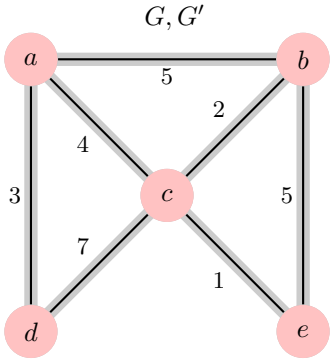
8.1 Greedy-Spanner eksempel

En gennemkørsel af Greedy-Spanner algoritmen vil blive udført nedenfor. Der vil blive vist, hvad der sker løbende for hver iteration. Endvidere vil spanneren blive visualiseret med rød farve oven i input grafen, for at understrege at det er en subgraf af samme, bestående af alle knuder og en delmængde af kanterne i den originale graf. Lad $G = (V, E)$ og $k = 3$ placement G



V	E
a	$\omega(a,b) = 5$
b	$\omega(a,c) = 4$
c	$\omega(a,d) = 3$
d	$\omega(b,c) = 2$
e	$\omega(b,e) = 5$
	$\omega(c,d) = 7$
	$\omega(c,e) = 1$

Vi følger framgangsmåden fra afsnit 4.2.
1. Kanterne E sorteres og lader disse være E_{sorted}.
2. Der laves en ny graf $G' = (V, \emptyset)$



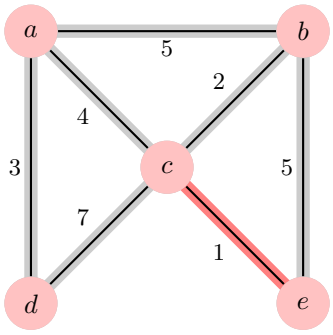
E_{sorted}
$\omega(c,e) = 1$
$\omega(b,c) = 2$
$\omega(a,d) = 3$
$\omega(a,c) = 4$
$\omega(b,e) = 5$
$\omega(a,b) = 5$
$\omega(c,d) = 7$

3. Der vil nu itereres over alle kanter i E_{sorted} og de vil blive streget ud, som de bliver betragtet.
Den letteste kant bliver betragtet og der undersøges, hvorvidt

$$k \cdot \omega(c,e) < \delta_{G'}(c,e) \Rightarrow 3 \cdot 1 < \infty$$

Testen er sand, så kanten tilføjes til G'

G, G'



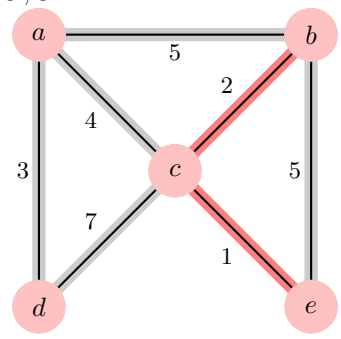
E_{sorted}
~~$\omega(c,e) = 1$~~
$\omega(b,c) = 2$
$\omega(a,d) = 3$
$\omega(a,c) = 4$
$\omega(b,e) = 5$
$\omega(a,b) = 5$
$\omega(c,d) = 7$

Der undersøges:

$$k \cdot \omega(b,c) < \delta_{G'}(b,c) \Rightarrow 3 \cdot 2 < \infty$$

Dette er sandt, så kanten tilføjes til G'.

G, G'

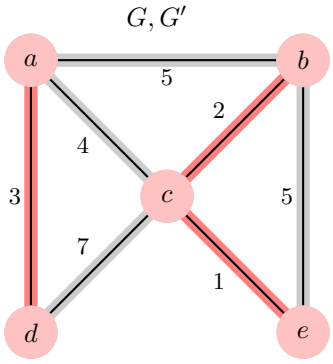


E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
$\omega(a,d) = 3$
$\omega(a,c) = 4$
$\omega(b,e) = 5$
$\omega(a,b) = 5$
$\omega(c,d) = 7$

Der undersøges:

$$k \cdot \omega(a,d) < \delta_{G'}(a,d) \Rightarrow 3 \cdot 3 < \infty$$

Dette er sandt, så kanten tilføjes til G'.

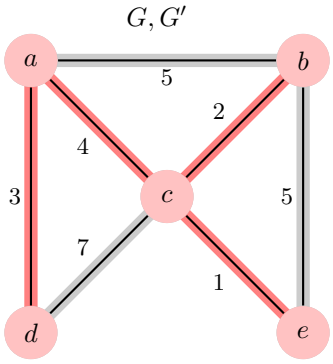


E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
~~$\omega(a,d) = 3$~~
$\omega(a,c) = 4$
$\omega(b,e) = 5$
$\omega(a,b) = 5$
$\omega(c,d) = 7$

Der undersøges:

$$k \cdot \omega(a,c) < \delta_{G'}(a,c) \Rightarrow 3 \cdot 4 < \infty$$

Dette er sandt, så kanten tilføjes til G'.

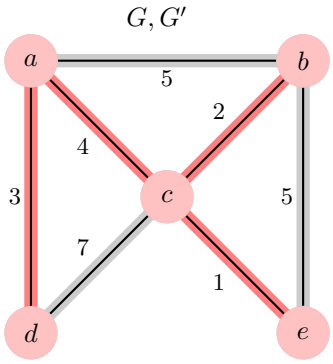


E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
~~$\omega(a,d) = 3$~~
~~$\omega(a,e) = 4$~~
$\omega(b,e) = 5$
$\omega(a,b) = 5$
$\omega(c,d) = 7$

Der undersøges:

$$k \cdot \omega(b,e) < \delta_{G'}(b,e) \Rightarrow 3 \cdot 5 < 3$$

Dette er ikke sandt, så kanten tilføjes ikke til G'.

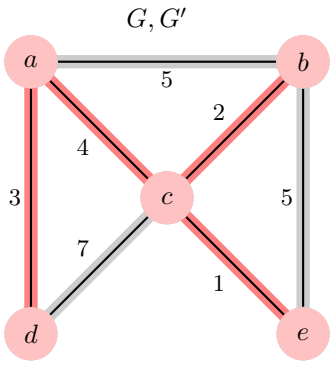


E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
~~$\omega(a,d) = 3$~~
~~$\omega(a,e) = 4$~~
~~$\omega(b,e) = 5$~~
$\omega(a,b) = 5$
$\omega(c,d) = 7$

Der undersøges:

$$k \cdot \omega(a,b) < \delta_{G'}(a,b) \Rightarrow 3 \cdot 5 < 6$$

Dette er ikke sandt, så kanten tilføjes ikke til G'.

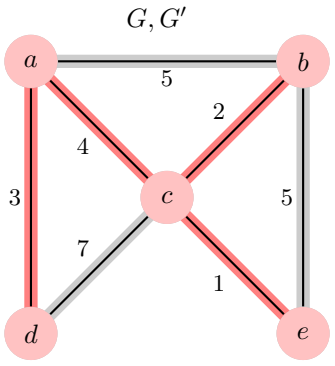


E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
~~$\omega(a,d) = 3$~~
~~$\omega(a,e) = 4$~~
~~$\omega(b,e) = 5$~~
~~$\omega(a,b) = 5$~~
$\omega(c,d) = 7$

Der undersøges:

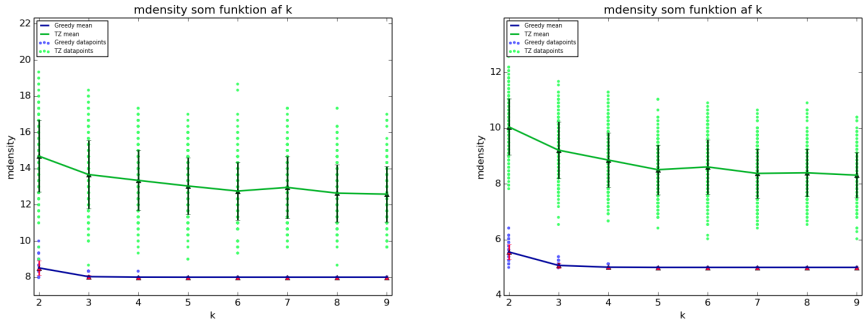
$$k \cdot \omega(c,d) < \delta_{G'}(c,d) \Rightarrow 3 \cdot 7 < 7$$

Dette er ikke sandt, så kanten tilføjes ikke til G'.
Der er ikke flere kanter tilbage i E_{sorted} og algoritmen terminerer.
4. 3-spanneren, er herved fundet i form af G'(rød)



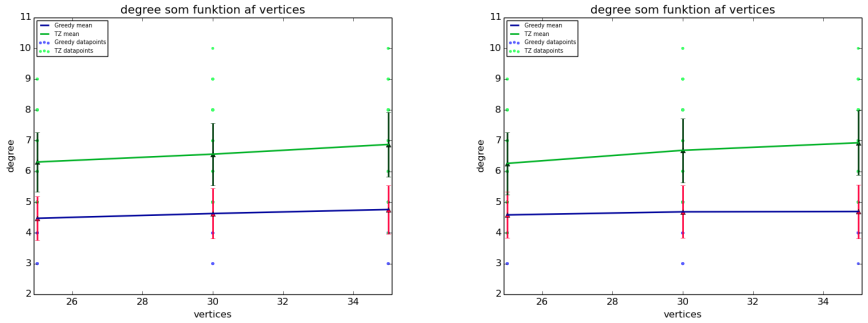
E_{sorted}
~~$\omega(c,e) = 1$~~
~~$\omega(b,e) = 2$~~
~~$\omega(a,d) = 3$~~
~~$\omega(a,e) = 4$~~
~~$\omega(b,e) = 5$~~
~~$\omega(a,b) = 5$~~
~~$\omega(c,d) = 7$~~

8.2 Grafer af data



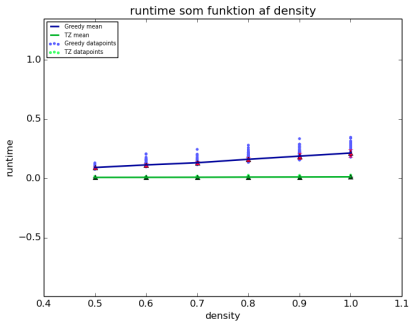
(a) $d = 1.0$, $|V| = 25$ (b) $d = 1.0$, $|V| = 40$

Figur 9: Spanner densitet som funktion af k.

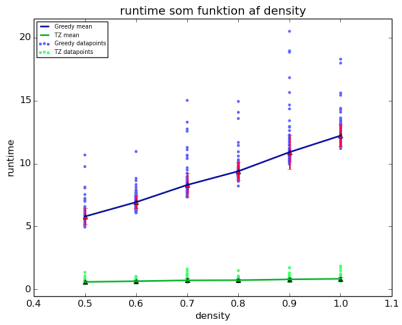


(a) $d = 0.7$, $k = 6$ (b) $d = 0.8$, $k = 4$

Figur 10: Spanner degree som funktion af knuder

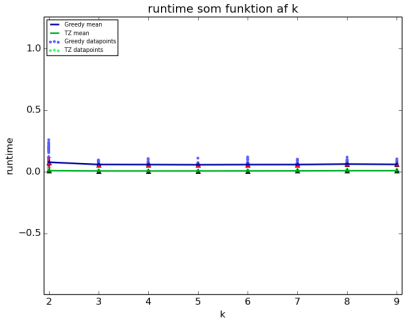


(a) $k = 3$, $|V| = 40$

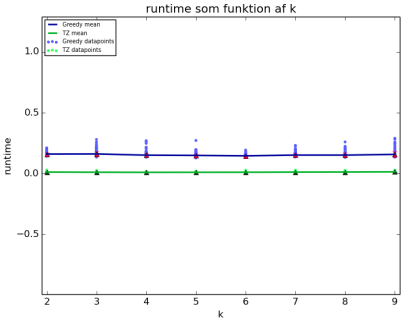


(b) $k = 4$, $|V| = 35$

Figur 11: Spanner køretid som funktion af densitet.

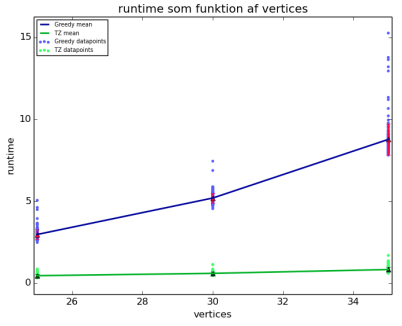


(a) $d = 0.5$, $|V| = 35$

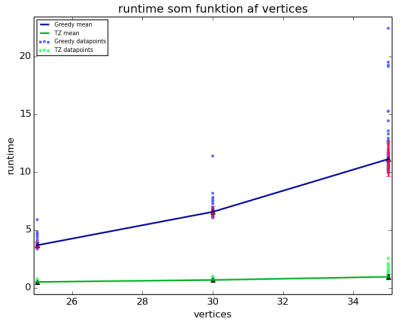


(b) $d = 0.8$, $|V| = 40$

Figur 12: Spanner køretid som funktion af k.

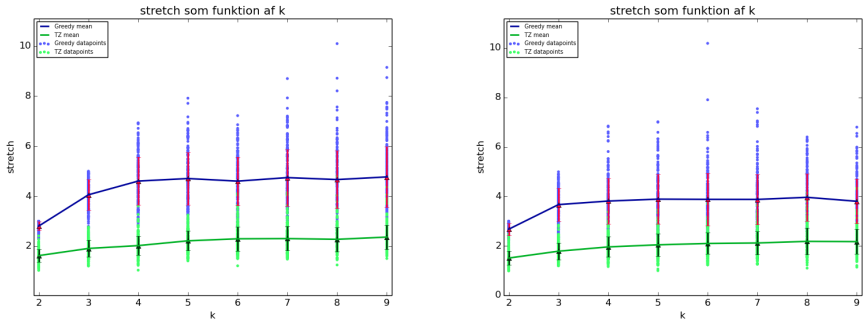


(a) $d = 0.7$, $k = 2$



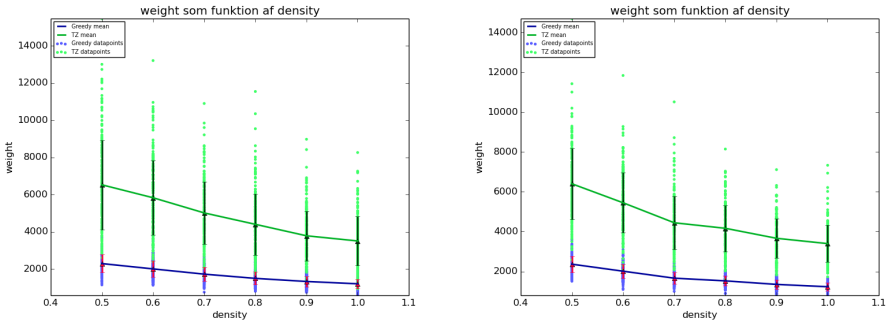
(b) $d = 0.9$, $k = 6$

Figur 13: Spanner køretid som funktion af $|V|$.



(a) $d = 0.6$, $|V| = 35$ (b) $d = 0.5$, $|V| = 25$

Figur 14: Spanner stretch som funktion af k.



(a) $k = 3$, $|V| = 25$ (b) $k = 8$, $|V| = 40$

Figur 15: Spanner vægt som funktion af densitet.