Norges miljø- og biovitenskapelige universitet
Fakultet for miljøvitenskap og teknologi
Institutt for matematiske realfag og teknologi

Masteroppgave 2014
30 stp

Ruteberegning med trafikkdata

Einar Bjørkaas Helle

Ruteberegning med trafikkmeldinger

Einar Bjørkaas Helle
NMBU - Norges miljø- og biovitenskaplige universitet

14. mai 2014

Sammendrag

Det finnes i dag lite tilgjenglige tjenester for å integrere nåværende og planlagte trafikkmeldinger i korteste-vei søk for persontrafikk. Dette til tross for at både vegnett og trafikkmeldinger er offentlig tilgjengelig i Norge.

Denne oppgaven ser på muligheter for benytte data fra Nasjonal vegdatabank (NVDB) sammen med trafikkmeldinger fra Statens Vegvesen for å gjøre tidsavhengige korteste-vei søk i pgRouting.

Mulige måter å koble datasettene blir utforsket, og en ser på måter å implementere en løsning i pgRouting. Prinsippet blir deretter testet i programmeringsspråket Julia.

Oppgaven viser en del utfordringer ved å gå fra trafikkmeldinger gitt ved egenkoordinater til noder og kanter i vegnettsdataene fra NVDB. Det blir også vist at kompleks tidsinformasjon gjør implementering vanskelig, samt at resultatet blir svært følsomt for endringer i avreisetid.

Abstract

There are today few available services to integrate current and forecast traffic information in shortest path search for road traffic. This is despite the fact that both topological road data and traffic information is publicly available in Norway.

This thesis examines posibilities of using data from the Norwegian National road database (NVDB) together with traffic data from The Norwegian Public Roads Administration (NPRA) to do time-dependent shortest path search in pgRouting.

Possible methods to connect the data sets, and how to implement a solution in pgRouting, are explored. The principle is then tested in the programming language Julia.

The thesis points to challenges in transforming traffic data given in coordinates to a system of edges and vertices. It is also shown that complex time information makes implementation difficult, and the result is very sensitive to changes in departure time.

Forord

Denne masteroppgaven utgjør avslutningen på min utdanning innen Geomatikk ved Norges miljø- og biovitenskaplige universitet (NMBU).

Ønsker å takke veileder Håvard Tveite og mine medstudenter ved Geomatikkstudiet på NMBU for godt støtte og hjelp.

NMBU, Ås 14. mai 2014

Einar Bjørkaas Helle

Innhold

Figurer

Kapittel 1

Introduksjon

1.1 Bakgrunn for oppgaven

Det finnes i Norge i dag fritt tilgjengelige data for vegnettverk og trafikkmeldinger fra Statens Vegvesen. Til tross for dette er det vanskelig å finne tjenester hvor trafikkinformasjon som forsinkelser, stengte veier og lignende er godt integrert med ruteberegning.

Mange gode tjenester finnes for å benytte "crowdsourced" informasjon for å beregne kjøretid på live data, som kan være nyttig i svært trafikkerte områder. Derimot finnes det lite gode tjenester som inkluderer fremtidige data som planlagt stengning av vei i forbindelse med vegarbeid eller lignende i selve beregningen.

1.2 Problemstilling

Hvilke muligheter finnes for å integrere vegdata fra Norsk Vegdatabase (NVDB) med trafikkmeldinger fra Statens Vegvesen (SVV) i en korteste-vei ruteberegning. Fokus vil være på bruk av Open-Source programvare som PostGis/PostgreSQL og pgRouting.

1.3 Metode og oppbygging

Jeg vil først gå gjennom en del relevant teori for emnet. Herunder databaser; relasjonsbaserte og romlige, ruteberegningsprogrammet pgRouting, FIFO nettverk, dijkstras algortime, samt litt om lagringsformatet XML.

Deretter vil jeg se på tidligere arbeid med å integrere tidsavhengige data i pgRouting og gjøre en vurdering på hvordan videre arbeid kan bygge på det.

Datasettet Vegnett i NVDB (Nasjonal Vegdatabank), og Statens Vegvesen sin tjeneste for levering av trafikkinformasjon vil bli gjennomgått i kapittel 4.

Kapittel 5 ser på en del problemstillinger rundt det å gå fra koordinater i trafikkmeldinger til noder og kanter i et topologisk nettverk.

Kapittel 6 ser på metoder for å integrere tidsavhengige kostnader i et FIFO-nettverk.

Kapittel 7 viser en kort test av konseptet.

Diskusjon og konklusjon kommer i kapittel 8.

1.4 Verktøy som er benyttet

- **PostgreSQL / Postgis** - Databaseverktøy
- **pgRouting** - Ruteberegningsutvidelse til PostgreSQL
- **pgAdmin** - Administrativt verktøy for å jobbe med PostgreSQL
- **qGIS** - v. 2.0.1-Dufour - for visualisering av geografiske data
- **GDAL** - verktøy for geografiske data
- **proj4** - bibliotek for konvertering mellom ulike projeksjoner.

- **Julia** - programmering
- **Python** -programmering
- **Inkscape** - for vektorgrafikk
- **Gimp** - for rastergrafikk

1.5 Forkortelser/Begreper

- **NVDB** - Nasjonal vegdatabank
- **SRS** - Spatial reference system
- **CRS** - Coordinate reference system
- **EPSG** - European Petroleum Survey Group
- **SRID** - Spatial Reference System Identifier
- **XML** - Extensible Markup Language
- **Datex** - Standard for DATa EXchange innen trafikk
- **Crowdsourcing** - Informasjon samlet fra stor gruppe brukere
- **SQL** - Structured Query Language
- **DBMS** - Databasehåndteringsystem
- **RDBMS** - Romleg databasehåndteringsystem
- **ORDBMS** - Objektorientert, romleg databasehåndteringsystem
- **FIFO** - First Inn, First Out

Kapittel 2

Teori

2.1 Databaser

Databaser er i følge Store Norske Leksikon et "arkiv der data lagres på elektroniske medier og gjøres enkelt tilgjengelig for autoriserte brukere gjennom særskilte programmer " [12] Disse særskilte programmene det refereres til kalles på engelsk Database Management Systems (DBMS). En norsk oversettelse kan være databaseapplikasjoner eller databasehåndteringsystem.

2.2 Relasjonsbaserte databaser

Den mest utbredte formen for databaser i dag er relasjonsdatabasen. Denne baserer seg på relasjonsmodellen av Edgar F. Codd. [41] En relasjonsdatabase kan kanskje forklares best som et sett med tabeller som kan kobles sammen gjennom et sett med nøkkelverdier. På denne måten slipper man mye dobbeltlagring som ville ellers oppstått.

Et standard eksempel kan være en tabell over studenter som er meldt til undervisning i et fag kan lagres med fagkode (og ev. årstall) som primærnøkkel, og studentnummer som fremmednøkkel. Denne vil da kunne kobles til en annen tabell som har studentnummer som primærnøkkel, og informasjon om studenten i de andre kolonnene (adresse, telefonnummer, o.l.) På den-

ne måten slipper en at all informasjon om studenten må lagres på hvert fag han eller hun er oppmeldt i.

2.2.1 PostgreSQL

PostgreSQL er et ORDBMS (objekt-relasjonsbasert databasehåndteringsystem). Dette ligner mye på et vanlig relasjonsbasert databasehåndteringsystem, men med en objekt-orientert modell i bunnen som gir støtte for objekter og klasser.

PostgreSQL har lange røtter helt tilbake til Ingres prosjektet ved Berkley (UCB) ved starten av -80 tallet. Prosjektet het i starten Postgres som i etter gres. I 1996 fikk prosjektet sitt nåværende navn og har blitt drevet som et open source prosjekt siden den gang. [39].

2.2.2 SQL

SQL står for Structured Query Language og er det mest vanlige grensesnittet for operasjoner mot en relasjonsdatabase, og det er et spesialdesignet for denne bruken. SQL er stort sett et deklarativt språk, men med enkelte prosedyrebaserte element. [42] Det at et språk er deklarativt betyr at man beskriver mer hva man vil resultatet skal bli (deklarativt), heller en hvordan man skal gå fram for å oppnå det(imperativ). Deklarativ programmering blir ofte sett på som renere, og det kan hjelpe til å abstrahere vekk en del av maskineriet i databasehåndteringsystemet.

Noen ganger blir operasjoner mot en database ganske komplekse, og det er vanskelig å samle alt i en spørring. I disse tilfellene kan det være greit å kunne samle flere spørringer i en prosedyre. PostgreSQL har her sin egen prosedyrebaserte variant av SQL: PL/pgSQL. Dette gir muligheter for å gruppere flere spørringer inne i databaseserveren og dermed slippe mye overhead ved kommunikasjon mellom server og klient. [23]

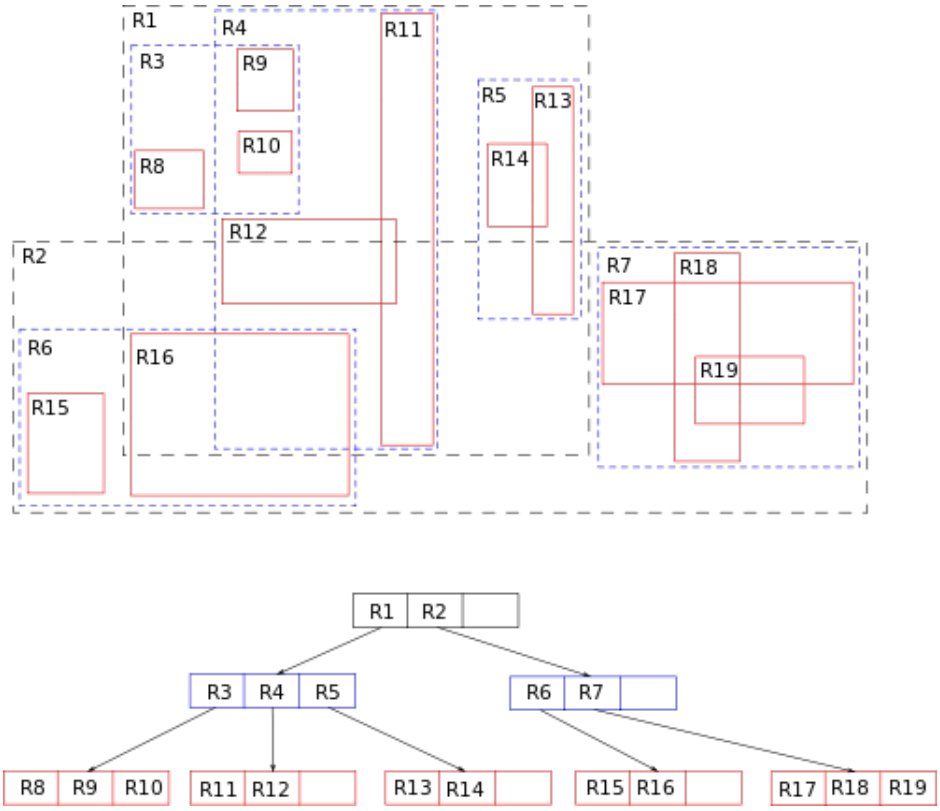
PostgreSQL støtter også andre prosedyrespråk. I base-distribusjonen har man tilgang på PL/Tcl, PL/Perl og PL/Python. Det finnes og en rekke andre språk som blir vedlikeholdt av eksterne [21], og det finnest og muligheter for å lage en handler selv for andre språk. [22].

2.2.3 Indeksering

Når et datasett blir tilstrekkelig stort trenger man indekser for å slippe å søke gjennom hele databasen hver gang man leter etter noe. PostgreSQL støtter mange ulike former for indeksering som B-tre, R-tre, hashtabeller og GiST. [25]

Indeksering kan gjøres på alle tabellverdier, men er kanskje mest vanlig på primærnøkler, da disse oftest nyttes ved oppslag. En populær metode er B-tre. Det er en trestruktur som gir oppslag i logaritmisk tid. [37] Den store forskjellen fra en hash tabell er b-treet sin mulighet til å hente ut sekvensielle data, og ikke bare en verdi per oppslag. [24]. B-tre er et svært nyttig verktøy for databasehåndtering, men er ikke egnet for romlige data. [27]

En populær metode for romlig indeksering er R-tre. Et R-tre er i likhet med B-tre et balansert tre. Men i stedet for å indeksere etter en enkelt verdi, så sorterer R-treet i rektangler. På laveste nivå så vil hver geometri lagres i henholdt til sitt omsluttende rektangel (BBOX/mbb - minimal bounding box). Disse rektanglene sorterest deretter i større directory rektangler som igjen er et minste omsluttende rektangel av verdiene det inneholder. [27]Ref. figur 2.1



Figur 2.1: Strukturen i et R-tre. [40]

Tidlige versioner av Postgis benyttet seg av PostgreSQL R-tre, men metoden er nå forkastet grunnet svakheter med implementeringen. [26] Teorien bak er derimot i full bruk gjennom GiST.

GiST (Generalized Search Tree) [7] er et avansert form for indekseringssystem med ulike søke- og sorteringsalgoritmer som B-tre, R-tre, B+-tre og mange andre. GiST gir mye av forutsetningene for Postgis, som er den romlige utvidelsen til PostgreSQL. [25]

2.3 Romlige databaser

En romlig database defineres av at den har et grensesnitt for spørringer mot koordinatfestede data som punkt, linjer eller polygon. Den kan svare på spørsmål som "Hvor mange hus ligger mindre en 100 meter fra E6?",

"Går E18 gjennom Arendal?" og lignende. Forutsetningen er selvfølgelig at vi har de gitte datasettene: "Veg", "Hus" og "Kommunegrenser".

2.3.1 Postgis

Postgis er en romlig utviding av PostgreSQL. Den gir romlige typer, romlige funksjoner og romlige indekser. Postgis følger OpenGIS sin "Simple Feature Specification for SQL" (SFSQL). Simple Features defineres i her som objekter med 2 eller færre dimensjoner. [15] Standarden gir svar på hvordan spørringer som snitt, union, inneholder og lignende skal tolkes.

2.3.2 Referansesystem

Postgis benytter seg av proj4 for å transformere mellom ulike romlige referanse system (SRS - Spatial Reference System). Dette biblioteket lister over 3000 ulike SRS.

De mest kjente systemene har sin egen kode i EPSG systemet. EPSG står for "European Petroleum Survey Group" [16] Kjente eksempler her vil være EPSG:4326 → WGS84 Long, Lat - EPSG:32633 → WGS84 UTM Sone 3.

Som eksempel vil proj4 koder for henholdsvis EPSG:4326 og EPSG:32633 være:

```
EPSG:4326
'+proj=longlat +datum=WGS84 +no_defs '

EPSG:32633
'+proj=utm +zone=33 +datum=WGS84 +units=m +no_defs '
```

2.3.3 Grafdatabaser som alternativ til RDBMS

En sammenligning gjort av Bart Baas i 2012 vist at en mer spesialisert grafhåndteringsdatabase som Neo4j kan være raskere en Postgis på kortestevei problem i ikke alt for store databaser. [1] Postgis er dog et mer allsidig

verktøy og har et større bruksområde innenfor GIS, så jeg har valgt å holde meg til løsninger knyttet til PostgreSQL/Postgis.

2.4 Ruteberegning i FIFO nettverk

Denne oppgaven vil kun se på FIFO nettverk (First-Inn-First-Out). Denne betingelsen er ofte satt for at algoritmene for å løse problemet ikke skal bli for komplekse. [14]. For korteste vei algoritmer i FIFO nettverk kan løsest i polynomisk tid. [10], I et ikke FIFO- netverk kan man fortsatt klare polynomisk tid om venting ved noder tillates, ellers er det NP-hard. [17]

Betingelser for FIFO nettverk [3]

1. Kjøretid er alltid positiv
2. lønner seg aldri å vente ved en node
3. korteste vei er aldri syklisk
4. et utsnitt av en korteste-vei beregning vil alltid være korteste vei mellom start og slutt i utsnittet.

2.4.1 Betingelse 1

Kjøretid i et veinettverk alltid er positiv. Man kan aldri tjene tid på å kjøre en veistrekning. Forutsetningen kan virke banal når man tenker på veinettverk, men den minker kompleksiteten i beregningsalgoritmene betraktelig.

2.4.2 Betingelse 2

Betingelsen at det lønner seg aldri å vente ved en node i et FIFO-nettverker litt mer vanskelig å overføre til et veinettverk. Her kan man fint se for seg at det kan lønne seg å vente på en midlertidig stengt vei, i forhold til å kjøre

rundt. Dette kan dog løses i måten man angir kjørekostnadene på:

Forutsetter stengt vei i tidsrommet $t_a - t_b$:

Gitt:
$t_a \leftarrow$ starttid for stenging
$t_b \leftarrow$ sluttid for stenging
$t_x \leftarrow$ beregnet tid ved noden

Alt 1: Oppfyller ikke betingelse for FIFO
if $t_a < t_x < t_b$ **then**
$cost \leftarrow \infty + normaltid$
else
$cost \leftarrow normaltid$
end if

Alt 2: Oppfyller betingelser for FIFO
if $t_a < t_x < t_b$ **then**
$cost \leftarrow t_b - t_x + normaltid$
else
$cost \leftarrow normaltid$
end if

2.4.3 Betingelse 3

Korteste vei er aldri syklisk. Dette følger av betingelse 1. $\Sigma cost_{t_a - t_b} > 0, cost \in < 0, ..]$. Så lenge kjøretid alltid er positiv vil en syklisk rute alltid gi en økning i total kjøretid.

2.4.4 Betingelse 4

Gitt raskeste rute fra a til b: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ så kan det ikke finnes en rute $b \rightarrow c' \rightarrow d$ kortere enn $b \rightarrow c \rightarrow d$.

2.5 Dijkstra

Dijkstras algoritme er kanskje den mest kjente algoritmen for å finne korteste vei i en graf. Oppkalt etter Edsger Dijkstra som laget og publiserte den i henholdsvis 1956 og 1959. Gir korteste vei mellom 2 noder, og krever at alle kostnader (f.eks. kjøretid) er positive. [38]

Dijkstras algoritme kan sees på et Breadth-first søk, men med ulike vekter på de ulike kantene, og en prioritetskø som finner noden med lavest reisetid. Fra en gitt startnode søker den kanter til alle nabonoder. Når alle noder er gjennomsøkt tas startnoden ut av prioritetskøen, og neste node med lavest reisetid settes til startnode. Prosessen fortsetter til målnoden er den noden med kortest reisetid i prioritetskøen.

Merk: Dijkstras orginale algoritme hadde ingen prioritetskø, og kjørte på $O(n^2)$, mot $O((k \cdot n) \log(n))$ med (binary heap) prioritetskø [14]. (k = kanter, n = noder)

2.6 pgRouting

pgRouting er en utvidelse til Postgis/postgreSQL for å tilby funksjonalitet for ruteberegning. Gitt at man har et topologiske data som en kantliste med verdier; "id", "source", "target", "cost" og eventuelt "reverse_cost", så kan pgRouting blant annet beregne korteste vei mellom to noder ("source" og "target").

I version 2.0 støtter pgRouting følgende funksjoner: [18]

- All Pairs Shortest Path, Johnson's Algorithm
- All Pairs Shortest Path, Floyd-Warshall Algorithm
- Shortest Path A*
- Bi-directional Dijkstra Shortest Path
- Bi-directional A* Shortest Path

- Shortest Path Dijkstra
- Driving Distance
- K-Shortest Path, Multiple Alternative Paths
- K-Dijkstra, One to Many Shortest Path
- Traveling Sales Person
- Turn Restriction Shortest Path (TRSP)

For oppgaven vil det bli fokusert på "Shortest path Dijkstra" Jeg vil referere til den som bare "Dijkstras".

Funksjonen for å beregne korteste-vei med Dijkstras algoritme tar som første argument en SQL-spørring som definerer området som skal benyttes. [18] Denne må passe overens med kantlisten som ble beskrevet over.

Vi ser av kildekoden til pgRouting [20] at denne benyttes til å bygge en graf i forkant av søket, for deretter å benytte C++ biblioteket BGL (Boost Graph Library) [31] til selve søket.

Det å bygge en graf tar ofte mye lengre tid en å søke gjennom den. Dette viser viktighet av bruk av bounding box når man vil gjennomføre korteste vei søk over en mindre del av et stort nettverk. Som eksempel vil et dijkstra søk fra Lindesnes til Kristiansand ta 1,44 sekund om man inkluderer hele databasen, mens om man tar en bounding box som på bildet under vil søket ta 0.08 sekund. Til sammenligning tar et søk fra Lindesnes til Kirkenes 1,48 sekund. Siden databasen kun er over Norge vil en bounding box på dette søket omfatte hele databasen. Se tillegg for kode over tidtaking. Ref Tillegg D

Figur 2.2: Eksempel på bounding box over Lindesnes og Kristiansand

2.7 XML

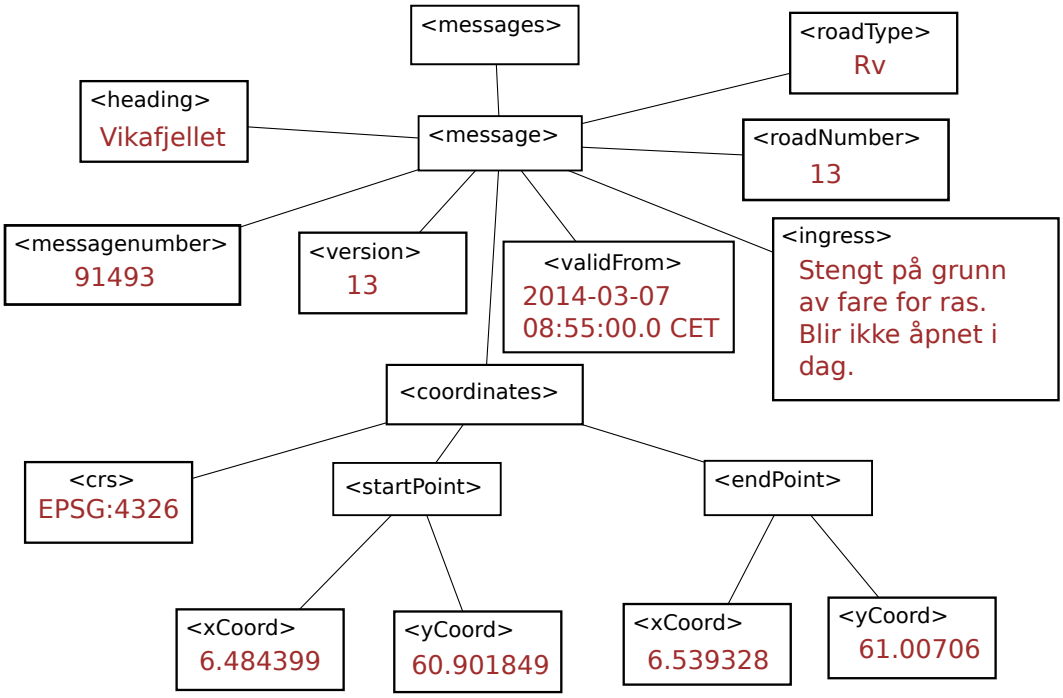
XML står for Extensible Markup Language. Det er et markeringspråk som nyttes for at teksten både skal være lett å lese for mennesker og lett å parse (lese) for datamaskiner.

XML er et metaspråk. Med det menes det at nye tagger kan defineres selv og man kan lage sin egen struktur som passer den informasjonen en har. Strukturen bør selvfølgelig være kjent både for den som skriver og den som leser meldingene, men trenger ikke være gitt i spesifikasjonene til "språket".

Meldinger blir sortert i en trestruktur og blir gitt merkelapper med av type: <merkelapp> melding </merkelapp>.

```
1 <messages>
2   <message>
3     <heading>Vikafjellet</heading>
4     <messagenumber>91493</messagenumber>
5     <version>13</version>
6     <ingress>Stengt på grunn av fare for ras. Blir ikke åpnet i dag.</
          ingress>
7     <roadType>Rv</roadType>
```

```
8       <roadNumber>13</roadNumber>
9       <validFrom>2014-03-07 08:55:00.0 CET</validFrom>
10      <coordinates>
11        <crs>EPSG:4326</crs>
12        <startPoint>
13          <xCoord>6.484399</xCoord>
14          <yCoord>60.901849</yCoord>
15        </startPoint>
16        <endPoint>
17          <xCoord>6.539328</xCoord>
18          <yCoord>61.00706</yCoord>
19        </endPoint>
20      </coordinates>
21    </message>
22  </messages>
```



Figur 2.3: XML trestruktur

Kapittel 3

Hvordan integrere i pgRouting

3.1 Tidligere prosjekter

3.1.1 TDSP - Gsoc2011

En forsøk på å implementere en "Time dependent Dynamic Shortest Path" (TDSP) [13] i pgRouting ble utført av Jay Mahadeokar som et Google summer of code (Gsoc) [8] prosjekt i 2011.

Prosjektet skisserer en løsning med to tabeller [13]:

1. Ways:
    - gid
    - source
    - target
    - name
    - the_geom
    - static_length (for vanlige, ikke time dependent søk))

- any other attributes

2. TimeDepCosts
    - gid
    - start_time
    - end_time
    - travel_time

Første tabllen "ways" er en kant-tabell over alle kanter i nettverket som skal søkes. Primærnøkkel er "gid". Andre tabell "TimeDepCost" er en tabell over kostnader på for hver kant på hvert tidspunkt.

Dokumentasjon av prosjektet er ikke svært detaljert. En prototype ble laget sommeren 2011, men ingen endringer ser ut til å ha blitt utført siden da. (pr. April 2014) [13]

Dataene i vårt tilfelle kunne nok ha blitt tilpasset til å fungere med denne modellen, men det er svært lite fleksibelt. Formlene ser ut til å være tilpasset datasett hvor man forehåndsberegner kjøretid på ulike tider av døgnet på bakgrunn av historiske data. Typisk eksempel vil være rushtrafikk langs motorveier og rundt store byer.

Jeg har valgt å ikke jobbe videre med denne løsningen.

3.1.2 Visitor pattern/callback funksjon

En mer generell løsning av problemet vil være å definere en lambda-funksjon [19] eller en visitor [30] løsning som gir mulighet for å legge inn noen ekstra linjer med kode i en ellers komplett funksjon.

Som nevnt i teoridelen så benytter pgRouting seg mye av Boost Graph Library (BGL). Dette burde gjøre det mulig å implementere BGL sine muligheter for visitor kode i en dijkstra algoritme.

Et visitor konsept gir muligheter for å legge inn kode på ulike deler av en

funksjon. BGL sin dijkstra funksjon gir muligheter for 7 ulike plasseringer: [29]

- **Initialize Vertex** - benyttes på hver node når grafen initialiseres
- **Examine Vertex** - benyttes på node når den tas fra prioritetskøen
- **Examine Edge** - benyttes på hver kant etter "Examine Vertex"
- **Discover Vertex** - benyttes når en kant blir brukt for første gang
- **Edge Relaxed** - hvis distansen til en kant reduseres
- **Edge Not Relaxed** - hvis distansen til en kant ikke reduseres
- **Finish Vertex** - benyttes på en node etter alle ut kanter er lagt til søks-treet.

For vårt tilfelle kunne for eksempel et par linjer kode i en "Examine Edge"-visitor forklart hvordan en tidsavhengig kostnad skal beregnes.

3.2 Veien videre

Utviklingen av TDSP ser ut til å ha stoppet [13], og det kan virke som om det er lite interesse for å implementere funksjoner for tidsavhengige data i pgRouting. [19]

I mitt videre arbeid vil jeg fokusere på utfordringene med utvikling av selve algoritmen. Min kunnskap om programmering i C er for dårlig til å kunne lage en fungerende løsning i pgRouting, så jeg har valgt å teste ut et konsept for min løsning i programmeringsspråket Julia.

Julia ble valgt da det er betraktelig mer brukervennlig enn C, men raskere enn Python. Julia er et relativ nytt programmeringsspråk og har fortsatt noe bugs i koden, og noe umoden bibliotek. Men det har en ganske intuitiv syntaks, og det er lett å prototype ny kode. [9]

Kapittel 4

Datasett / Verktøy

4.1 NVDB

NVDB står for Nasjonal vegdatabank. I denne oppgaven benytter jeg datasettet "vegnettsom er en del av NVDB. Det er beregnet på navigasjonsformål. Vegnett ble frigitt 27. September 2013 ifm et større statlig fri-slipp av kartdata. [33] Dataene er lastet ned fra Statens vegvesen sin ftp på adressen:

`ftp://vegvesen.hostedftp.com/~StatensVegvesen/vegnett/`

Dataene er tilgjengeliggjort i både ESRI- og spatiaLiteformat. Jeg har benyttet meg at data i spatialiteformat datert 15.10.2013.

«Inneholder data under norsk lisens for offentlige data (NLOD) tilgjengeliggjort av Statens vegvesen.»

4.1.1 Verdier

Vegnett består av en tabell **ruttger_link_geom** med en rekke kolonner.

Noen av de viktige er:

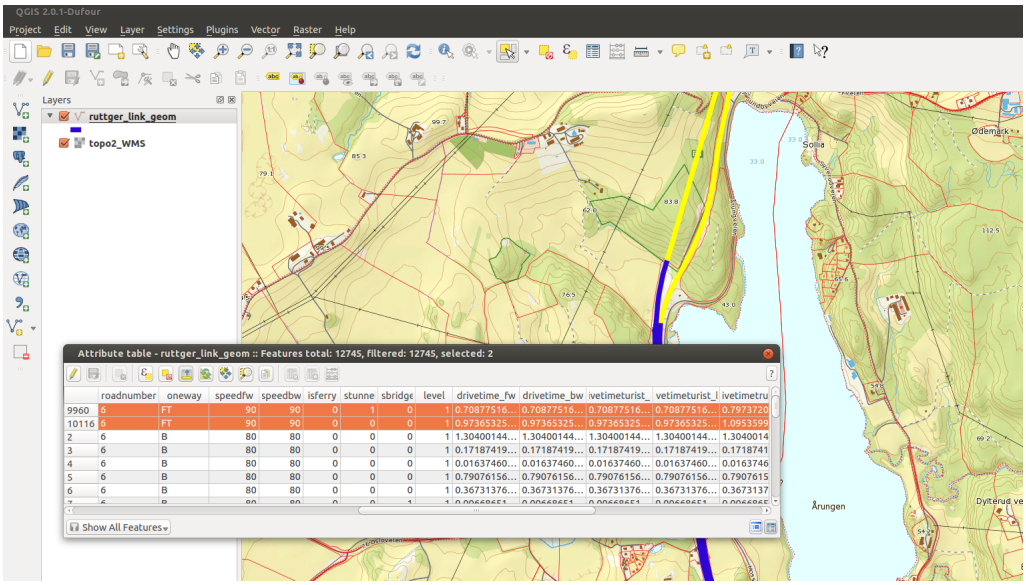
- **ogc_fid** - primærnøkkel

```
Resultset metadata for executed query
-------------------------------------
Query: SELECT * FROM ruttger_link_geomColumns: 31
Rows: 1415176
31x5 DataFrame:
                 Column Names                Types   Sizes Digits Nullable
[1,]                "ogc_fid"    ("SQL_INTEGER",4)      10      0        0
[2,]           "wkb_geometry"   ("SQL_VARCHAR",12) 8354052      0        1
[3,]                 "linkid"    ("SQL_INTEGER",4)      10      0        1
[4,]               "fromnode"    ("SQL_INTEGER",4)      10      0        1
[5,]                 "tonode"    ("SQL_INTEGER",4)      10      0        1
[6,]             "streetname"   ("SQL_VARCHAR",12)     255      0        1
[7,]              "fromangle"    ("SQL_INTEGER",4)      10      0        1
[8,]                "toangle"    ("SQL_INTEGER",4)      10      0        1
[9,]              "formofway"    ("SQL_INTEGER",4)      10      0        1
[10,]         "funcroadclass"    ("SQL_INTEGER",4)      10      0        1
[11,]               "routeid"   ("SQL_VARCHAR",12)     255      0        1
[12,]          "from_measure"      ("SQL_FLOAT",6)      15      0        1
[13,]            "to_measure"      ("SQL_FLOAT",6)      15      0        1
[14,]            "roadnumber"   ("SQL_VARCHAR",12)     255      0        1
[15,]                "oneway"   ("SQL_VARCHAR",12)     255      0        1
[16,]               "speedfw"    ("SQL_INTEGER",4)      10      0        1
[17,]               "speedbw"    ("SQL_INTEGER",4)      10      0        1
[18,]               "isferry"    ("SQL_INTEGER",4)      10      0        1
[19,]              "istunnel"    ("SQL_INTEGER",4)      10      0        1
[20,]              "isbridge"    ("SQL_INTEGER",4)      10      0        1
[21,]                 "level"    ("SQL_INTEGER",4)      10      0        1
[22,]          "drivetime_fw"      ("SQL_FLOAT",6)      15      0        1
[23,]          "drivetime_bw"      ("SQL_FLOAT",6)      15      0        1
[24,]    "drivetimeturist_fw"      ("SQL_FLOAT",6)      15      0        1
[25,]    "drivetimeturist_bw"      ("SQL_FLOAT",6)      15      0        1
[26,]     "drivetimetruck_fw"      ("SQL_FLOAT",6)      15      0        1
[27,]     "drivetimetruck_bw"      ("SQL_FLOAT",6)      15      0        1
[28,]               "roadid"   ("SQL_VARCHAR",12)     255      0        1
[29,]            "roadclass"   ("SQL_VARCHAR",12)     255      0        1
[30,]           "attributes"   ("SQL_VARCHAR",12)     255      0        1
[31,]                  "fgf" ("SQL_VARBINARY",-3)     255      0        1
```

- **wkb_geometry** - geometrien
- **linkid** - kantnummer
- **fromnode** - startnode
- **tonode** - sluttnode
- **oneway** - kjøreretning
- **drivetime_fw** - kjøretid fra **fromnode** til **tonode**
- **drivetime_bw** - kjøretid fra **tonode** til **fromnode**
- **roadid** - vegid (E6, E18, etc.)

Noe som kan være greit å bemerke er at datasettet gir gyldige verdier for både **drivetime_fw** og **drivetime_bw** selv om vegen er envegskjørt. Dette skaper noe problemer ved søk i pgRouting da det (meg bekjent) ikke har noen funksjon for å slå opp i en annen kolonne (**oneway**) for å sjekke kjøreretning. Dette kommer nok av at Statens Vegvesen benytter andre verktøy en pgRoouting i sitt arbeid, hvor denne måten å vise dataen på passer bedre.

I figur 4.1 ser vi verdier for E6 nord- og sørgående vest for Årungen. Her er det like verdier i begge kjøreretninger, mens **oneway** er satt til **FT : fromnode** $\rightarrow$ **tonode**.



Figur 4.1: Kjøretid med og mot kjørerretning. E6 ved Årungen

For pgRouting hadde det vert bedre om kjøretider mot kjøreretning ble satt til uendelig (Infinty).

4.1.2 Koordinatsystem

Konvertering av data fra Spatialite til Postgis ble gjort med ogr2ogr. I første omgang oppstod det en del problemer som følge av at ogr2ogr/Postgis ikke klarte å i finne ut datum på koordinatene.

ogr2ogr [6] er en del av GDAL-biblioteket (Geospatial Data Abstraction Library). [5] Ved å benytte programmet gdalsrsinfo til å finne informasjon om koordinatsystemet til databasen i originalt spatialiteformat fikk jeg følgende output:

```
1 eh@eh-W350STQ-W370ST:~/stud/master/julia$ gdalsrsinfo trfnetwork_20131015.
      sqlite -o epsg
2 Warning 1: EPSG detection is experimental and requires new data files (see
       bug #4345)
3 EPSG:-1
4 eh@eh-W350STQ-W370ST:~/stud/master/julia$ gdalsrsinfo trfnetwork_20131015.
      sqlite
5
```

```
6  PROJ.4 : '+proj=utm +zone=33 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m
       +no_defs '
7
8  OGC WKT :
9  PROJCS["UTM Zone 33, Northern Hemisphere",
10     GEOGCS["GRS 1980(IUGG, 1980)",
11         DATUM["unknown",
12             SPHEROID["GRS80",6378137,298.257222101],
13             TOWGS84[0,0,0,0,0,0,0]],
14         PRIMEM["Greenwich",0],
15         UNIT["degree",0.0174532925199433]],
16     PROJECTION["Transverse_Mercator"],
17     PARAMETER["latitude_of_origin",0],
18     PARAMETER["central_meridian",15],
19     PARAMETER["scale_factor",0.9996],
20     PARAMETER["false_easting",500000],
21     PARAMETER["false_northing",0],
22     UNIT["Meter",1]]
```

Vi ser her at koordinatene er i UTM sone 33 Nord. Valgte derfor å sette koordinatsystemet til EPSG:32633 med $-a_srs$ taggen.

```
1  ogr2ogr -f PostgreSQL -a_srs "EPSG:32633" PG:"host=localhost user=postgres
       password=postgres dbname=vei01" trfnetwork_20131015.sqlite -
       skipfailures
```

Dette fungerte i utgangspunktet bra, men jeg oppdaget senere at det strengt tatt var feil og at vegnett-datasettet sin SRID er Sweref99 TM. Forskjellen mellom UTM sone 33 Nord og Sweref99 TM er dog ikke stor. Det svenske Lantmäteriet opplyser på sine sider på internett: "*Den nationalla kartprojektionen till SWEREF 99, benämnd SWEREF 99 TM, är definierad på samma sätt som UTM zon 33, med den skillnaden att den är utökad till att gälla för hela landet.*" [11] Vi ser også fra gdalsrsinfo fra de to epsg-id'ene at forskjellen består i at Sweref benytter GRS 1980, mens UTM benytter WGS 84. Da forskjellen mellom de to er minimal har jeg fortsatt å bruke UTM33N (EPSG:32633). (Se Tillegg C for detaljer)

4.2 Trafikkmeldinger

Statens Vegvesen utgir for tiden sine trafikkmeldinger i XML format. Det er planer om innføring av en felles europeisk standard for levering av trafikkmeldinger som heter Datex II. Datex står for DATa EXchange og er tenkt

som en standard for informasjonsutveksling mellom ulike aktører i trafikkbransjen. [34] [2] Standarden baserer seg på XML, og eg har derfor valgt å benytte meg av dagens trafikkmeldinger i XML-format og regner da prinsippene vil være like.

Trafikkmeldingen ble lastet ned fra:

http://www.vegvesen.no/Trafikkinformasjon/Reiseinformasjon/Trafikkmeldinger/Videreformidling

Statens Vegvesen anbefaler ikke mellomlagring av trafikkdata da det er hyppige endringer av disse. [36] Ser det vanskelig å løse den her type problem uten mellomlagring av data, så har valgt å se bort fra dette rådet.

4.2.1 Fritekst

Trafikkinformasjonen er ofte mer kompleks enn en gitt XML mal kan håndtere. Her ser vi et eksempel på en trafikkmelding fra Silatunnelen.

```
<message>
 <heading>Silatunnelen</heading>
 <messagenumber>90469</messagenumber>
 <version>2</version>
 <ingress>Stengt på grunn av vedlikeholdsarbeid i periodene:
         Mandag til fredag fra 23:00 til 06:00 (neste dag).
         Gjelder fra 11.02.2014 klokken 23:00 til 31.05.2014
         klokken 06:00.</ingress>
 <freetext>Trafikken slippes gjennom kl 01:00, 03:00 og 05:00.</freetext>
 <messageType>Midlertidig stengt</messageType>
 <urgency>U</urgency>
 <roadType>Fv</roadType>
 <roadNumber>17</roadNumber>
 <validFrom>2014-02-11 23:00:00.0 CET</validFrom>
 <validTo>2014-05-31 06:00:00.0 CEST</validTo>
 <actualCounties>
   <string>Nordland</string>
 </actualCounties>
 <coordinates>
   <crs>EPSG:4326</crs>
   <startPoint>
     <xCoord>13.176012</xCoord>
     <yCoord>66.318463</yCoord>
   </startPoint>
 </coordinates>
```

```
26  <distributionPattern>
27    <period>
28      <from-date>2014-02-11T23:00:00.0 +0100</from-date>
29      <to-date>2014-05-31T06:00:00.0 +0200</to-date>
30      <from-time>23:00</from-time>
31      <to-time>06:00</to-time>
32      <days-of-week>
33        <day>mandag</day>
34        <day>tirsdag</day>
35        <day>onsdag</day>
36        <day>torsdag</day>
37        <day>fredag</day>
38      </days-of-week>
39    </period>
40  </distributionPattern>
41 </message>
```

Til tross for for at **distributionPattern**-taggen gir mange muligheter for å spesifisere klokkeslett og ukedager hvor meldingen er gyldig, så har man sett seg nødt til å gi ekstra informasjon i **freetext**-taggen om at trafikken slippes gjennom kl 01:00, 03:00 og 05:00.

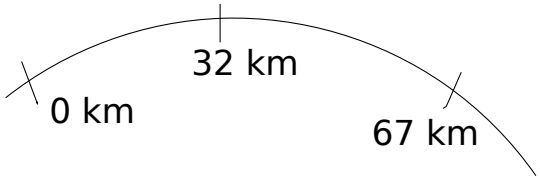
Å søke etter og utnytte informasjon i en fritekst gir stort rom for feil, med mindre man har standardisert hva som kan skrives, og hvordan det skal skrives.

Jeg vil for mine tester i denne oppgaven forholde meg kun til **validFrom** og **validTo** for å holde ting enkelt og siden jeg kun vil teste et konsept.

4.2.2 Referansesystem

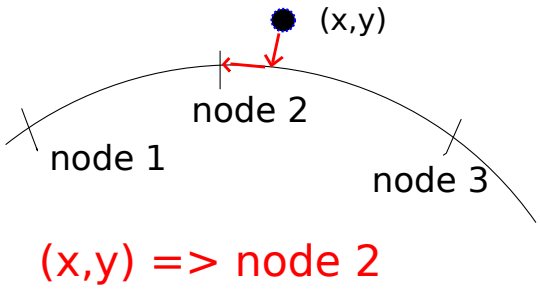
Statens Vegvesen har valgt å benytte egenkoordinater for å oppgi start- og sluttpunkt i sine trafikkmeldinger.

Et alternativ kunne vært å benytte et lineært referansesystem, kanskje bedre kjent som kilometrering. Dette kunne beskrevet mer presist hvor langs veien man er (ref. avsnitt 5.4), men på bekostning av at referansene vil hoppe om geometrien endres. (lengden på veien)



Figur 4.2: Lineært referansesystem

Til tross for at trafikkmeldingene oppgir x og y koordinat for start- og sluttpunkt for vegmelding, så vil restriksjonene **roadType** og **roadNumber** legge føringer for at punktene projiseres ned til et 1-dimensjonalt referansesystem.



Figur 4.3: Fra egenkoordinater til noder

Kapittel 5

Fra koordinater til noder og kanter

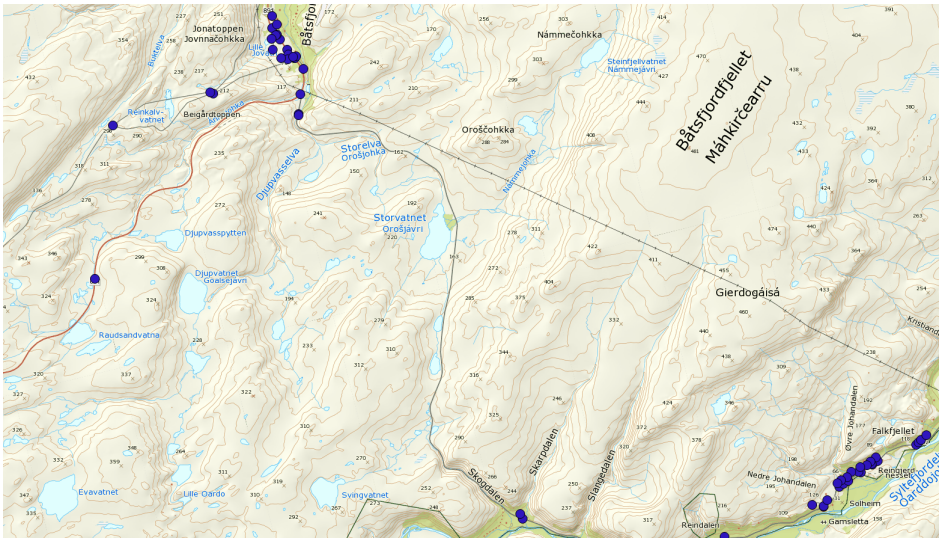
5.1 Problemstilling

Gitt trafikkmelding oppgitt med vegid, start-koordinater og slutt-koordinater, hvordan konvertere dette til noder og kanter?

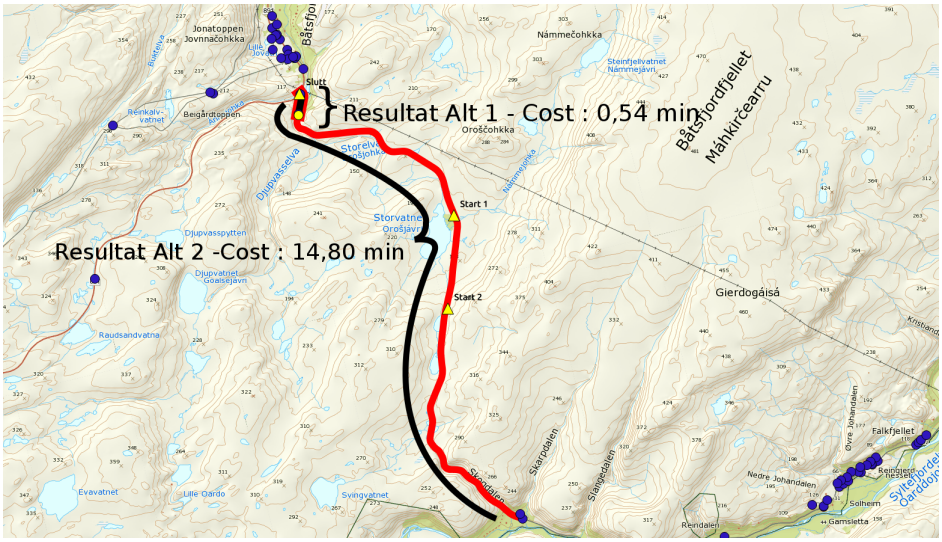
5.2 Nodetetthet

Et topologisk nettverk vil som regel kun ha noder i kryss for å minimere antall noder som må lagres. Dette medfører at alle korteste-vei søk må starte og slutte ved en node. I et detaljert topologisk nett som det vi bruker hvor også mindre private veier også er inkludert så vil dette som regel ikke medføre store problemer. Men noen lengre strekninger uten kryss kan det få litt merkelige konsekvenser som i eksempelet fra Båtsfjordfjellet i figur 5.1 og 5.2.

Her ser vi en lang veistrekning med få noder. Gitt at man skulle foreta et korteste-vei søk fra midt opp på fjellet og ned til Båtsfjorddalen så vil man kunne få over 14 minutt forskjell i tid avhengig av hvilken ende (node) av strekningen man er nærmest.



Figur 5.1: Båtsfjordfjellet, kun noder og veier.



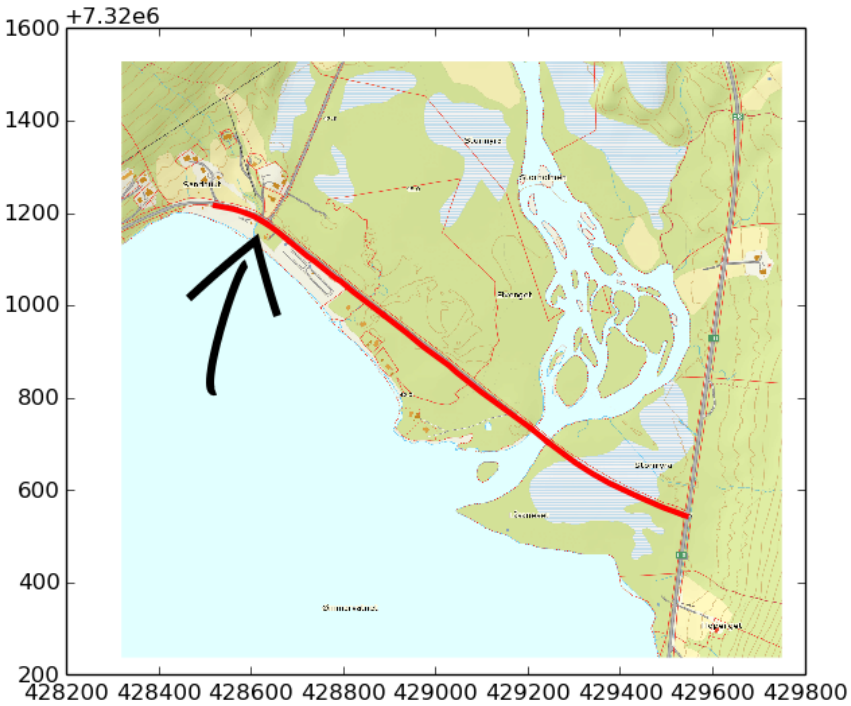
Figur 5.2: Båtsfjordfjellet. Reisetid i forhold til oppgitt startkoordinat

5.3 Nøyaktighet i meldingene

Følgende trafikkmelding ble funnet i SVV sin XML rapport over stengte veier.

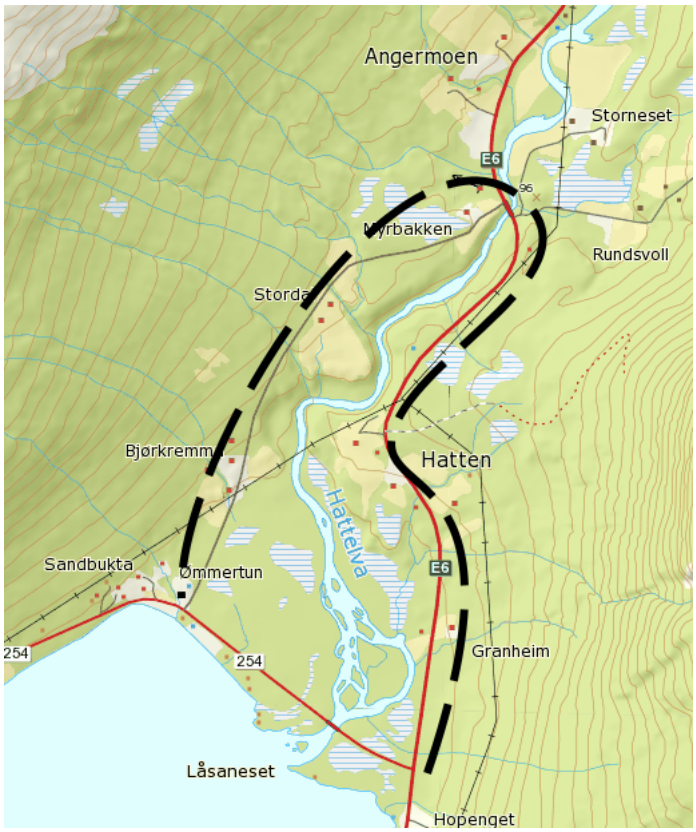
```
<message>
  <heading>Ømmervatn - Sandbukta, på strekningen
           Blåfjell - Ømmervatn</heading>
  <messagenumber>88707</messagenumber>
  <version>1</version>
  <ingress>Stengt på grunn av vedlikeholdsarbeid.
           Omkjøring er skiltet.</ingress>
  <freetext>Omkjøringen er via kommunal veg nr. 49
            over Stordalen (Angermoen til Sandbukt).</freetext>
  <messageType>Midlertidig stengt</messageType>
  <!-- ... -->
  <coordinates>
    <crs>EPSG:4326</crs>
    <startPoint>
      <xCoord>13.447618</xCoord>
      <yCoord>65.997868</yCoord>
    </startPoint>
    <endPoint>
      <xCoord>13.423868</xCoord>
      <yCoord>66.003679</yCoord>
    </endPoint>
  </coordinates>
</message>
```

Strekningen som kan hentes ut av meldingen her tilsvarer rød strek på bildet i figur 5.3. Vi ser tydelig at sperret vei går forbi krysset til venstre i bildet.



Figur 5.3: Stengt vei: Ømmervatn - Sandbukta

I **freetext**-taggen på trafikkmeldingen ser vi at foreslått omkjøring er over Stordalen, Angermoen til Sandbukt. Dette tilsvarer stiplet svart strek på bildet i figur 5.4.



Figur 5.4: Omkjøring: Ømmervatn - Sandbukta

Her ser vi tydelig eksempel på at om nøyaktigheten i meldingene ikke er god nok, så vil man kunne oppleve uønskede effekter. I dette tilfellet vil ikke et korteste-vei søk med restriksjoner kunne klare å kjøre den veien som er gitt i friteksten, da dette ikke stemmer overens med koordinatene i meldingen.

5.4 Søk etter nærmeste node vs. nærmeste kant

Geometriske søk er en relativ kostbar operasjon, og det kan være fristende å lage egne nodetabeller for søk etter nærmeste node i stedet for nærmeste kant.

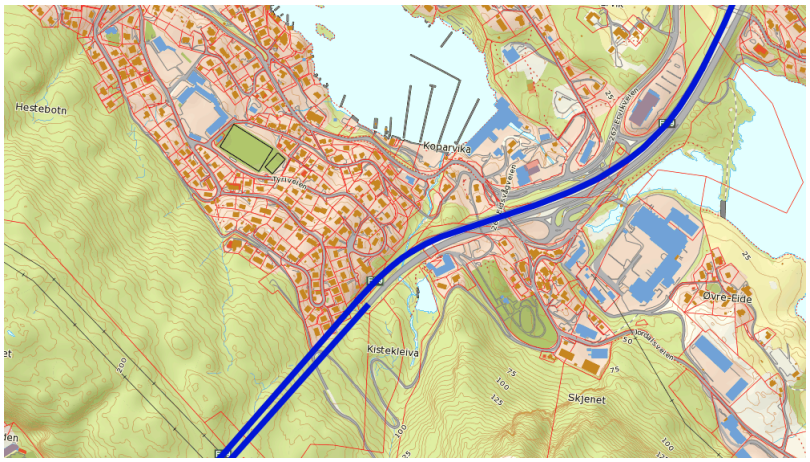
Figur 5.5: Nærmeste node vs. nærmeste kant

På bildet i figur 5.5 et tenkt eksempel hvor koordinatene vi søker fra er merket med **start**.

Søk etter et nærmeste punkt i en nodeliste vil kunne gi en halvering av tid kontra å søke etter nærmeste kant (linestring) og deretter nærmeste punkt av første og siste punkt i linestring-geometrien. Dessverre vil ikke dette gi ønsket svar i alle tilfeller. I dette tilfellet gir nærmeste node til **start** svar **306767**, mens vi ser tydelig at punktet ligger nærmere en annen veistrekning. Riktig node skal være **302520** som blir svaret ved den andre beskrevne metoden. Se tillegg B.

Greit også å bemerke at dersom man begrenser søket etter veistrekning så vil tiden også halveres. Dette er greit å vite da dette vil være oppgitt i en trafikkmelding.

Men også ved spesifisering av vegstrekning kan man noen ganger få feil vei. I figur 5.6 ser vi at ene koordinaten tydeligvis har havnet på riktig veg-id, men i feil kjøreretning.

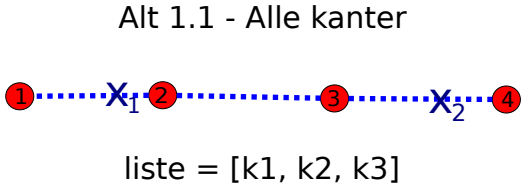


Figur 5.6: Motorvei - node kommer på feil kjøreretning

5.5 Mulige løsninger for lagring av veistrekning

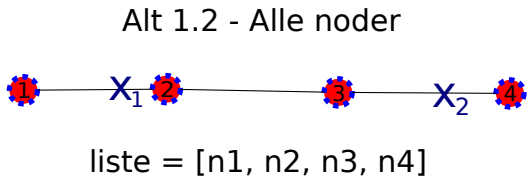
Gitt trafikkmelding oppgir startkoordinater $\mathbf{X}_1$ og sluttkoordinater $\mathbf{X}_2$. Vi har noder **n1, n2, n3** og **n4**, og kanter $\mathbf{k1}_{n1-n2}$, $\mathbf{k2}_{n2-n3}$ og $\mathbf{k3}_{n3-n4}$. Hvordan bør man lagre informasjon om veistrekningen slik at man lettest mulig kan integrere dem i en korteste-vei algoritme.

5.5.1 Alt 1.1



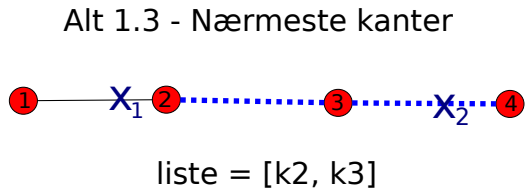
Vurdering: Mest intuitive løsning på problemet. Man finner nærmeste kant på begge sider, og alle kanter mellom. Men jeg vil senere i oppgaven vise at dette skaper noen problemer for om man ønsker et fast tillegg i tid på trafikkmeldingen.

5.5.2 Alt 1.2



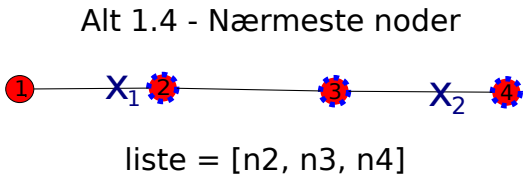
Vurdering: En strategi med lagring av nodelister og ikke kanter er et alternativ jeg vil studere videre i denne oppgaven, men i dette tilfellet vil adgang til **node 1** og **4** bli sperret til tross for at de er utenfor gitt veistrekning $\mathbf{X}_1 - \mathbf{X}_2$.

5.5.3 Alt 1.3



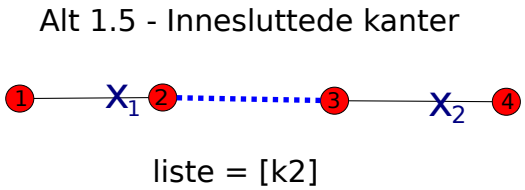
Vurdering: Her vil man kunne kjøre **kant 1 (n1 -n2)** uten at ekstra kostnader beregnes. Går ikke videre med dette alternativet.

5.5.4 Alt 1.4



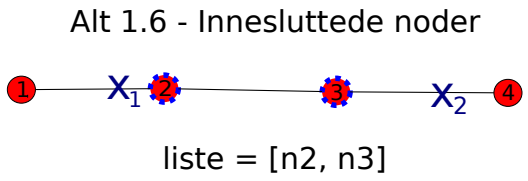
Vurdering: $\mathbf{X}_1$ ligger her mellom en node i listen og en utenfor. $\mathbf{X}_2$ ligger mellom to noder i listen. Dette gjør det vanskelig å utforme en algoritme som kan håndtere begge tilfeller. Går ikke videre med dette alternativet.

5.5.5 Alt 1.5



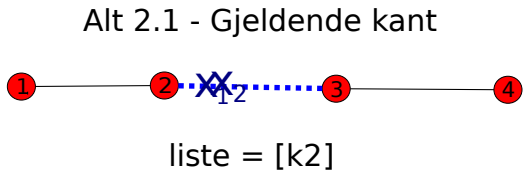
Vurdering: Dette blir noe likt problemene med **Alt 1.3**. Her kan man kjøre **kant 1** og **3** uten at ekstra kostnader beregnes. Går ikke videre med dette alternativet.

5.5.6 Alt 1.6



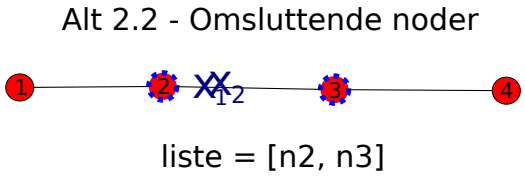
Vurdering: Denne metoden vil jeg benytte videre i arbeidet med å lage en funksjon for å bestemme tidsavhengig kostnad. Her ser vi at alle nodene som er innenfor $\mathbf{X}_1 - \mathbf{X}_2$ ligger i nodelisten, og ikke flere. En metode der **v** (til-node) er i listen, og **u** (fra-node) er utenfor vil gi muligheter til å sette en ventetidskostnad ved første ankomstpunkt på strekningen. Mer om dette senere.

5.5.7 Alt 2.1



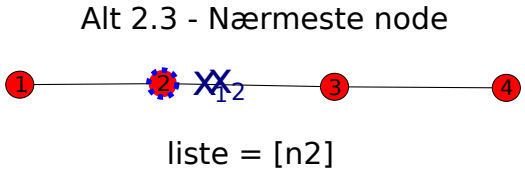
Vurdering: I tilfeller der både start og sluttpunkt for vegmelding ligger på samme kant vil det være naturlig at det er kun denne kanten som er berørt av trafikkmeldingen. **n2** og **n3** vil fortsatt kunne nås fra henholdsvis **n1** og **n4**, men strekningen **n2-n3** får en ekstra kostnad.

5.5.8 Alt 2.2



Vurdering: Et alternativ med lagring av noder kunne vert interessant også for tilfeller der trafikkmeldingen kun berører en kant, men den vil ha noen uheldige bi-effekter ved at **n2** og **n3** blir tatt med i det sperrede området selv om de ikke skal være berørt. Går ikke videre med dette alternativet.

5.5.9 Alt 2.3



Vurdering: Dette blir mye samme feil som ved **Alt 2.2**. Går ikke videre med dette alternativet.

5.6 Statisk endring av kostnad

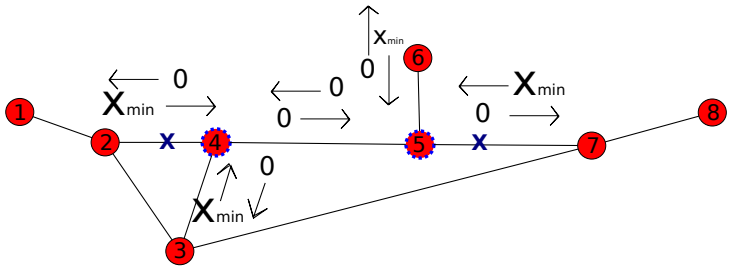
5.6.1 Faktor

Forsinkelser i form av en faktor (f.eks. 20% økning i kjøretid) er forholdsvis enkelt å løse. Kjøretid blir lik **normalkjøretid** + (**normalkjøretid** · **fak-**

tor). Eksempler kan være glatt vegbane o.l. Her passer det bra å ha lagret kantlister av gjeldende område som i **Alt 1.1**.

5.6.2 Fast tillegg

Et fast tillegg i tid er vanskeligere å løse med kantlister som i **Alt 1.1**. Et alternativ da vil være å fordele den faste kostnaden på hver av kantene, dette vil virke i noen tilfeller, men ikke om man velger en trase som kun kjører over noen av kantene. Det man ønsker her er at det faste tillegget i tid legges til første gang man kommer inn på strekningen. Et eksempel kan være opp til 30 min venting grunnet veiarbeid. Her vil det være ser vi tydelig at ventingen ikke vil avhenge av om du kjører hele eller halve strekningen. Et annet eksempel kan være i tilfeller der man måler avvik fra normal kjøretid langs en akse. Dette gjøres mellom annet med Autopass-brikker gjennom bomstasjoner. [35] Ønsket effekt er skissert på tegningen i figur 5.7.



Figur 5.7: Kostnadt ved fast tillegg, fast tid

Med lagring av indre noder, som i **Alt 1.6**, så vil man kunne sortere tilfeller som dette med **u** $\notin$ **nodeliste, v** $\in$ **nodeliste**, her **u** er fra-node og **v** er til-node. Dette vil også fange opp tilfeller der man forsøker å kjøre inn til de avgrensede nodene fra andre veier som i tilfellet **n3-n4** og **n6-n5**. En ulempe med metoden er om man kjører inn på veistrekningen, for deretter å ta en detour ut og inn igjen. Da vil kostnaden påløpe to ganger, noe som trolig ikke er ønskelig. Men da det å følge en hovedvei uten detourer i dei fleste tilfeller vil være å foretrekke, så anser jeg dette ikke å være et stort problem.

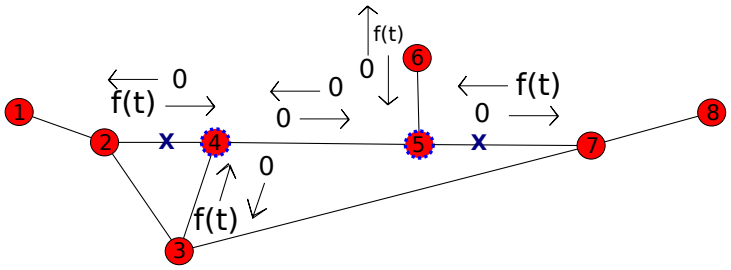
5.7 Tidsavhengig endring av kostnad

5.7.1 Faktor

Når tillegget ligger på hver kant, så kan ekstrakostnaden beskrives som det minste av beregnet tillegg og gjenværende tid av trafikkmeldingens gyldighet. Kjøretid blir lik **normalkjøretid + minimum((normalkjøretid · faktor), ($\mathbf{t}_b$- $\mathbf{t}_x$))**, der $\mathbf{t}_b$ er sluttid for vegmelding, og $\mathbf{t}_x$ er tid ved kanten.

5.7.2 Fast tillegg

Metoden blir ganske lik ved fast tillegg. Man bytter ut den faste kostnaden fra trafikkmeldingen **x**, og bytter den ut med en **f(x) = minimum(x ,($\mathbf{t}_b$- $\mathbf{t}_x$))**



Figur 5.8: Kostnadt ved fast tillegg, variabel tid

Merk også her at at ulempen med detourer nevnt for et par avsnitt siden ikke vil gjelde den tidsavhengige faktoren, da denne vil sette kostnad lik **0** når $\mathbf{t} > \mathbf{t}_b$. Det vil si at en tidsavgrenset stenging **($\mathbf{x} = \infty$)** vil ikke ha noen slike feil. **minimum(Inf,($\mathbf{t}_b$- $\mathbf{t}_x$))**

5.8 Foreslåtte formler for tidsavhengig kostnad

Deler sorterer meldinger etter to hash-tabeller / Dictionary. En for oppslag etter noder og en for oppslag etter kanter. På denne måtene kan meldinger som skal ha en fast kostnad for hele strekningen (type 1 kostnad) lagres med indre noder som i **Alt 1.6**. Kostnader med prosentvis endring av kjøretid, eller hvor meldingen kun har en kant og ingen indre noder, kan lagres som i kant-tabellen. (**Alt 1.1** og **Alt 2.1**)

Funksjonen **td_cost** kan deretter under internt i en dijkstra-funksjon sjekke opp mot de to hash-tabellene om det er eventuelle meldinger som gjelder. Hvis så kan man beregne kostnad av denne meldingen med en ny funksjon **getnewcost**. Resultatkostnaden av funksjonen vil være den høyeste kostnaden av alle meldinger som gjelder kanten man beregner.

```
1  type1data = [
2  "Midlertidig Stengt",
3  "Vinterstengt",
4  "Kolonnekjøring"
5  ] # etc.
6
7  noder = Dict()
8  kanter = Dict()
9  for (n, mnr) in enumerate(trfdata["mnr"])
10   # finn meldingstype
11   mtype = trfdata["mtype"][n]
12
13   # hvis type 1 data
14   # legg indre noder i "noder"
15   if (in(mtype, type1data)
16      && length(trfdata["kl"][n]) > 1)
17
18     for node in trfdata["nl"][n]
19       if !haskey(noder, node)
20         noder[node] = [mnr]
21       else
22         push!(noder[node], mnr)
23       end
24     end
25
26   # hvis type 2 data eller kun en kant
27   # legg kanter i "kanter"
28   else
29     for kant in trfdata["kl"][n]
```

```
30        if !haskey(kanter, kant)
31          kanter[kant] = [mnr]
32        else
33          push!(kanter[kant], mnr)
34        end
35      end
36
37    end
38  end
39
40  function td_cost(u, v, k, t, d)
41
42  # u = franode
43  # v = tilnode
44  # k = kant
45  # t = tid ved u
46  # d = normal kjøretid fra u til v
47
48    cost = 0.0
49    if !haskey(noder, u)
50      if haskey(noder, v)
51         for mnr in noder[v]
52           newcost = getnewcost(mnr, t, d)
53           cost = maximum([cost, newcost])
54         end
55      end
56    end
57    if haskey(kanter, k)
58      for mnr in kanter[k]
59         newcost = getnewcost(mnr, t, d)
60         cost = maximum([cost, newcost])
61      end
62    end
63    return cost
64  end
```

5.9 Dynamiske systemer

Som nevnt i datasett kapittelet så anbefaler ikke Statens Vegvesen mellomlagring av trafikkdata da det er hyppige endringer av disse. [36] I den grad nye trafikkmeldinger som kommer i løpet av kjøreturen kan behandles, så har vi et dynamisk system. Dette skaper en del problemer med optimaliseringstiltak som krever preprosessering av nettverket som vi skal vise i neste avsnitt. Det medfører også at nytt korteste-vei søk må gjøres etter hver oppdatering

av trafikkmeldinger.

Om vi har svært hyppige oppdateringer i trafikkmeldinger kan man også vurdere å lagre hvilke trafikkmeldinger som ble behandlet av gjeldene søk. Deretter sammenligne om oppdaterte meldinger gjelder disse eller gjeldende kantliste for korteste vei. I praksis tror jeg det er lite å spare for vårt tilfelle.

5.10 Optimaliseringer i ikke-statiske nettverk

I statiske nettverk finnes det mange metoder for å effektivisere søk. Mange av disse går ut på å preprosessere snarveier mellom noder. Dette går fint så lenge man vet kostnaden til snarveien, men ettersom de vil variere med tid, så blir det straks mer komplisert.

A$\star$ (star): Mulig å integrere. A$\star$ benytter ingen preprosessering en kun et ekstra argument i prioritets-nøkkelen; en (under)estimering av avstanden til mål-noden (euklidsk avstand) [14]. Men gir lite besparelse i pgRouting sammenlignet med tiden brukt for å aksessere data og å bygge grafen. [28]

Bidirectional søk: Anses ikke mulig da kostnadene avhenger av tid, og ankomsttidspunktet er ukjent. I alle fall med de bidirectional funksjonene som er implementert i pgRouting.

ALT, Highway Hierarchies og SHARC er alle godt etablerte metoder i statiske nettverk, men ingen av dem støttes av pgRouting. [18] Metoder finnes for å utnytte disse metodene i time-dependent nettverk [4], men videre analyse av om de kunne tilpasses vårt tilfelle har ikke vert utført da de ikke er implementert i pgRouting.

Kapittel 6

Test av konsept

6.1 Hva er utført

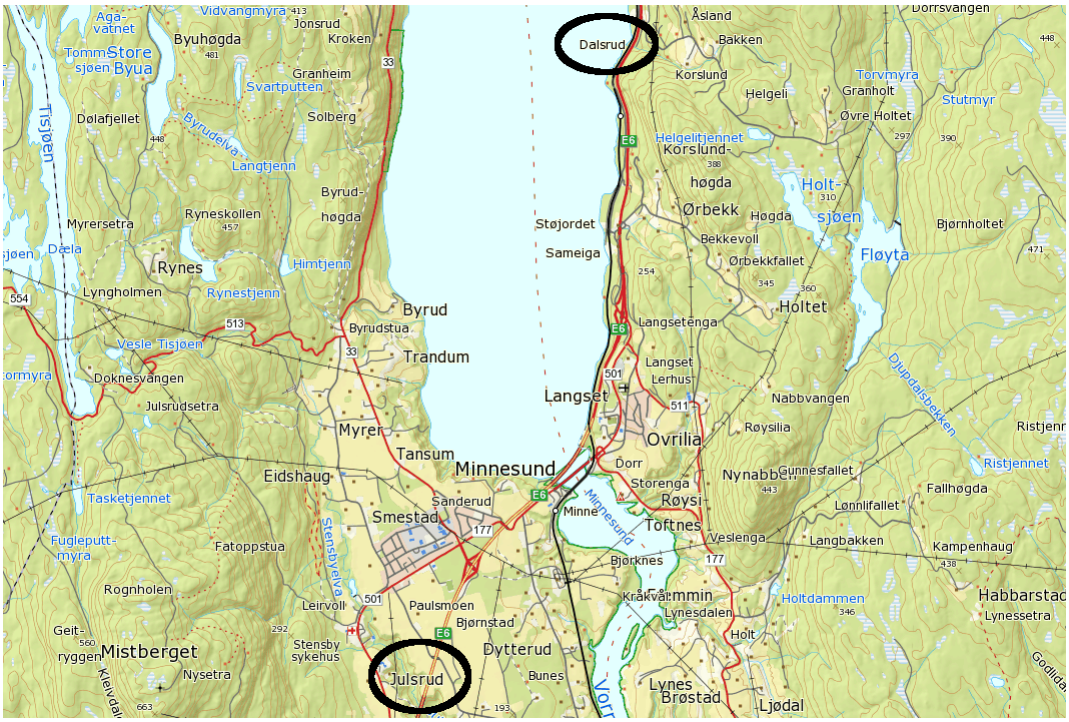
For å teste at skisserte løsning fungerer har det blitt laget et script i Julia som utfører en tidsavhengig korteste-vei beregning basert på NVDB og trafikkdata fra SVV.

Koden er fordelt på fire dokument som alle ligger i Tillegg A. Løsningen er noe ad hoc, hvor romlige spørringer gjøres direkte mot PostgreSQL, mens topologiske spørringer gjøres mot et nedlastet utsnitt av datasettet. Dette er gjort for å enklest mulig teste funksjonaliteten uten bruke alt for mye tid på programmering.

6.2 Et eksempel: Midlertidig stengt vei ved Minnesund

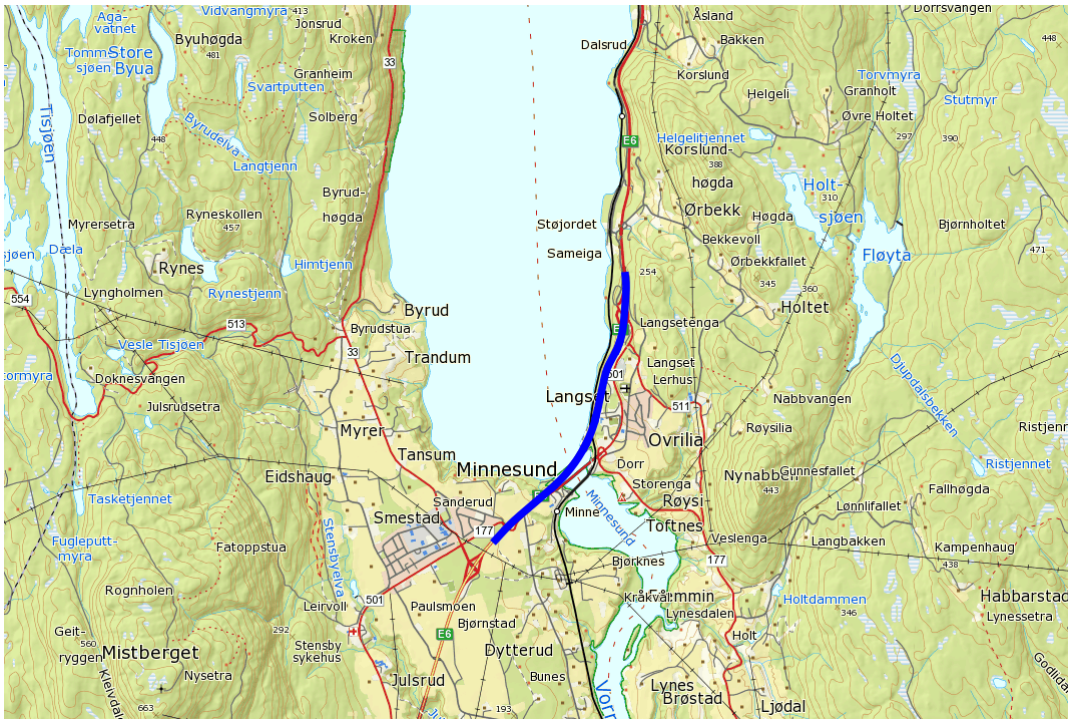
6.2.1 Start- og sluttpunkt for ruteberegning

Beregning av start- og sluttpunkt gjøres av formelen **closestnode** som gjør en spørring mot vegenett-tabellen i PostgreSQL.




```
julia> julsrud = closestnode(291084,6699736)
226445

julia> dalsrud = closestnode(293273,6706449)
1082267
```

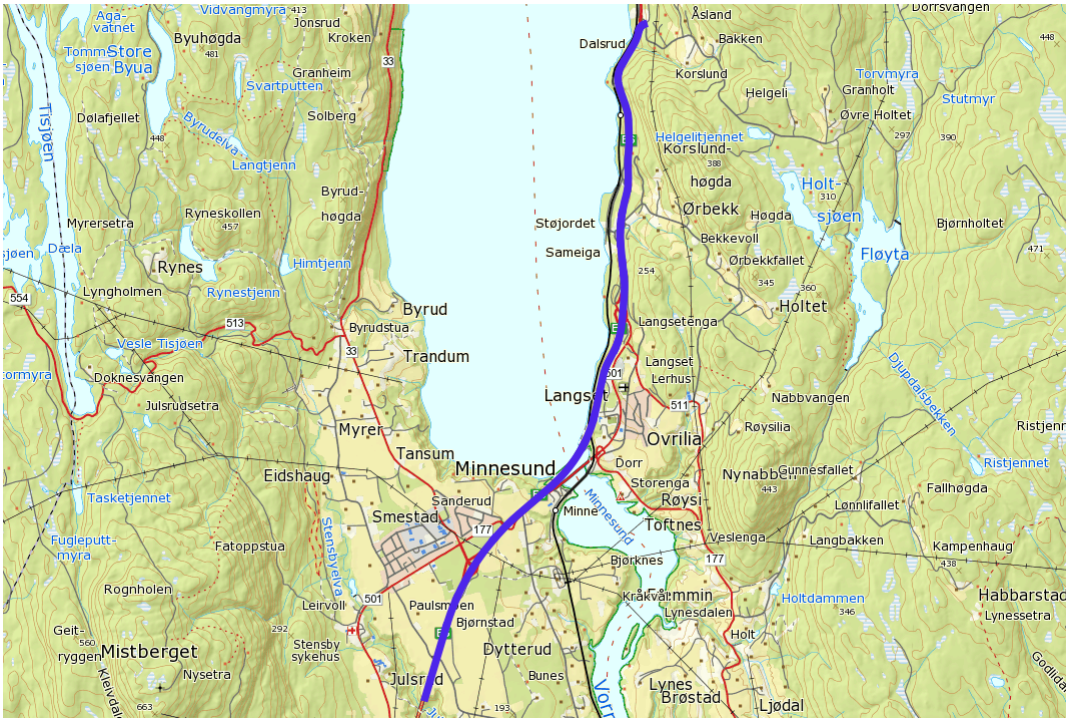
6.2.2 Veiområdet som er meldt som sperret




```
<messages>
  <message>
    <heading>[58] Minnesundkrysset - [59] Langsetkrysset, på strekningen
        Minnesund - Hamar, i retning mot Hamar</heading>
    <messagenumber>94487</messagenumber>
    <version>3</version>
    <ingress>Stengt på grunn av vegarbeid. Omkjøring er skiltet.</ingress>
    <messageType>Midlertidig stengt</messageType>
    <urgency>X</urgency>
    <roadType>Ev</roadType>
    <roadNumber>6</roadNumber>
    <roadDirectionString>N:Hamar</roadDirectionString>
    <validFrom>2014-05-05 22:00:00.0 CEST</validFrom>
    <validTo>2014-05-06 05:30:00.0 CEST</validTo>
    <actualCounties>
      <string>Akershus</string>
    </actualCounties>
    <coordinates>
      <crs>EPSG:4326</crs>
      <startPoint>
        <xCoord>11.22298</xCoord>
        <yCoord>60.394312</yCoord>
```

```
      </startPoint>
      <endPoint>
        <xCoord>11.241533</xCoord>
        <yCoord>60.415276</yCoord>
      </endPoint>
    </coordinates>
  </message>
</messages>
```

6.2.3 Eksempel 1: Søk utenfor gyldighetsområdet




```
julia> p1, c1 = td_dijkstra(vegnett, julsrud, dalsrud,
               "2014-05-05 14:00:00"; printd=true)
elapsed time: 0.680798843 seconds
elapsed time: 0.103289808 seconds
([112990,112991,112992,730166,730167,730168,730169,1047053,1047054,1047055
     ...
    1047057,1047058,1047059,1047060,1047061,1047062,1047063,1047064,1047065,997341],
     6.195608580436362)
```

Kommentar: Som forventet påløper det ingen tidsavhengige kostnader når man tidspunktet ligger utenfor gyldighet av trafikkmeldingen.

6.2.4 Eksempel 2: Søk med start midt i gyldighetsområdet


```
julia> p2, c2 = td_dijkstra(vegnett, julsrud, dalsrud,
                "2014-05-05 23:00:00"; printd=true)
elapsed time: 0.617160924 seconds
----------> linkid: 730166 <---------------
| mnr: 94487 cost: 388.74116008356214
| [58] Minnesundkrysset - [59] Langsetkrysset, på strekningen Minnesund -
    Hamar, i retning mot Hamar
| Stengt på grunn av vegarbeid. Omkjøring er skiltet.
| Midlertidig stengt
-------------------------------------------------
----------> linkid: 1047055 <---------------
| mnr: 94487 cost: 383.54852579161525
| [58] Minnesundkrysset - [59] Langsetkrysset, på strekningen Minnesund -
    Hamar, i retning mot Hamar
| Stengt på grunn av vegarbeid. Omkjøring er skiltet.
| Midlertidig stengt
-------------------------------------------------
----------> linkid: 1047053 <---------------
| mnr: 94487 cost: 383.44179202988744
| [58] Minnesundkrysset - [59] Langsetkrysset, på strekningen Minnesund -
    Hamar, i retning mot Hamar
```

```
| Stengt på grunn av vegarbeid. Omkjøring er skiltet.
| Midlertidig stengt
-------------------------------------------------
----------> linkid: 1047054 <---------------
| mnr: 94487 cost: 383.44179202988744
| [58] Minnesundkrysset - [59] Langsetkrysset, på strekningen Minnesund -
    Hamar, i retning mot Hamar
| Stengt på grunn av vegarbeid. Omkjøring er skiltet.
| Midlertidig stengt
-------------------------------------------------
elapsed time: 0.015001081 seconds
([112990,112991,112993,112994,730248,730249,730250,730257,730255,730254
    ...
    1047057,1047058,1047059,1047060,1047061,1047062,1047063,1047064,1047065,997341],
     8.821167469304651)
```

Kommentar: Vi ser her at algoritmen har forsøkt å komme inn på den sperrede veien 4 plasser, men har ikke lykkes. Kostnaden er satt til ca 380 min som tilsvarer 6 timer, 20 min. Som forventet.

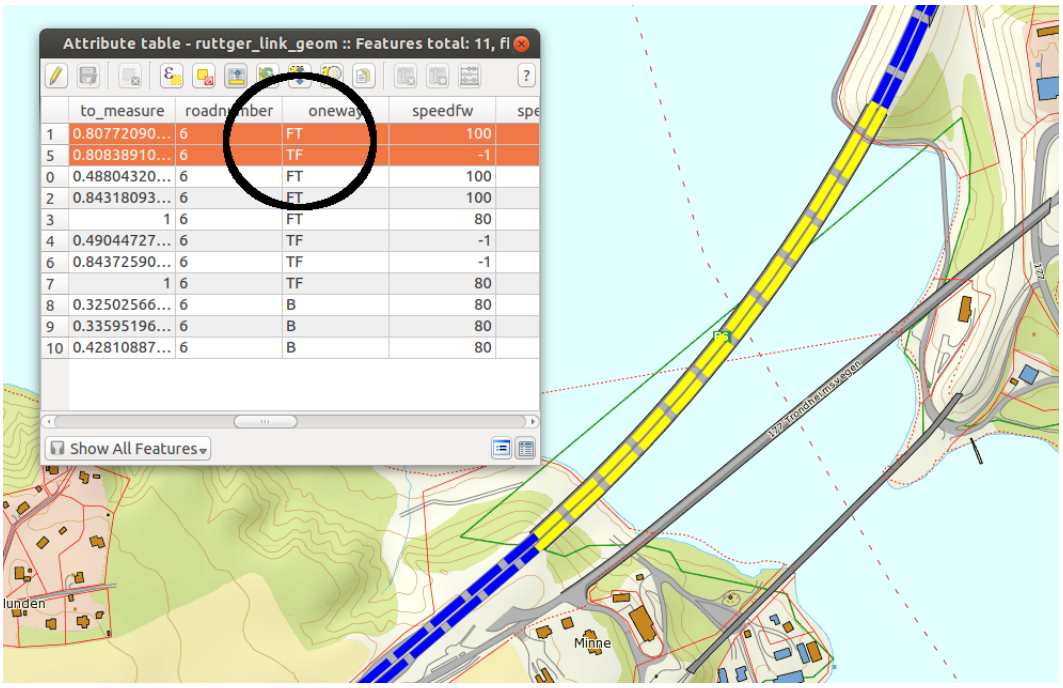
6.2.5 Eksempel 3: Søk med start 3 min før utløp av gyldighet

```
julia> p3, c3 = td_dijkstra(vegnett, julsrud, dalsrud,
               "2014-05-06 05:27:00"; printd=true)
elapsed time: 1.19254776 seconds
----------> linkid: 730166 <---------------
| mnr: 94487 cost: 1.7411600835621357
| [58] Minnesundkrysset - [59] Langsetkrysset, på strekningen Minnesund -
    Hamar, i retning mot Hamar
| Stengt på grunn av vegarbeid. Omkjøring er skiltet.
| Midlertidig stengt
-----------------------------------------------
elapsed time: 0.004800116 seconds
([112990,112991,112992,730166,730167,730168,730169,1047053,1047054,1047055
     ...
    1047057,1047058,1047059,1047060,1047061,1047062,1047063,1047064,1047065,997341],
     7.936768663998498)

julia> p1 == p3
true

julia> c3 - c1
1.7411600835621357
```

Kommentar: Vi ser her at søket kun treffer på en kant som har tidsavhengig kostnad. Kostnaden er 1,74 minutt, og vi ser at kjøreveien til eksempel 1 (**p1**) er lik den til eksempel 3 (**p3**). Tilsvarende er kjøretid **c3** 1,74 minutt lenger en **c1**. Dette viser at for kort gyldighet av trafikkmelding kan det lønne seg å vente.

6.3 Begrensinger

For å sperre begge retninger på en motorvei har valgt løsning antatt at ulike kjøreretninger har ulik **oneway**-verdi. Dette var en feil antagelse og vil kunne føre til feil resultater i veier som i figur 4.1 på side 24. Dessverre ble det oppdaget noe sent, og det har ikke vert tid å rette feilen. Kunne vert ønskelig å hatt tilgang til lineære referanser for vegnettet for å kunne velge alle kanter med lik veg-id mellom to punkt/referanser.



Figur 6.1: Minnesund bro. Oneway verdier.

Kostnader for ulike typer vegmeldinger er kun gitt for et fåtall typer meldinger, da flere ikke har vert nødvendig for å teste konseptet.

Tar ikke hensyn til detaljerte tidspunkt for stenging gitt i **distributionPattern**-taggen, ei heller informasjon i **freetext**-taggen.

Tar kun informasjon fra en XML-fil, og har ikke laget noen funksjoner for oppdatering av verdier.

Kapittel 7

Diskusjon / Konklusjon

Selv om jeg i denne oppgaven skisserer en mulig løsning for å integrere tidsavgrensede trafikkmeldinger i en korteste-vei ruteberegning så er det fortsatt en del kompliserende faktorer:

- Koordinater i trafikkmeldinger er ikke alltid nøyaktige nok.
- Tidsinformasjon i fritekst kompliserer og gir rom for feil.
- Kan man stole på at tidsinformasjon er er reell?
- Ulike datasett kan gi utfordringer med generell implementering i pgRouting.

7.1 Nøyaktighet av koordinater

I et topologisk nettverk der noder kun er satt ved kryss, vil en koordinat som er plassert på feil side av krysset kunne føre til at en vei som skulle vert åpen blir merket som stengt, og omvendt. En har sett i oppgaven at slike feil kan forekomme i trafikkmeldingene, men det er uvisst i hvor stort omfang.

7.2 Nøyaktighet av tidsinformasjon

SVV sine trafikkmeldinger inneholder ganske presise beskrivelser av tidsinformasjon. I denne oppgaven har en kun benyttet seg av start og sluttid for meldingens gyldighet, men dette kunne enkelt vert utvidet til også å gjelde data i **distributionPattern**-taggen i XML meldingene. Data i **freeText**-taggen kan være litt verre om ikke det er standardisert format.

Men hovedproblemet er ikke å beskrive tidsinformasjonen presist nok. Problemet er om det er nøyaktig. Starter og slutter vegarbeidet som meldt, og ikke minst er bilen der på planlagt tidspunkt? Vet man nøyaktig nok når man reiser, og bruker man akkurat like lang tid som beregningen sier?

En del meldinger kan greit integreres som f.eks. vinterstengte veier, eller veier som er stengt over flere dager. Men disse tilfellene kan også løses med midlertidig endring av kjøretidsverdiene i databasen. Problemene oppstår i grenseområdet rundt start og slutt for trafikkmeldingen, og problemet er ikke å programmere det, men om man tør å stole på at det stemmer.

Et eksempel kan være hvis ruteberegningen ved tid **x** sier at det er ingen stengte veier, men en 15 minutt kaffepause fører til at du burde hatt en ruteberegning **x+15** som gir stengt vei. Du vil få ingen forvarsel ved første beregning og resultatet kan bli at du må snu og ta en annen vei.

7.3 Generalisere for pgRouting

Kvaliteten på open-source program henger ofte sammen med at mange har interesse av å jobbe med de over tid. Redigering av kode, feilsøking, oppdatering o.l. For at det skal være interesse for å lage et brukergrensesnitt for korteste-vei søk med trafikkinformasjon til pgRouting krever det at man klarer å generalisere problemstillingen. Løsningen må ikke bare virke for vårt tilfelle, men kunne generaliseres til å kunne gjelde andre lignende problemer. Jeg tror visitor funksjonaliteten i BGL som var diskutert i denne oppgaven vil kunne gi denne fleksibiliteten.

7.4 Konklusjon

Kompleksiteten til trafikkmeldingene er svært stor. Feil plassering av koordinater eller tidsinformasjon som endrer seg kan føre til lange, unødvendige omkjøringer.

En ren integrasjon av alle fremtidige trafikkdata i en ruteberegning gir mange utfordringer i implementering. Noen grad av varsling av relevante trafikkmeldinger med nærhet i tid kan synes nødvendig.

Formlene fra denne oppgaven blir svært følsomt for variasjoner i starttid. Vil anbefale å jobbe videre med løsninger som inkluderer en form for input fra bruker om hvorvidt trafikkmeldinger skal tas med eller ikke.

Tillegg A

Julia-kode for ruteberegning

```
using ODBC
using DataStructures
using DataFrames

# Koble til Postgis
co = ODBC.connect("myswitch")

# Last inn database
db = sql"SELECT linkid id, fromnode, tonode,
               drivetime_fw cost_fw,
               drivetime_bw cost_bw,
               roadid, oneway
        FROM ruttger_link_geom"

# ODBC gir Int32 istedet for Int64
# fungerer dårlig sammen med hash. ref. #6580
# https://github.com/JuliaLang/julia/issues/6580
# løkke for konvertering:

for i in ["id", "fromnode", "tonode"]
  db[i] = int(db[i])
end

# ODBC gir en del feil av typen ekstra } på slutten av roadid
# har ikke funnet ut hvorfor dette skjer, men retter for det.

for (num,val) in enumerate(db["roadid"])
  m = match(r"([{]){1}([A-Z]{1})([0-9]{1,5})([}]{1})", val)
  if m != nothing
    db["roadid"][num] = m.match
  else
```

```
32      println(val)
33    end
34  end
35
36  # Setter kjøretid lik Inf for feil retning på envegskjørt veg
37  for (num,val) in enumerate(db["oneway"])
38    if val == "FT"
39      db["cost_bw"][num] = Inf
40    elseif val == "TF"
41      db["cost_fw"][num] = Inf
42    end
43  end
44
45
46  type VegnettDB
47    data::DataFrame
48    naboer::Dict
49    n::Int
50  end
51
52  function createdb(db)
53
54    # hash-tabell for raskt oppslag av nabonoder:
55
56    naboer = Dict()
57    for (n, (i, j)) in enumerate(zip(db["fromnode"], db["tonode"]))
58      if !haskey(naboer, i)
59        naboer[i] = [n]
60      else
61        push!(naboer[i], n)
62      end
63      if !haskey(naboer, j)
64        naboer[j] = [n]
65      else
66        push!(naboer[j], n)
67      end
68    end
69
70    # Maxverdi av node
71
72    n = maximum(vcat(maximum(db["fromnode"]),
73                     maximum(db["tonode"])))
74
75    # Returverdi
76
77    VegnettDB(db, naboer, n)
78
79  end
80
```

```
function show(io::IO, x::VegnettDB)
  print(io, "VegnettDB")
end

vegnett = createdb(db)

include("dbtools.jl")
include("sp.jl")
include("tdd.jl")
include("trfdata.jl")
```

```
list2tup(list) = ntuple(length(list), i -> list[i])

function getgeom(list_of_linkid)
  tup = list2tup(list_of_linkid)
  geom = query("SELECT ST_AsGeoJSON(ST_Union(wkb_geometry))
               FROM ruttger_link_geom WHERE linkid IN $tup")[1,1]
end


function closestnode(x, y, srid=32633; where="")
  node = query("
    WITH selection AS
      (WITH line AS
        (SELECT fromnode, tonode, wkb_geometry FROM ruttger_link_geom
        $where
        ORDER BY ST_Distance(ST_Transform(ST_GeomFromText(
        'POINT($x $y)', $srid), 32633), wkb_geometry)
        ASC LIMIT 1)
      SELECT fromnode node, ST_STARTPOINT(wkb_geometry) geom FROM line
      UNION
      SELECT tonode node, ST_ENDPOINT(wkb_geometry) geom FROM line)
    SELECT node FROM selection
    ORDER BY ST_Distance(ST_Transform(ST_GeomFromText('POINT($x $y)',
    $srid), 32633), geom) ASC LIMIT 1")[1,1]
  node = int(node)
end


function closestedge(x, y, srid=32633; where="")
  kant = query("
    SELECT linkid FROM ruttger_link_geom $where
    ORDER BY ST_Distance(ST_Transform(ST_GeomFromText(
    'POINT($x $y)', $srid), 32633), wkb_geometry) ASC LIMIT 1")[1,1]
  kant = int(kant)
end

function closestedges_fw_bw(x, y, srid=32633; where="", rec=false)
  # finner nermeste kant som er toveiskjørt
```

```
39    # eventuellt nermeste kanter fram og tilbake
40    # forutsetter ulike kjøreretninger har ulik FT/TF
41    # ikke god løsning
42    kanter = query("
43      SELECT linkid, oneway FROM ruttger_link_geom $where
44      ORDER BY ST_Distance(ST_Transform(ST_GeomFromText(
45      'POINT($x $y)', $srid), 32633), wkb_geometry) ASC LIMIT 1")
46
47    oneway = kanter["oneway"][1]
48    linkid = int(kanter["linkid"][1])
49
50    if oneway == "B"
51      return linkid
52    elseif in(oneway, ["FT", "TF"])
53      if rec
54        # skal kun kjøre en rekursjon for åfinne både FT og TF
55        return linkid
56      else
57        whereand = (where == "" ? "WHERE " : " AND ")
58        otherway = oneway == "FT" ? "TF" : "FT"
59        return vcat(linkid, closestedges_fw_bw(x, y, srid;
60               where=(where * whereand * "oneway = '$otherway'"),
61               rec=true))
62      end
63    else
64      error("$oneway er ikke godkjent verdi for oneway, kun B, FT, og TF")
65    end
66  end
67
68  function kantliste2nodeliste(db, kl)
69    nl = Int[]
70    if kl == []
71      return nl
72    end
73
74    if length(kl) == 2
75      # finn midtnode
76      r1,r2 = findfirst(db["id"], kl[1]), findfirst(db["id"], kl[2])
77      if (((n1f = db["fromnode"][r1]) == db["tonode"][r2])
78      || (n1f == db["fromnode"][r2])) # just in case
79        push!(nl, n1f)
80      else
81        push!(nl, db["tonode"][r1])
82      end
83    else
84      # ta noder fra annenhver kant
85      for linkid in kl[2:2:end-1]
86        row = findfirst(db["id"], linkid)
87        push!(nl, db["fromnode"][row])
```

```
88        push!(nl, db["tonode"][row])
89      end
90      if iseven(length(kl))
91        # hvis partall ta midtnode av siste 2 kanter
92        row = findfirst(db["id"], kl[end-1])
93        if in(db["fromnode"][row], nl[end-1:end])
94          push!(nl, db["tonode"][row])
95        else
96          push!(nl, db["fromnode"][row])
97        end
98      end
99    end
100   return nl
101 end
102
103
104 function koord2list(db, sx, sy, ex, ey, srid, rid)
105   # lag ny VegnettDB med kun gitt vei (rid)
106   tmp = createdb(db[(db["roadid"] .== rid), :])
107   tmpdb = tmp.data
108   # Finn nermeste kanter (ev fram og tilbake) fra ODBC / Postgis
109   sk = closestedges_fw_bw(sx, sy, srid; where="WHERE roadid = '$rid'")
110   ek = closestedges_fw_bw(ex, ey, srid; where="WHERE roadid = '$rid'")
111   if sk != ek
112     # hvis mer en en kant
113     l = length(sk) * length(ek)
114     costlist = FloatingPoint[]
115     pathlist = Array[]
116     for skn in sk, ekn in ek
117       # test mulige kombinasjoner av start og sluttnoder
118       if skn == ekn
119         continue
120       end
121       sn = tmpdb[findfirst(tmpdb["id"], skn), "fromnode"]
122       en = tmpdb[findfirst(tmpdb["id"], ekn), "fromnode"]
123       path, cost = my_dijkstra(tmp, sn, en)
124       rev_p, rev_c = my_dijkstra(tmp, en, sn)
125       if rev_c < cost
126         path, cost = rev_p, rev_c
127       end
128       if path != []
129         if first(path) != skn
130           unshift!(path, skn)
131         end
132         if last(path) != ekn
133           push!(path, ekn)
134         end
135       end
136       push!(costlist, cost)
```

```
      push!(pathlist, path)
    end
    # finn 1 eller 2 korteste veier og legg til kant- og nodeliste
    order = sortperm(costlist)
    n = minimum([1, 2])
    kantliste = Int[]
    nodeliste = Int[]
    for path in pathlist[order[1:n]]
      kantliste = vcat(kantliste, path)
      nodeliste = vcat(nodeliste, kantliste2nodeliste(tmpdb, path))
    end
  else
    # hvis kun en kant (ev fram og tilbake)
    kantliste = isa(sk, Array) ? sk : [sk]
    nodeliste = []
  end
  return kantliste, nodeliste
end
```

```
using Calendar

function td_dijkstra(datab::DataFrame, startnode, endnode, starttid;
    printd=false)
  datab = createdb(datab)
  td_dijkstra(datab, startnode, endnode, starttid; printd)
end


function td_dijkstra(datab::VegnettDB, startnode, endnode, starttid;
    printd=false)
  # starttid må oppgis i formatet : "yyyy-MM-dd HH:mm:ss"
  tic()
  db = datab.data
  n = datab.n
  naboer = datab.naboer

  path = zeros(Int, n)
  parent = zeros(Int, n)
  poped = falses(n)
  dist = fill(Inf, n)
  dist[startnode] = 0.0

  starttid = parsetime(starttid)
  if starttid <= 0
    error("starttid må oppgis i formatet : \"yyyy-MM-dd HH:mm:ss\"")
  end

  q = mutable_binary_minheap(dist)

```

```
  toc()
  tic()
  while !isempty(q) && (u = q.nodes[1].handle) != endnode
    pop!(q)
    poped[u] = true

    if !haskey(naboer, u)
      continue
    end

    for row in naboer[u]
       if (to = db[row, "tonode"]) == u
         to = db[row, "fromnode"]
         linkcost = db[row, "cost_bw"]
       else
         linkcost = db[row, "cost_fw"]
       end
       cost = dist[u] + linkcost

       td = td_cost(u, to, db[row,"id"], dist[u]+starttid,
                    linkcost; printd=printd)
       cost += td

       if !poped[to] && cost < dist[to]
         dist[to] = cost
         update!(q, to, cost)
         path[to] = db[row, "id"]
         parent[to] = u
       end
    end
  end

  edges = Int[]
  i = endnode
  while i != startnode
    unshift!(edges, path[i])
    if path[i] == 0
      return [], Inf
    end
    i = parent[i]
  end
  toc()
  return (edges, dist[endnode])
end

function td_cost(u, v, k, t, d; printd=false)
  cost = 0.0
  if !haskey(noder, u)
    if haskey(noder, v)
```

```
78        for mnr in noder[v]
79          newcost = getnewcost(mnr, t, d)
80          cost = maximum([cost, newcost])
81          if printd
82            printdata(mnr, k, cost)
83          end
84        end
85      end
86    end
87    if haskey(kanter, k)
88      for mnr in kanter[k]
89        newcost = getnewcost(mnr, t, d)
90        cost = maximum([cost, newcost])
91        if printd
92          printdata(mnr, k, cost)
93        end
94      end
95    end
96    return cost
97  end
98
99  function printdata(mnr, k, cost)
100   if cost == 0.0 return 0 end
101   row = findfirst(trfdata["mnr"], mnr)
102   println("----------> ", "linkid: ", lpad(k,10), " <---------------")
103   println("| ", "mnr: ", mnr, " cost: ", cost)
104   println("| ", trfdata["heading"][row])
105   println("| ", trfdata["ingress"][row])
106   println("| ", trfdata["mtype"][row])
107   println("-----------------------------------------------")
108   return 0
109 end
110
111 function getnewcost(mnr, t, d)
112   row = findfirst(trfdata["mnr"], mnr)
113   st = parsetime(trfdata["stime"][row]; alt=0.0)
114   et = parsetime(trfdata["etime"][row]; alt=Inf)
115   t_diff = del_in_min(st, et, t)
116   wait = msgcost(row, d)
117   newcost = minimum([wait, t_diff])
118 end
119
120 function msgcost(row, d)
121   mtype = trfdata["mtype"][row]
122   mtype == "Midlertidig stengt" ? Inf :
123   mtype == "Vinterstengt" ? Inf :
124   mtype == "Kolonnekjøring" ? 120 : # + 2 timer
125   mtype == "Glatt veg" ? d * 0.2 : # + 20 %
126   0 # etc.
```

```
end

function parsetime(strng, form="yyyy-MM-dd HH:mm:ss"; alt=0)
  t = try
    Calendar.parse(form, strng).millis / 60000
  catch
    alt
  end
end

function del_in_min(st, et, t_now)
  if st < t_now < et
    diff = (et-t_now)
  else
    diff = 0
  end
  return diff
end
```

```
function my_dijkstra(datab::DataFrame, startnode, endnode)
  datab = createdb(datab)
  my_dijkstra(datab, startnode, endnode)
end

function my_dijkstra(datab::VegnettDB, startnode, endnode)
  tic()
  db = datab.data
  n = datab.n
  naboer = datab.naboer

  path = zeros(Int, n)
  parent = zeros(Int, n)
  poped = falses(n)
  dist = fill(Inf, n)
  dist[startnode] = 0.0

  q = mutable_binary_minheap(dist)

  toc()
  tic()
  while !isempty(q) && (u = q.nodes[1].handle) != endnode
    pop!(q)
    poped[u] = true

    if !haskey(naboer, u)
      continue
    end

    for row in naboer[u]
```

```
31        if (to = db[row, "tonode"]) == u
32          to = db[row, "fromnode"]
33          linkcost = db[row, "cost_bw"]
34        else
35          linkcost = db[row, "cost_fw"]
36        end
37        cost = dist[u] + linkcost
38
39        if !poped[to] && cost < dist[to]
40          dist[to] = cost
41          update!(q, to, cost)
42          path[to] = db[row, "id"]
43          parent[to] = u
44        end
45      end
46    end
47
48    edges = Int[]
49    i = endnode
50    while i != startnode
51      unshift!(edges, path[i])
52      if path[i] == 0
53        return [], Inf
54      end
55      i = parent[i]
56    end
57    toc()
58    return (edges, dist[endnode])
59  end
```

```
1  using LightXML
2  using DataFrames
3
4  #xdoc = parse_file("trf.xml")
5  #xdoc = parse_file("stng2.xml")
6  xdoc = parse_file("minnesund.xml")
7
8  xroot = root(xdoc)
9
10 iter=0
11 while name(xroot) != "messages"
12   for c in child_elements(xroot)
13     iter += 1
14     iter < 10 ?
15     xroot = c :
16     (println("can't find messages"); return)
17   end
18 end
19
```

```
20 msg = collect(child_elements(xroot))
21 len = length(msg)
22
23 mnr = Array(Int, len)
24 version = Array(Int, len)
25 heading = Array(String, len)
26 ingress = Array(String, len)
27 mtype = Array(String, len)
28 roadid = Array(String, len)
29 srid = Array(Int, len)
30 start_x = Array(FloatingPoint, len)
31 start_y = Array(FloatingPoint, len)
32 end_x = Array(FloatingPoint, len)
33 end_y = Array(FloatingPoint, len)
34 stime = Array(String, len)
35 etime = Array(String, len)
36 kl = Array(Array, len)
37 nl = Array(Array, len)
38
39
40 for i in 1 : len
41
42   #find messagenumber, version and messagetype
43   mnr[i] = integer(content(find_element(msg[i], "messagenumber")))
44   version[i] = integer(content(find_element(msg[i], "version")))
45   heading[i] = utf8(content(find_element(msg[i], "heading")))
46   ingress[i] = utf8(content(find_element(msg[i], "ingress")))
47   mtype[i] = utf8(content(find_element(msg[i], "messageType")))
48
49   #find roadid
50   ty = first(content(find_element(msg[i], "roadType")))
51   nr = content(find_element(msg[i], "roadNumber"))
52   roadid[i] = string("{", ty, nr, "}")
53
54   #find coordinate system
55   coord = find_element(msg[i], "coordinates")
56   crs_full = content(find_element(coord, "crs"))
57   srid[i] = integer(match(r"\d+", crs_full).match)
58
59   #finding start coord
60   startp = find_element(coord, "startPoint")
61   start_x[i] = float(content(find_element(startp, "xCoord")))
62   start_y[i] = float(content(find_element(startp, "yCoord")))
63
64   #find end coord
65   endp = find_element(coord, "endPoint")
66   if typeof(endp) == XMLElement
67     end_x[i] = float(content(find_element(endp, "xCoord")))
68     end_y[i] = float(content(find_element(endp, "yCoord")))
```

```
  else
    end_x[i] = start_x[i]
    end_y[i] = start_y[i]
  end

 #find times
 stime[i] = content(find_element(msg[i], "validFrom"))
 et = find_element(msg[i], "validTo")
 if typeof(et) == XMLElement
   etime[i] = content(et)
 else
   etime[i] = "unknown"
 end

 kl[i], nl[i] = koord2list(db,
                       start_x[i], start_y[i],
                       end_x[i], end_y[i],
                       srid[i], roadid[i])
end



trfdata = DataFrame(
                   mnr = mnr,
                   version = version,
                   heading = heading,
                   ingress = ingress,
                   mtype = mtype,
                   roadid = roadid,
                   srid = srid,
                   start_x = start_x,
                   start_y = start_y,
                   end_x = end_x,
                   end_y = end_y,
                   stime = stime,
                   etime = etime,
                   kl = kl,
                   nl = nl)

type1data = [
"Midlertidig Stengt",
"Vinterstengt",
"Kolonnekjøring"
] # etc.

noder = Dict()
kanter = Dict()
for (n, mnr) in enumerate(trfdata["mnr"])
  mtype = trfdata["mtype"][n]
```

```
118
119   # hvis type 1 data:
120   if (in(mtype, type1data)
121      && length(trfdata["kl"][n]) > 1)
122
123     for node in trfdata["nl"][n]
124       if !haskey(noder, node)
125         noder[node] = [mnr]
126       else
127         push!(noder[node], mnr)
128       end
129     end
130
131   # hvis type 2 data eller kun en kant
132   else
133     for kant in trfdata["kl"][n]
134       if !haskey(kanter, kant)
135         kanter[kant] = [mnr]
136       else
137         push!(kanter[kant], mnr)
138       end
139     end
140
141   end
142 end
```

Tillegg B

Node-timing

```
1  CREATE TABLE noder (
2    node  INTEGER,
3    roadid VARCHAR(256),
4    wkb_geometry GEOMETRY
5    );
6
7  CREATE TABLE n_tmp (
8    node  INTEGER,
9    roadid VARCHAR(256),
10   geom  GEOMETRY
11   );
12
13
14 INSERT INTO n_tmp (node, roadid, geom)
15 SELECT fromnode, roadid, ST_Startpoint(wkb_geometry) FROM
       ruttger_link_geom;
16
17 INSERT INTO n_tmp (node, roadid, geom)
18 SELECT tonode, roadid, ST_Endpoint(wkb_geometry) FROM ruttger_link_geom;
19
20 INSERT INTO noder (node, roadid, wkb_geometry)
21 SELECT DISTINCT node, roadid, geom FROM n_tmp;
22
23 CREATE INDEX nodergeom_idx ON noder USING GIST (wkb_geometry);
24
25 DROP TABLE n_tmp;
```

```
1  eh@eh-W350STQ-W370ST:~$ psql -d vei01 -c "select count(*) from noder"
2    count
3  ---------
4   1306830
```

```
(1 row)

eh@eh-W350STQ-W370ST:~$ time psql -d vei01 -c
       "SELECT node FROM noder ORDER BY ST_Distance(
     ST_GeomFromText('POINT(32693 6838848)'
     , 32633), wkb_geometry) ASC LIMIT 1"
  node
--------
 306767
(1 row)


real 0m0.679s
user 0m0.035s
sys 0m0.013s

eh@eh-W350STQ-W370ST:~$ psql -d vei01 -c "select count(*) from noder_rd"
  count
---------
 2042785
(1 row)

eh@eh-W350STQ-W370ST:~$ time psql -d vei01 -c
     "SELECT node FROM noder_rd WHERE roadid='{F13}'
     ORDER BY ST_Distance(
     ST_GeomFromText('POINT(32693 6838848)'
     , 32633), wkb_geometry) ASC LIMIT 1"

  node
--------
 302520
(1 row)


real 0m0.258s
user 0m0.036s
sys 0m0.012s

eh@eh-W350STQ-W370ST:~$ time psql -d vei01 -c
  "WITH selection AS
    (WITH line AS
      (SELECT fromnode, tonode, wkb_geometry FROM ruttger_link_geom
      ORDER BY ST_Distance(ST_GeomFromText(
      'POINT(32693 6838848)', 32633), wkb_geometry)
      ASC LIMIT 1)
    SELECT fromnode node, ST_STARTPOINT(wkb_geometry) geom FROM line
    UNION
    SELECT tonode node, ST_ENDPOINT(wkb_geometry) geom FROM line)
  SELECT node FROM selection
```

```
  ORDER BY ST_Distance(ST_GeomFromText('POINT(32693 6838848)',
  32633), geom) ASC LIMIT 1"

  node
--------
 302520
(1 row)


real 0m1.110s
user 0m0.034s
sys 0m0.011s

eh@eh-W350STQ-W370ST:~$ time psql -d vei01 -c
  "WITH selection AS
    (WITH line AS
      (SELECT fromnode, tonode, wkb_geometry FROM ruttger_link_geom
      WHERE roadid = '{F13}'
      ORDER BY ST_Distance(ST_GeomFromText(
      'POINT(32693 6838848)', 32633), wkb_geometry)
      ASC LIMIT 1)
    SELECT fromnode node, ST_STARTPOINT(wkb_geometry) geom FROM line
    UNION
    SELECT tonode node, ST_ENDPOINT(wkb_geometry) geom FROM line)
  SELECT node FROM selection
  ORDER BY ST_Distance(ST_GeomFromText('POINT(32693 6838848)',
  32633), geom) ASC LIMIT 1"

  node
--------
 302520
(1 row)


real 0m0.478s
user 0m0.037s
sys 0m0.014s


eh@eh-W350STQ-W370ST:~$ psql -d vei01 -c
      "(SELECT fromnode, tonode FROM ruttger_link_geom
      ORDER BY ST_Distance(ST_GeomFromText(
      'POINT(32693 6838848)', 32633), wkb_geometry) ASC LIMIT 1)"

 fromnode | tonode
----------+--------
   302520 | 302521
(1 row)
```

Tillegg C

SRID på NVDB

```
julia> using SQLite

julia> co = SQLite.connect(
       "stud/master/julia/trfnetwork_20131015.sqlite")

sqlite connection
-----------------
File: stud/master/julia/trfnetwork_20131015.sqlite
Connection Handle: Ptr{Void} @0x00000000095e46a8
Contains resultset? No

julia> blob = query(
       "SELECT hex(SHAPE) FROM ruttger_link_geom LIMIT 1")[1,1]

"0001BE0B00005C8FC2F5320C0F410AD7A30049E25941DB1E85EB7D110F4114AE470
186E259417C020000000700000005C8FC2F5320C0F4114AE470186E25941310040E1A
E0C0F41050085DB7BE259412E0020AE070D0F41FC7F66B676E259410800A070010E0
F41000000B06CE2594122000000DA0F0F410100B8CE57E259410600400A11110F410
100B84E4CE25941DB1E85EB7D110F410AD7A30049E25941FE"

julia> if blob[3:4] == "01" # little endian
        srid = parseint(Int,
              bytes2hex(flipud(hex2bytes(
              blob[5:12]
              ))), 16) # reverse byteorder
                       # and convert hex to decimal
       end

3006
```

[32]

```
eh@eh-W350STQ-W370ST:~/stud/master/julia$ gdalsrsinfo epsg:32633 -o
    wkt_simple
PROJCS["WGS 84 / UTM zone 33N",
    GEOGCS["WGS 84",
        DATUM["WGS_1984",
            SPHEROID["WGS 84",6378137,298.257223563]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",15],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",0],
    UNIT["metre",1]]

eh@eh-W350STQ-W370ST:~/stud/master/julia$ gdalsrsinfo epsg:3006 -o
    wkt_simple
PROJCS["SWEREF99 TM",
    GEOGCS["SWEREF99",
        DATUM["SWEREF99",
            SPHEROID["GRS 1980",6378137,298.257222101],
            TOWGS84[0,0,0,0,0,0,0]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",15],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",0],
    UNIT["metre",1]]
```

```
julia> begin
         # Gitt f = (a-b) / a
         # f = flattrykking
         # a = store halvakse
         # b = lille halvakse

         a_WGS84 = a_GRS80 = 6378137
         f_WGS84 = 1/298.257223563
         f_GRS80 = 1/298.257222101
         b(a, f) = a * (1 - f)
         b_WGS84 = b(a_WGS84, f_WGS84)
         b_GRS80 = b(a_GRS80, f_GRS80)
         println("differanse i lille halvakse: ",
                 signif(b_WGS84 - b_GRS80, 1),
                 " meter")
```

```
16         end
17
18  differanse i lille halvakse: 0.0001 meter
```

Tillegg D

Tidtaking BBOX på Dijkstra-søk

```
julia> lindesnes = closestnode(30689, 6454809)
172871

julia> kristiansand = closestnode(85590, 6464160)
554903

julia> kirkenes = closestnode(1077014, 7802250)
494670

julia> begin
       tic()
       query("SELECT * FROM pgr_dijkstra(
            'SELECT ogc_fid as id, fromnode as source, tonode as target,
            drivetime_fw as cost, drivetime_bw as reverse_cost
            FROM ruttger_link_geom',
            $lindesnes, $kirkenes, true, true)");
       toc()
       end
elapsed time: 1.475241624 seconds
1.475241624

julia> begin
       tic()
       query("SELECT * FROM pgr_dijkstra(
            'SELECT ogc_fid as id, fromnode as source, tonode as target,
            drivetime_fw as cost, drivetime_bw as reverse_cost
            FROM ruttger_link_geom',
            $lindesnes, $kristiansand, true, true)");
```

```
29        toc()
30        end
31  elapsed time: 1.43822449 seconds
32  1.43822449
33
34  julia> begin
35        tic()
36        query("SELECT * FROM pgr_dijkstra(
37              'SELECT ogc_fid as id, fromnode as source, tonode as target,
38              drivetime_fw as cost, drivetime_bw as reverse_cost
39              FROM ruttger_link_geom
40              WHERE wkb_geometry && ST_MakeEnvelope(
41              24068, 6449689, 104800, 6485824, 32633)',
42              $lindesnes, $kristiansand, true, true)");
43        toc()
44        end
45  elapsed time: 0.086904533 seconds
46  0.086904533
```

Bibliografi

[1] B. Baas. Nosql spatial, neo4j vs postgis. Master's thesis, Geographical Information Management and Applications (GIMA), 2012. http://www.baasgeo.com/wp-content/uploads/2013/03/MSc_report_final_v101.pdf.

[2] DatexII. Datex2. http://www.datex2.eu/, 2014. Accessed: 15.02.2014.

[3] B. Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms. *Rapport technique, Massachusetts Institute of ...*, pages 1–13, 2004.

[4] D. Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[5] GDAL. Gdal - geospatial data abstraction library. http://www.gdal.org/, 2014. Accessed: 7.03.2014.

[6] GDAL. Gdal:ogr2ogr. http://www.gdal.org/ogr2ogr.html, 2014. Accessed: 7.03.2014.

[7] GiST. Introduction to gist. http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml, 2014. Accessed: 7.03.2014.

[8] Google. Gsoc 2011. http://www.google-melange.com/gsoc/homepage/google/gsoc2011, 2014. Accessed: 15.04.2014.

[9] Julia. The julia language. http://julialang.org/, 2014. Accessed: 20.04.2014.

[10] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *I V H S Journal*, 1(1):1–11, 1993.

[11] Lantmateriet. *UTM - Lantmateriet*, 2014. http://www.lantmateriet.se/Kartor-och-geografisk-information/GPS-och-geodetisk-matning/Om-geodesi/Kartprojektioner/UTM/, Accessed: 20.03.2014.

[12] S. N. Leksikon. database - it. http://snl.no/database/IT, 2014. Accessed: 15.02.2014.

[13] J. Mahadeokar. Time dependent dynamic shortest path. https://github.com/pgRouting/pgrouting/wiki/Time-dependent---Dynamic-Shortest-Path, 2014. Accessed: 15.04.2014.

[14] G. Nannicini. *Point-to-point shortest paths on dynamic time-dependent road networks*. PhD thesis, Ecole Polytechnique ParisTech, 2009. http://pastel.archives-ouvertes.fr/docs/00/50/12/51/PDF/final-thesis.pdf.

[15] OGC. Ogc sfsql. http://www.opengeospatial.org/standards/sfs, 2014. Accessed: 15.04.2014.

[16] OGP. European petroleum survey group. http://www.epsg.org/, 2014. Accessed: 15.04.2014.

[17] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[18] pgRouting. *pgRouting Manual*, 2.0.0 edition, 2013. http://docs.pgrouting.org/, Accessed: 21.01.2014.

[19] pgRouting. issue 243, pgrouting. https://github.com/pgRouting/pgrouting/issues/243, 2014. Accessed: 15.04.2014.

[20] pgRouting. pgrouting dijkstra code. https://github.com/pgRouting/pgrouting/tree/master/src/dijkstra/src, 2014. Accessed: 15.04.2014.

[21] PostgreSQL. Other pl. `http://www.postgresql.org/docs/9.3/static/external-pl.html`, 2014. Accessed: 15.04.2014.

[22] PostgreSQL. Pl handler. `http://www.postgresql.org/docs/9.3/static/plhandler.html`, 2014. Accessed: 15.04.2014.

[23] PostgreSQL. Pl/pgsql. `http://www.postgresql.org/docs/8.3/static/plpgsql.html`, 2014. Accessed: 15.04.2014.

[24] PostgreSQL. Postgresql- index types. `http://www.postgresql.org/docs/9.3/static/indexes-types.html`, 2014. Accessed: 15.04.2014.

[25] PostgreSQL. Postgresql, about. `http://www.postgresql.org/about/`, 2014. Accessed: 7.03.2014.

[26] PostgreSQL. Postgresql faq. `http://www.postgis.org/docs/PostGIS_FAQ.html#id560450`, 2014. Accessed: 15.04.2014.

[27] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases with application to GIS*. Morgan Kaufmann, 2002.

[28] H. Seki, Y. Jacolin, D. Kastl, and F. Junod. pgrouting workshop. `http://workshop.pgrouting.org/chapters/shortest_path.html`, 2014. Accessed: 27.04.2014.

[29] J. Siek, L.-Q. Lee, and A. Lumsdaine. Bgl dijkstra visitor concept. `http://www.boost.org/doc/libs/1_36_0/libs/graph/doc/DijkstraVisitor.html`, 2014. Accessed: 15.04.2014.

[30] J. Siek, L.-Q. Lee, and A. Lumsdaine. Bgl visitor concepts. `http://www.boost.org/doc/libs/1_36_0/libs/graph/doc/visitor_concepts.html`, 2014. Accessed: 15.04.2014.

[31] J. Siek, L.-Q. Lee, and A. Lumsdaine. Boost graph library. `http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html`, 2014. Accessed: 15.04.2014.

[32] SQLite. *SQLite Internal BLOB-Geometry format*. `http://www.gaia-gis.it/gaia-sins/BLOB-Geometry.html`, Accessed: 20.03.2014.

[33] S. vegvesen. Hvor finner jeg vegnettsdata til na-

vigasjon? http://www.vegdata.no/2013/08/08/hvor-finner-jeg-vegnettsdata-til-navigasjon/, 2013. Accessed: 17.01.2014.

[34] S. vegvesen. Datex informasjonside. http://www.vegvesen.no/Trafikkinformasjon/Reiseinformasjon/Trafikkmeldinger/Videreformidling/Datex, 2014. Accessed: 14.02.2014.

[35] S. vegvesen. Reisetider statens vegvesen. http://www.reisetider.no/reisetid/forside.html, 2014. Accessed: 7.03.2014.

[36] S. vegvesen. Retningslinjer for viderformidling av trafikkmeldinger. http://www.vegvesen.no/Trafikkinformasjon/Reiseinformasjon/Trafikkmeldinger/Videreformidling/Teknisk+informasjon, 2014. Accessed: 14.02.2014.

[37] Wikipedia. B-tree. http://en.wikipedia.org/wiki/B-tree, 2014. Accessed: 15.04.2014.

[38] Wikipedia. Dijkstras algorithm. http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, 2014. Accessed: 7.03.2014.

[39] Wikipedia. Postgresql. http://en.wikipedia.org/wiki/PostgreSQL, 2014. Accessed: 7.03.2014.

[40] Wikipedia. R-tree. http://en.wikipedia.org/wiki/R-tree, 2014. Accessed: 15.04.2014.

[41] Wikipedia. Relational model. http://en.wikipedia.org/wiki/Relational_model, 2014. Accessed: 15.04.2014.

[42] Wikipedia. Sql. http://en.wikipedia.org/wiki/SQL, 2014. Accessed: 15.04.2014.

Norges miljø- og
biovitenskapelige
universitet

Postboks 5003
NO-1432 Ås
67 23 00 00
www.nmbu.no