ANVENDT INFORMATIKK

eNytt

OPPTRYKK AV STUDENTARBEIDER VED
HØGSKOLEN I MOLDE
INSTITUTT FOR INFORMATIKK

JUNI 2005
NUMMER 2

REDAKSJON:
JUDITH MOLKA-DANIELSEN
HALVARD ARNTZEN

INNHOLD

Anmerkninger fra redaktørene:

Denne utgaven av Anvendt Informatikk eNytt inneholder fire utvalgte studentprosjekter ved institutt for informatikk i vårsemesteret 2005. Rapprotene som er presentert kan være noe forkortet i forhold til originalarbeidene, men de er ellers ikke redigert eller modifisert av redaksjonen eller andre.

Stikkeord for de presenterte arbeidene er: identifisering, ruteplanlegging, RFID (*Radio Frequency Identification*) og IT-sporing.

1. TEACHER EXCHANGE AV ANNE KARIN WALLACE

MOLDE UNIVERSITY COLLEGE – BRNO UNIVERSITY OF TECHNOLOGY.

Molde University College in Norway has an agreement with Brno University of Technology of the Czech Republic for teacher exchange. The exchange supported by the Erasmus program and allows teachers from Molde to visit Brno for one week for teaching. Teachers from Brno visit Molde University College for the same reason.

In November/December 2004, Anne Karin Wallace an Assistant Professor in the Informatics Departement at Molde University College spent one week in Brno to teach computer programming for students in the Mathematical Engineering Department. The students participating were in the first and second year of studies. They were attending a course of computer graphics, and the teaching was to fit in that context. The teachers from Brno who participated were RNDr. Pavel POPELA and Doc. Paed Dr. Dalibor MARTIŠEK.

TAGGING AND TRACING OF ANIMALS.

Figure 1. Labels show the country of origin and batch number of a meat product.

In Molde taught courses have been with emphasis on project work and practical work in groups, and this way of teaching was chosen for the lessons in Brno. The topic chosen for the project was "Tagging and tracing of animals". In my class in Norway it is relevant to make applications for registration of birth, death and moving of animals. The data registered are to be stored in a database which can be accessed through the Internet. Retrieval of data from the database for different kinds of use is also relevant, for instance to create reports on the locations of animals infected with different kinds of diseases.

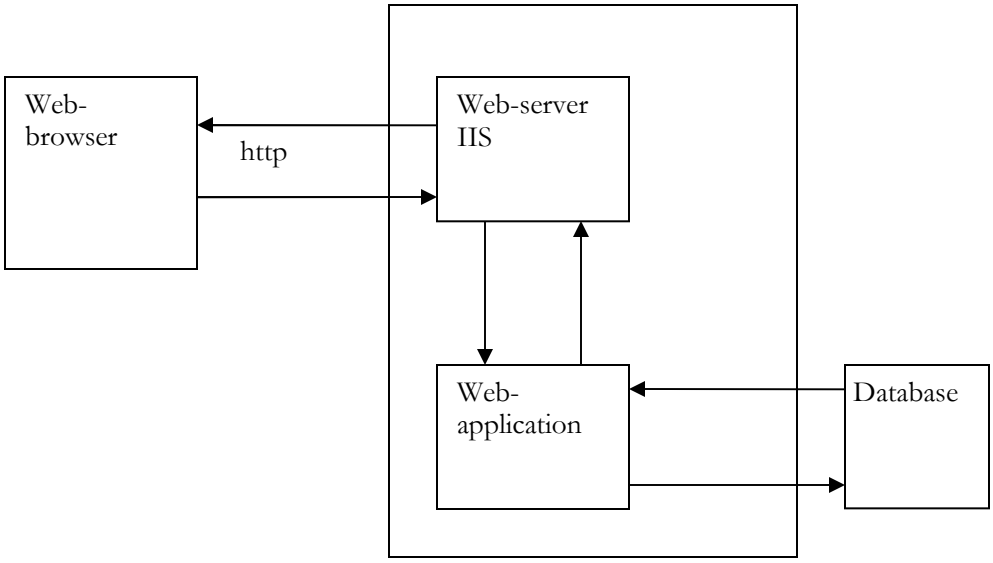


Figure 2. The principle behind a web-application.

In a lecture they were presented for the reasons for tagging and tracing animals. This is mainly motivated by food-safety, but also food-quality and breeding is relevant. Technology for tagging and for registration (bar code, RFID) was discussed. A prototype for a web-program for registration of data in a database and retrieving data from the database was demonstrated.

Figure 3. Calf with human-readable ear-tags.

Figure 4. RFID-tag.

VISUALIZATION OF THE DATA RECORDED

For the Czech students the relevant problem was to visualize the data in the database. The students made groups and worked on the project for three days (in addition to attending their normal classes).

They were given a map of "Møre og Romsdal" county and the content of the database. It contained data on 10 farms, one slaughterhouse and 100 animals. The students were free to do the visualization in the way they wanted. Farms were located at the map in areas suitable for farming and the number of animals in each location were represented in different ways. Some visualized the spread of a disease, and we also saw animation of the transport of an animal along roads from one location to another. The programming language used was Delphi.

Figure 5. Group 1 has concentrated on placing the farms on the map and registration of data about the farm.

Group 1 made a project where the user can visually put the farms on the map and add some data like the number of animals and the number of infected animals. The farms must be placed in areas suitable for farming (not in water, not in mountain areas). The farms are marked with nice farm-houses.

The program made by group 2 places the farms randomly on the map in areas suitable for farming. They are marked by circles. The size of the circle indicates the number of animals in the farm. The slaughterhouse (Gilde) is marked with a pink circle. By placing the mouse on a farm, information about the farm (Farm-identification and the number of animals) is shown.

Group 3 has hard-coded the locations of the farms, they are put in areas suitable for farming. The slaughterhouse is in Molde. They use bars and numbers next to each farm to show the number of animals present. The number of infected animals in each location is also shown. When clicking on a farm detailed information on the animals here is shown. This project also has the possibility of animation of the moving of animals along roads. In addition a simulation of diffusion of a disease can be done.

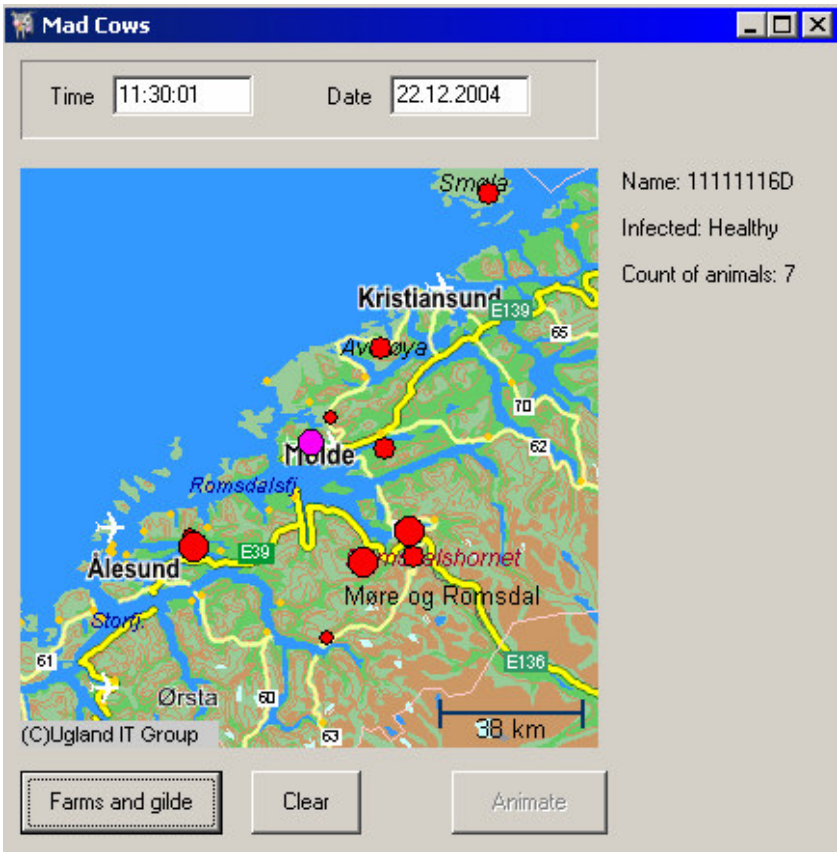


Figure 6. Group 2 visualizes the number of animals at each farm by using the size of a circle.

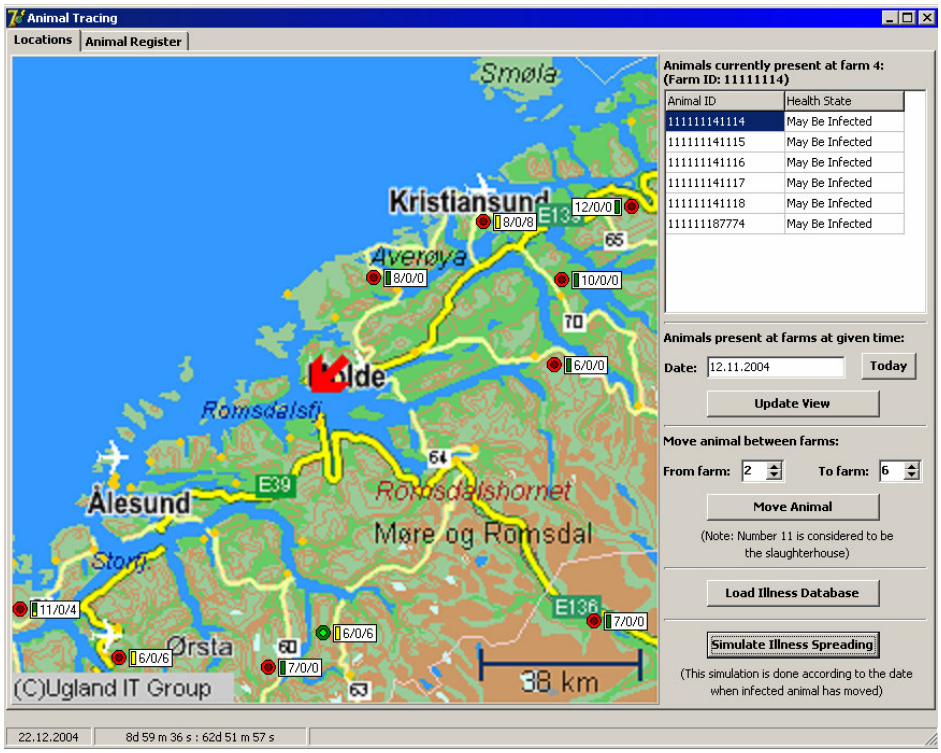


Figure 7. Group 3 has made a simulation of spread of a disease and shows the state of the animals after simulation.

The projects were presented for the teachers and the class at the end of the week, the presentation was done in English.

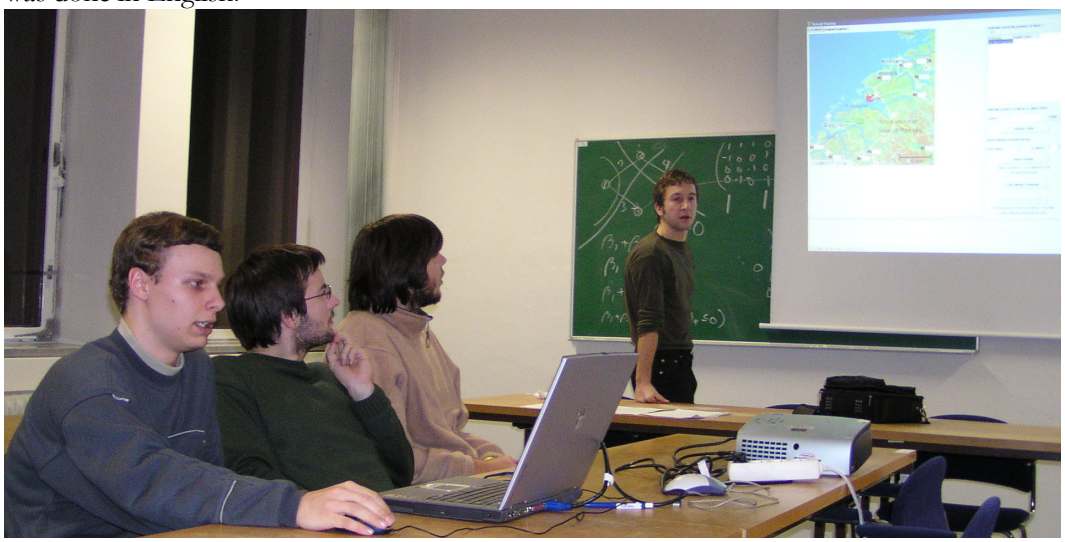


Figure 8. Group 3 presents their work.

FURTHER WORK

Tagging and tracing of animals and food is also the subject for the project in the course IN151 Programming practice taught in the spring semester 2005. In this course the focus is on the database and a web-application for registration and retrieving of data.

The teacher exchange program continued in 2005. Two teachers from Brno visited Molde in the spring semester. They both teach at our Masters program.

---

2. LO205: BIOMETRI AV IVER FAUKSTAD

Tema: Biometri
Navn: Iver Faukstad

Beskriv/ definer hva biometri er og hvor man bruker det til.

Biometri kommer av de to latinske ordene bios som betyr liv og metri som betyr måle. Detta innebærer altså at biometri betyr "å måle liv", dvs måte noe levende.
(http://www.lysator.liu.se/~ngn/specarb/ngnspec8.html)

Biometri er en samlebetegnelse for metoder for å gjenkjenne en person basert på oppførselsmessige og fysiologiske forhold. Dvs oppførselsmønsteret til en person, men også de individuelle fysiologiske trekkene hver enkelt individ har, som er unikt for den enkelte. Stikkord er; skanning av iris, fingeravtrykk, håndflåte, ansikt, stemme, tastaturbruk, duft, signatur, øre og andre områder som har sin spesielle særegenhet hos det enkelte individ.

Biometri blir mer og mer aktuelt ettersom behovet for identifisering forsterkes for hver dag som går. Det handler ikke bare om å identifisere den enkelte personen til en hver pris. Det skal være effektivt, og kostnadseffektivt, og ikke minst mest mulig sikkert ift at det blir større og større krefter som

jobber med å knekke "kodene" for identifisering. Identifiseringen jeg sikter til er bl.a. nøkler, PIN-koder og/eller (bank)kort. Dette er vel de identifikasjonstypene vi kjenner best til i dag. Disse identifikasjonene kan mer eller mindre enkelt utnyttes av uvedkommende. Et mastercard er enkelt å misbruke dersom det kommer i uvedkommendes hender. Det samme er koder og egentlig mer eller mindre alle sikkerhetstiltak som ikke kommer inn under betegnelsen biometri. Biometri tar sikte på å identifisere personen etter personens kjennetegn, altså kan kun den enkelte person som det gjelder, knekke koden for eks å ta ut penger fra mastercardet sitt, eller ta seg inn på kontoret sitt/jobben. Identifisering er å gjenkjenne folk for å være sikker på at en person er den ha utgir seg for å være. Autentisering er prosessen som kontrollerer identifiseringen. For eksempel: Et pass identifiserer hvem vi er (navn), og med bildet i passet autentiserer passkontrolløren identiteten. Med biometri lagret på et smartkort er det mulig å identifisere lokalt (fingeravtrykk mot data lagret i chipsen), mens autentiseringen går mot en sentral server. Dermed er det ikke nødvendig at (biometriske) data ligger lagret på en plass andre har tilgang til.
(http://www.computerworld.no/index.cfm?fuseaction=artikkel&cwid=4A532A94D9D5ACFEC12569F50003201B)

Behovet for biometriske løsninger er aktuelt overalt, og kanskje særlig innen finansbransjen, Forsvaret, Stortinget, detalj handel og større selskaper, men også innen helse og sosial tjenesten, som alt har kommet et stykke på vei i forhold til (ift) å implementere denne nye teknologien. De siste 10 årene har Internett fått en større og større rolle. Derfor er i stor grad god identifisering på nettet viktig som forsikring ift handel og andre tjenester på nettet som innlegging på nettbank osv. Benyttelse av dette alene eller sammen med teknologiske "nyvinninger" som kryptokort og digitale signaturer berører biometri mange aspekter i økopnomien til daglig
(http://www.biometrics.org/html/introduction.html)

Hva slags teknologi brukes ifm biometri?

Nedenfor har jeg tatt for meg ulik teknologi og deres fordeler, ulemper og bruksområder. som benyttes ifm biometri.

Iris

- En videokamera tar på noen sekunder et opptak av irisen (eller retina, netthinnen i øyet). En iris tilsvarer mer informasjon enn menneskelig DNA.
- Fordeler: Den mest pålitelige måten å bruke biometri på.
- Ulemper: Trenger i høy grad samarbeid av personene som identifiseres, omstendelig, iris/retina kan endre seg ved øyesykdommer, tar tid, svært dyr
- Bruksområder: Inngangsidentifikasjon for små grupper, butikker, minibanker

Fingeravtrykk

- En skanner leser mønsteret på en av fingrene. Alle mennesker har unike fingeravtrykk, teknikken brukes allerede mye.
- Fordeler: Rask og enkel, ganske pålitelig, hvis vi ser bort i fra de billigste skannerne til rundt 2000,-
- Ulemper: Trenger samarbeid fra brukeren, forbindes av mange med kriminalitet, billige skannere kan gjøre feil, tåler vanligvis ikke veske (regn, svette)
- Bruksområder: Tilgang til bygninger, innlogging på nettverk, minibanker, sikring av datamaskin.

Håndflate

- Skanner mønsteret av hele hånden, som er unik. Måler lengde på fingrene, formen, tykkelsen av håndflate, osv.
- Fordeler: Rask og pålitelig.
- Ulemper: Trenger samarbeid fra brukeren, ringer og møkk på hendene kan gi utslag på skanneren, dyrere enn fingeravtrykk skanning
- Bruksområder: tilgang til bygninger, i USA til og med brukt i barnehagen

Ansikt

- En videokamera kan identifisere ansiktet på flere måter, for eksempel avstanden mellom øyene, nesen og munn. Med infrarød skanning kan varmeforskjeller i ansiktet eller mønsteret i blodårene måles. Kan skanne folk i forbifarten.
- Fordeler: Trenger i liten grad samarbeid fra personer, for eksempel, se til venstre mens du går forbi.
- Ulemper: Dyr, gjerne titusener av kroner
- Bruksområder: Tilgangskontroll for bygninger med store mengder mennesker, som flyplasser

Tastaturbruk

- Programvare som for eksempel NetNanny måler hvordan du bruker tastaturet, gjerne hurtighet og rytme. På verdensbasis er det flere hundre patenter på denne type teknologi.
- Fordeler: Svært billig
- Ulemper: I praksis er den alt for upålitelig og har har lite gjennomslag.
- Bruksområder: Sikring av maskinvare, innlogging på nettverk og lignende

Stemme

- Stemmeverifikasjon måler munnen, stemmebåndene og bihulene utifra stemmen.
- Fordeler: Trenger ikke samarbeid fra brukerne
- Ulemper: Ikke veldig pålitelig, for eksempel ved støy rundt stemmen
- Bruksområder: Telefontjenester og postordrebedifter

Duft

En e-nese kjenner igjen menneskers kroppslukt, som er bygget opp av forskjellige kjemikalier. Verdiene gjøres om til elektroniske pulser, som sammensatt gir et unikt personlig bilde. Teknologien er i utviking, og det er ikke tatt i bruk ennå.

Signatur

To typer, den statiske og den dynamiske. Den statiske sammenligner en signatur med et lagret eksempel. Den dynamiske måler trykk, rytme og hastighet.
(http://www.computerworld.no/index.cfm?fuseaction=artikkel&cwid=B67B85FB3F279821C12569F50003236F)

Hvordan nyttes biometrisk identifikasjon på - og utenfor Internett.

Utenfor Internett kan biometrisk identifikasjon benyttes på måtene som er nevnt tidligere. Det være seg: Iris, fingeravtrykk, håndflate, Signatur, Duft, ansikt, stemmebruk osv. Mens man i dag stort sett benytter kort med PIN - kode for å handle, vil man litt frem i tid sannsynligvis benytte seg av et utall ulike biometriske identifikasjons muligheter isteden fordi det blir påkrevd ifm økende krav til sikkerhet,(går ikke an å miste) at det er enkelt å benytte(ikke noe å bære rundt på) og at det har helt presist og nøyaktig funksjonering.

Når det gjelder biometri på Internett, så er det ikke like enkelt å innføre biometri, og jeg er noe usikker på hvilke biometriske momenter man kan benytte utover elektronisk signatur og webcam-bilde. Vi ser allikevel noen muligheter og områder det nok jobbes med for å dekke.

Den nye teknikken kan drastisk øke sikkerheten i automatiske adgangssystem. Så sant datasystemet som styrer det hele, har fått informasjon om hvordan du ser ut, vil systemet se på deg, og evt. kjenne deg igjen. Da slipper du inn, eller ikke…

Tenk deg du står i Minibanken: Du er registrert som innehaver av kontonummer xxxx, med foto osv.. Du stiller deg opp, stikker kortet i sprekken, et elektronisk øye stirrer på deg - og så kan du ta ut penger. (http://www.tu.no/nyheter/naturvitenskap/article26992.ece)

I år kunne foreksempel alle med Internettilgang levere sin selvangivelse på nettet. For å gjøre det måtte du benytte tre-3-pinkoder. Tenk så mye enklere det hadde vært om det var nok med en skanning av øyeeple, gjort på få sekunder? Eller du er sosialklient og skal kontakte din saksbehandler via Internett. Nå skal jo offentlige tjenester være tilgjengelig 24 timer i døgnet. Dersom både sosialklienten og saksbehandleren er pålogget via biometriske system vil begge parter kunne være trygg på at konfidensielle opplysninger forblir konfidensielle.

Digitale signaturer benyttes for å autentisere at begge parter er den de gir seg ut for å være. Men siden digitale signaturer ofte er basert på passord, ligger det en mulighet for at uautoriserte kan signere med din signatur. Med fingeravtrykk- og retinagjenkjenning derimot vil dette ikke bli et så stort autoriseringsproblem. At en vanlig bruker skal velge retinagjenkjenning bare for å slippe å skrive inn et windows-passord er tvilsomt. Men dersom det fremholdes som et konsept som effektivt kan sikre at ingen andre enn du kan få tilgang til din bankkonto via internett, vil det være et stort marked også blant oss ordinære databrukere.(http://it-mo.hinesna.no/~pag/ped/ithuset/biometri3.html)

Hvilke andre biometriske bruksområder kjenner du?

Som jeg har nevnt tidligere i besvarelsen så har man særlig innen helseomsorgen kommet et stykke ift implementering av biometri.

Et norsk eksempel er Nordisk Språkteknologi (NST) på Voss i samarbeid med IBM som har levert en automatisk dikteringsløsning som muliggjør at (røntgen)leger kan diktere journaler direkte til tekst på skjerm. Selv om dette er et språk vanlige mennesker neppe ville skjønne mye av, er det et begrenset vokabular, som gir høy gjenkjennelse. Radiologisystemet har en ordgjenkjenning på over 98 prosent og potensialet innen helsevesenet for diktering av journaler er stort sies det. Bare i Norge går det med 3500 årsverk i skrivestuer, og her tror jeg det kan ligge an til en revolusjon i produktivitet og raskere produksjon av den enkelte pasients journal og lignende. Språkteknologi slik som den vi har utviklet for (røntgen)leger kan ha potensiale til å redusere 20-30% av årsverkene ifølge personell innen denne teknologien.

Også andre store grupper vil ha glede av å kunne diktere til skjerm. Rundt 10 000 av de som er sykmeldt i dag er, klarer ikke å benytte et tastatur. Svært mange av disse kunne ha utført arbeidsoppgaver hvis de hadde fått tilgang til en talestyrt dikteringsløsning. (http://www.tu.no/nyheter/ikt/article24554.ece)

Dessuten har Europas første smarthus fått et system som identifiserer beboerne ved hjelp av fingeravtrykk. Systemet, som erstatter nøkler, PIN-kode og magnetkort, er montert i en villa utenfor. Dette kan også implementerers inn i helseomsorgen.
Stockholm.(http://www.tu.no/nyheter/ikt/article3078.ece)

Beskriv en internasjonal offentlig organisasjon innen biometri.

Jeg synes det var litt vanskelig å finn en offentlig internasjonal organisasjon og var noe usikker på hva du så for deg, men jeg fant frem til at staten er jo på sett og vis en internasjonal organisasjon. Jeg har valgt å vinkle besvarelsen på spm opp mot fremtidig bruk av biometri innen noe som i mine øyne er særs viktig, nemlig passkontroll.

Fremtidens pass er biometriske pass, smartkort, som inneholder bilde av passinnehaveren i tillegg til annen identitetsinformasjon. I en passkontroll vil ett bilde bli tatt av passeieren og sammenlignet med fotoet lagret i passet. En systemdefinert parameter vil avgjøre om disse to fotoene er like nok til å fastslå om personen i de to fotoene er en og samme person.

Standardiseringsarbeidet rundt biometri i pass og andre applikasjoner er i full gang. Men før myndigheter tar i bruk biometriske egenskaper til autentisering og identifikasjon av mennesker, kreves det mer forskning innen området. Det er en av konklusjonene til Marijana Kosmerlji,i hennes nye mastergradsavhandling . (www.telecom.no/showarticle.php?articleID=10420)

Beskriv en internasjonal privat organisasjon innen biometri.

Index er et Norsk firma som har spesialisert seg på gjenkjenningsteknologi innen fingeravtrykk. Målet deres er å bli ledende innen dette området. Deres plan er å designe revolusjonerende komponenter og deretter oppnå samarbeid med verdensledende aktører, som har organisasjon og fasiliteter til å masseprodusere og markedsføre det. Et eksempel på dette er samarbeidet med STMicrelectronics og SmartFinger
Teknologien er unik- og selvfølgelig patentert. Nå om dagen er de midt i en prosess hvor de søker å minimere teknologien, som vil resultere i en billig liten skanner av høy kvalitet. 53% av aksjene eies av nordmenn, mens resten eies av utenlandske investorer. Aksjen kan handles på "unotert"-listen" på Oslo børs, og aksjekursen økte med over 1500% fra 01.05.03 til 01.05.04. Siden den gang har den ligget stabilt. Selskapet har ingen inntekt, men baserer seg naturlig nok derfor på inntekter av de nyvinningene de nå gjennomfører. (http://www.idex.no/about.html)

Er biometri et marked for mange forskjellige aktører. hvorfor/hvorfor ikke?
Jeg mener biometri må kunne være et marked med fri konkurranse på lik linje med alle andre markeder. Det er naturlig at det vil dukke opp flere og sterkere aktører etter hvert som denne teknologien blir mer og mer anerkjent, og det viser seg at det er store penger å tjene her og nå, og ikke slik det er nå, hvor mye er basert på fremtidig usikker inntjening. Konkurranse setter jo i verk 100% innsats hos den enkelte bedrift til å produsere best mulig løsninger for å tjene best mulig og oppnå mest mulig positiv respons fra markedet, og dette vil jo være positivt for utviklingen av biometri.

Konklusjon

Jeg tror videre at det blir viktig å opprette visse standarder innen de enkelte områdene innen bioteknologi, av praktiske årsaker slik at det ikke blir et utall ulike standarder. Dette ser vi jo innen de fleste andre teknologier. Eks. har vi felles standarder ift DVD- spillere, batteristørrelse, og programmeringsspråk. Utover det ser jeg ingen spesielle grunner til at man ikke skal kunne ha

konkurranse mellom flere aktører i sin higen etter nye og forbedrede løsninger og jeg er meget spent på utviklingen de neste 10 årene, som jeg tror kan bli revolusjonerende.

3. IN380: ROBUSTE OG SIKRE SYSTEM – INTEGRITET AV FREDDY SKARBØ, ARNE GJENGSTØ OG VENKE AASGÅRD IVERSEN

Generelt om Integritet

Tidligere i kurset har vi diskutert dataintegritet, men nå i dette kapittelet blir det mer generelt om integritet. Til forskjell fra dataintegritet snakker vi nå typisk om systemintegritet. Systemintegritet kan sies å være avhengig av både konfidensialitet og tilgjengelighet som vi tidligere har diskutert i kurset. Troverdigheten til et IKT-system inkluderer:

- Informasjon som sendes
- Informasjon som lagres
- Kontroll- og styringsinformasjon
- At systemet virker som det skal, når det skal

Sending av informasjon skjer typisk ved e-post, en bestilling over nettet, overvåkning etc. Med kontroll- og styringsinformasjon mener vi ofte konfigurasjonsdata, systemer for drift osv. Hvordan oppnår vi så systemintegritet? Blant flere ting vi bør gjøre for å oppnå dette, finner vi risikovurderinger. En risikovurdering skal være et styringsverktøy for den som har ansvaret for informasjonssikkerheten. Vi må også ha handlingsplaner som følge av risikovurderingen, og også klargjøre juridiske problemstilliger ved kontrakter og avtaler. Man må klargjøre hva systemet skal gjøre og hvordan det skal oppføre seg, og hvem som har ansvaret for de forskjellige komponentene i systemet.

Undergrupper av integritet:
Data-integritet, som er en undergruppe av det generelle integritetsbegrepet
Personlig-integritet som sier noe om er en den en gir seg ut for, eller den andre sier at en er?
Integritet til bedrift skapes ved troverdig informasjon

I dette kurset brukes integritet som et samlebegrep for systemintegritet. IKT-systemet må være det eierene ønsker det skal være, og det er dette brukerene oppfatter at systemet er.

Risiko er et sentralt begrep i dette kapitellet. Risiko uttrykker en hypotese om den fare en hendelse representerer overfor verdiene i virksomheten. Risiko er kombinasjon av konsekvens av en hendelse og sannsynlighet for at den inntreffer. Dersom konsekvens beskrives som økonomisk tap av en viss størrelse, og sannsynlighet angir forventet hyppighet, vil risiko uttrykke totalt tap over tid. Vurdering av risiko er utgangspunktet for ethvert sikkerhetsarbeid, og sikkerhet er håndtering av eventuell risiko. Sikkerhetstiltakene må stå i forhold til sannsynligheten og konsekvensen av sikkerhetsbrudd. Som et eksempel kan vi bruke Høgskolen i Molde. Her brukes samme server både for mail og web, og det kan sies å være uheldig. Hva er risikoen dersom serveren svikter og studenter og ansatte blir uten både mail og web?

For å sikre oss mot feil som kan oppstå i systemet vårt har vi teknikker som kan deles i to grupper:
- Før feil (pro-aktivt)
- Etter feil

Før feilen har oppstått kan vi passe på å kjøpe gode komponenter, designe et robust nettverk, installere brannmur etc. Etter feilen blir det dynamisk håndtering, trafikkdirigering, logger, grenseverier og slike ting.

Personopplysningsforskriften står sentralt når vi diskuterer integritet. Denne gjelder for behandling av personopplysninger ved elektroniske hjelpemidler. Generelt så gjelder personopplysningsloven. Disse gjelder for tilfeller der der er:

- Fare for tap av liv og helse
- Fare for økonomisk tap
- Fare for tap av anseelse og personlig integritet
- Nødvendig å sikre konfidensialitet, tilgjengelighet og integritet for opplysningene

Personopplysningsforskiften sier også at det skal være en behandlingsansvarlig. Den behandlingsansvarlige MÅ gjennomføre en risikovurdering for å klarlegge risiko ved behandling av personopplysninger. Disse skal stå i forhold til behovet for konfidensialitet, tilgjengelighet og integritet. Resultatet av vurderingen skal sammenlignes med akseptabelt risikonivå fastsatt på forhånd. Riskonivået fastsettes av bedriften selv basert på egne regler, rammebetingelser og lover.

Ved fysisk sikring mener vi sikring av tilgangen til selve utstyret, bygningsmessinge tiltak etc. Når vi snakker om sikring av konfidensialitet så ønsker vi tiltak mot uautorisert innsyn. Med sikring av tilgjengelighet mener vi sikring av tilgang til informasjon av betydning for informasjonssikkerheten. Man må også forbedre alternativ behandling for de tilfeller informasjonssystemet ikke er tilgjengelig. Sikring av integritet vil si sikring mot uautorisert endring av personopplysninger der integritet er nødvendig, og det skal også treffes tiltak mot ødeleggende programvare. Hvilke opplysninger som skal sikres og hvilke tiltak som skal gjøres blir en følge av resultatet av risikovurderingen.

Det siste vi skal ha med i denne teksten er litt om risikovurdering, siden dette er sentralt innen dette temaet. En risikovurdering har ikke fokus på sikring av tekniske komponenter i et datasystem, men vi vurderer her risiko for hendelser som medfører brudd på sikkerhet i forhold til konfidensialitet, tilgjengelighet og integritet. Når vi vurderer risiko så vurderer vi konsekvens og frekvens av hendelser. Fasene i en slik risikovurdering er delt inn som følger:

- Beskrive systemet
- Trusselvurdering (Trusler og årsakene til truslene)
- Sannsynlighetsvurdering (Sannsynlighet for at truslene kan bli utløst)
- Konsekvensvurdering (Konsekvenser dersom truslene utløses)
- Resultat (Beskrivelse av risikobilde)

Etter dette vil vi ofte beskrive tiltak for å bringe risikobildet i overenstemmelse med akspetkriteriene, men dette er ikke en del av selve risikovurderingen.

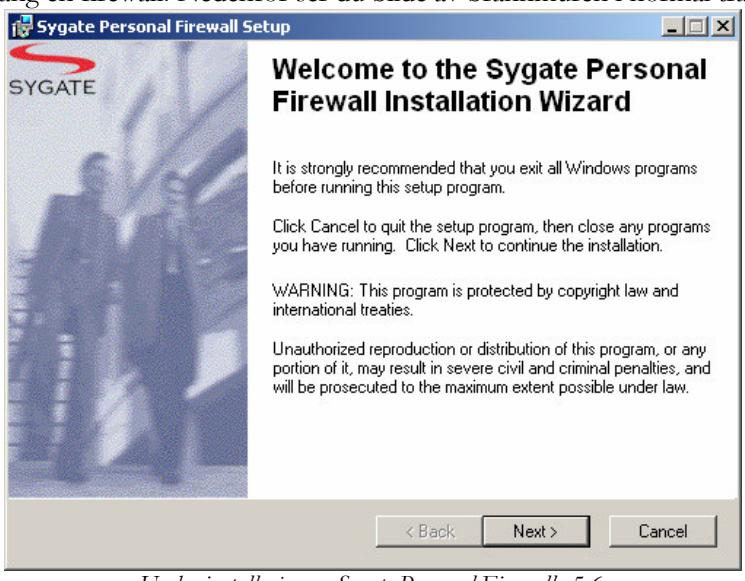
Del 1: Installer en personlig brannmur

I denne oppgaven skal vi installere en personlig brannmur på vår hjemme-pc. Jeg har tidligere hatt veldig dårlig erfaring med slike "sikkerhetspakker" som for eksempel Norton Internet Security fra Symantec og PC-cillin Internet Security fra Trend. Disse alt-i-ett pakkene stanser mye unødvendig trafikk på pc'en, og det blir ofte problemer med å sende på både e-post og direktemeldinger gjennom et program som for eksempel MSN Messenger.

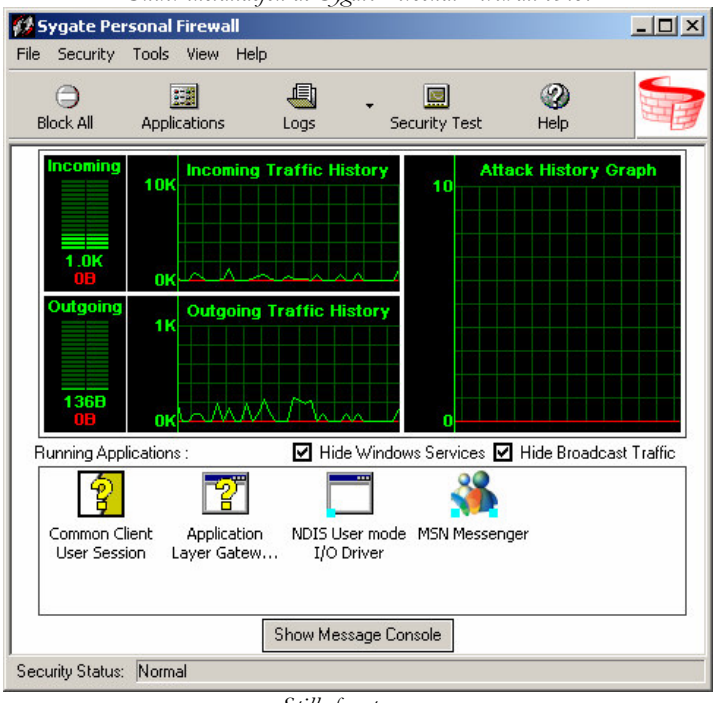
Jeg valgte derfor å installere en gratis brannmur (freeware) ved navn Sygate Personal Firewall. Dette er den mest brukte freeware-brannmuren i verden, og er utstyrt med en avansert logging-funksjon,

som er perfekt til vårt bruk i denne øvingen. Versjonen vi testet er v5.6 og hadde en nedlastingsstørrelse på 8.8 MB.

Etter installasjon måtte jeg restarte maskinen, og så var vi oppe å kjøre. Brannmuren stilte noen få spørsmål om hvilke applikasjoner som skulle få tilgang til internett. Fikk også opp tilgangsspørsmål første gang jeg startet MSN Messenger og Internet Explorer, men utenom dette så gikk det helt smertefritt å få i gang en firewall. Nedenfor ser du bilde av brannmuren i normal trafikk.



Under installasjon av Sygate Personal Firewall v5.6.

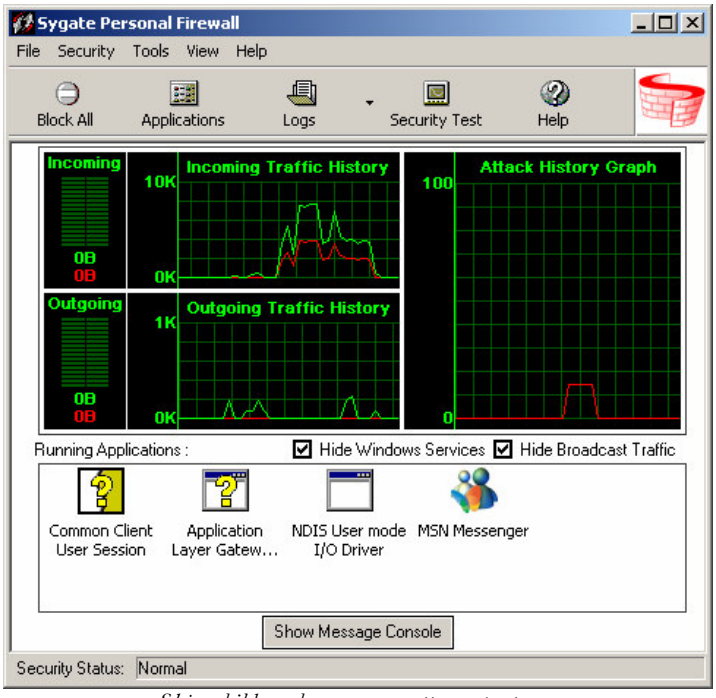


Stille før stormen.

Del 2: Test den personlige brannmuren

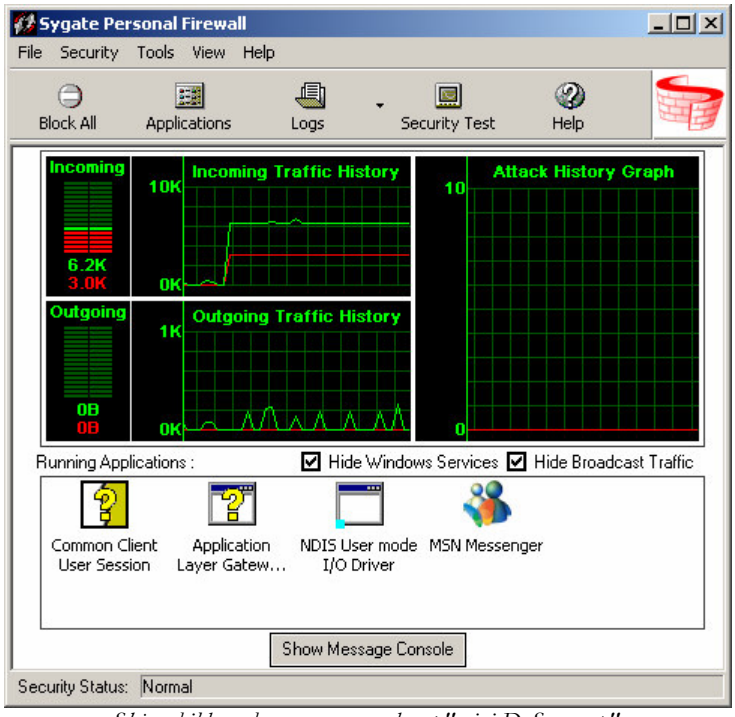
Og så begynner morroa. I del 2 av øvingen skal vi teste brannmuren mot forskjellige typer angrep, og se hva brannmuren logger av aktivitet inn mot datamaskinen vår. Eneste verktøyet vi fikk oppgitt vi kunne bruke var en portscanning ved hjelp av NMAP, men ved bruk av denne skjedde det ingenting. Heller ikke når vi prøvde windows-versjonen av samme applikasjon fikk vi til å scanne portene på en annen maskin. Heldigvis for oss så har en av gruppemedlememene vært en jævel før når han har vært på internett, og har massevis av stygge programmer vi kan teste ut i denne øvingen. Etter å ha deaktivert alt av antivirus-programmer så får vi faktisk lov å kopiere programmene inn på maskinen også, så da kan testingen begynne. Måten vi gjorde dette på er at jeg (Freddy) sitter på min hjemmepc med brannmur installert, og Arne sitter hjemme på sin hybel og prøver å angripe meg med diverse verktøy jeg sendte han.

Det første vi prøver er portscanning ved hjelp av et lite program som i all enkelhet blir kallt PortScan v1.2. Dette programmet scanner alle portene til datamaskinen og prøver å opprette en tilkobling til eventuelle ledige porter den finner. Etter at Arne startet denne scanningen mot min maskin gikk det ikke mange sekundene før jeg fikk en popup-beskjed om at et "Port Scan Attack" hadde blitt logget. Det som skjer da er at min firewall logger alle portene Arne prøver å koble seg til, og lukker disse slik at han ikke får tilgang. Beskjeden jeg får hvis jeg klikker for mer info om angrepet er: "Somebody is scanning your computer. Your computer's TCP ports: 949, 950, 957 and 951 have been scanned from 62.16.162.113.". Legg merke til at den grønne streken er innkommende trafikk, mens den røde er mengden trafikk som blokkeres.



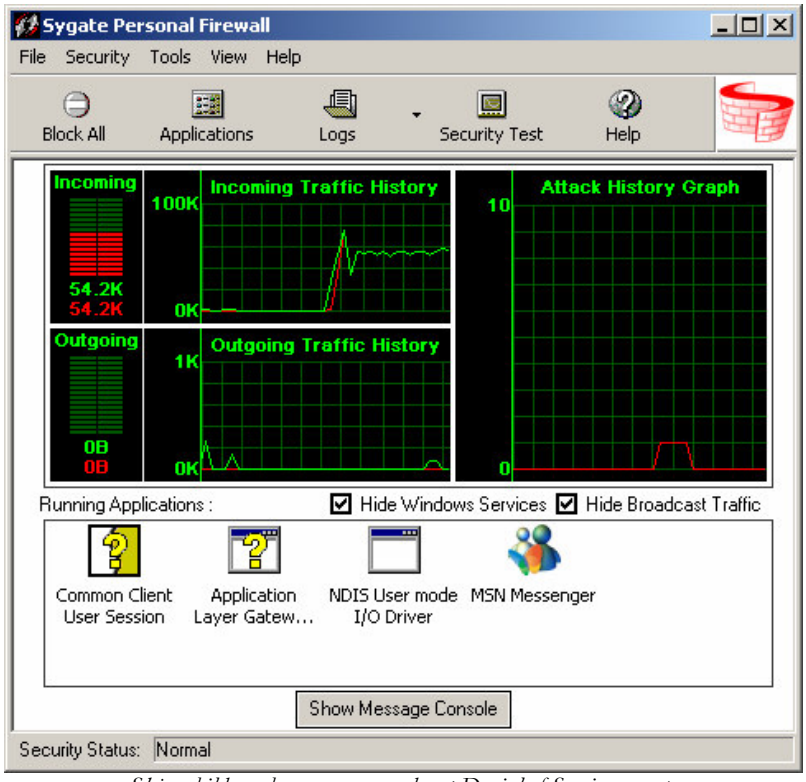
Skjermbilde av brannmuren etter en portscan.

Etter dette fikk vi lyst å bli litt mer destruktive. Vi prøvde et program som het X-Script ICMP Bomber, og er nøyaktig det det høres ut som. Et program hvor du "bomber" en annen maskin med større pakker enn det som er normalt, og forårsaker en Ping of Death, eller såkallt Denial of Service. Pakkene bruker ICMP-protokollen. I programmet oppgir du en ip-adresse til den du vil angripe, en pakkestørrelse og så hvor mange pakker du vil sende. Det første vi prøvde var en pakkestørrelse på 1000 byte og 10 pakker. Dette er ikke veldig mye, og burde ikke forårsake store "ødeleggelser" hos offerets maskin.



Skjermbilde av brannmuren under et "mini DoS-angrep".

Selv om vårt beskjedne angrep tydelig ga utslag på trafikken inn på datamaskinen ble det ikke tolket som et angrep, fordi pakkestørrelsen var for liten. Jeg ga derfor beskjed om å trykke på litt. Jeg ba om en pakkestørrelse på 15000 byte og 200 pakker, og da ble det reaksjon fra brannmuren. Beskjeden "Denial of Service 'Ping of Death' attack detected." kom raskt opp på skjermen, og som vi ser på bildet under er det mye innkommende trafikk som blokkeres.

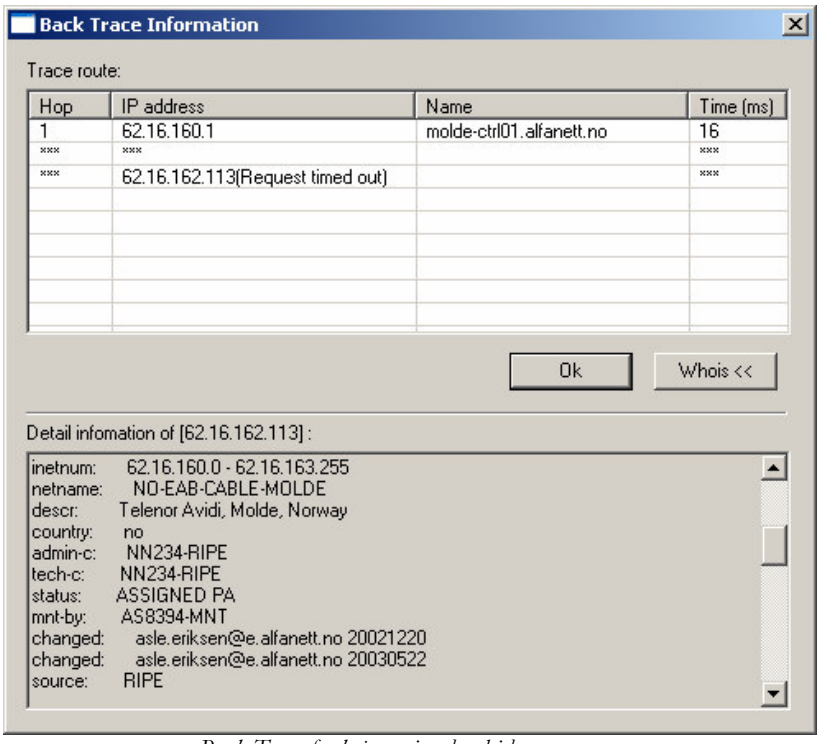


Skjermbilde av brannmuren under et Denial of Service angrep.

Time	Security Type	Severity	Direction	Protocol	Remote Host
27.04.2005 19:32:27	Denial of Service	Major	Incoming	ICMP	62.16.162.113
27.04.2005 19:10:58	Port Scan	Minor	Incoming	TCP	62.16.162.113

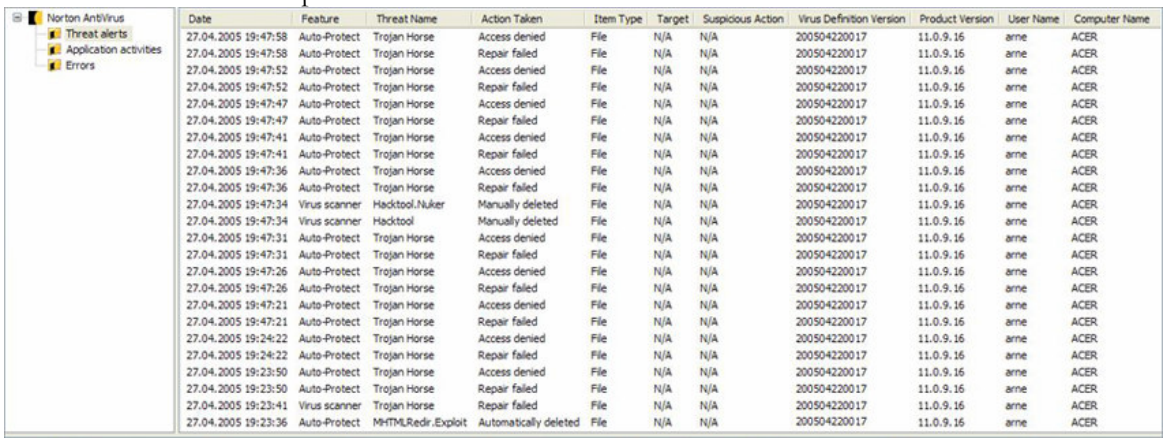
Logg av de 2 angrepene utført så langt.

I oppgaven skal vi også finne ut hvem angriperen var. Via Samspade finner vi at 62.16.162.113 = [062016162113.customer.alfanett.no], som egentlig ikke forteller oss så veldig mye, men det gir oss hvertfall en host og ikke bare en ip-adresse. I Sygate brannmuren ligger det innebygd en BackTrace-funksjon man kan bruke på innkommende angrep. Det denne gjør er å ta en mer avansert whois på ip-adressen enn det Samspade gjør, slik at vi får mer informasjon om hvem som eier ip-adressen etc. I tillegg gjør den også en traceroute på pakkene, og finner veien pakkene må gå for å komme fra avsender til mottaker. Dette er en fin funksjon å ha i brannmuren, og kan være nyttig for å identifisere angriperen.



Back Trace-funksjonen innebygd i brannmuren.

Vi ble også fortalt i oppgaven at vi skulle teste angrep som for eksempel installasjon av trojanske hester, angrep som er lagt inn i innkommende datastrøm (ofte via html-kode), eller angrep som skjules i web-applikasjonen (HTTP), som for eksempel "buffer overflow"-angrep. Dette er ikke angrep som en firewall stanser. Sikkerhetspakker som Norton Internet Security eller PC-cillin Internet Security som vi ble anbefalt å prøve i oppgaveteksten er komplette sikkerhetspakker som inneholder både Firewall, Anti-virus programvare, Spam-filter, Spyware scanning og ofte flere ting også. Angrepene som er nevnt stoppes ikke av firewall, men av "resten av sikkerhetspakken". Vi testet disse angrepene på en maskin, og filene ble automatisk stoppet, og slettet fra maskinen uten at vi fikk noe beskjed om dette. I loggen derimot så vi at anti-virus delen av sikkerhetspakken hadde slettet filene. Konklusjonen blir derfor at slike angrep kommer igjennom brannmuren, men stoppes av anti-virus programvare. Det er derfor viktig å ikke stole blindt på brannmuren, men også ha oppdatert anti-virus program på pc'en. Anbefalte programmer er Norton, Norman, PC-cillin med flere. Vi kommer mer inn på dette i del 3 rett nedenom her.



Logg fra Norton Antivirus 2005 etter at vi hadde herjet litt med datamaskinen.

Del 3: Systemsikkerhet uten brannmur?

- Brannmur blir litt som med døren til huset ditt. Det er selvfølgelig mindre risiko for at noen kommer gjennom den hvis du låser den, men samtidig er du nødt til å åpne den hvis du selv skal ut en tur.
- Ikke nødvendigvis. Huset har jo vinduer (Windows).

Gir brannmur en falsk følelse av sikkerhet? Det korte og mest opplagte svaret er ja, men vi skal nå i teksten nedenfor prøve å begrunne et svar på denne problemstillingen. Vi antar at det er privatpersoner og små bedrifter vi diskuterer nå. Ingen er vel i tvil om at en datamaskin hos Microsoft må ha en eller annen form for brannmurbeskyttelse, da disse maskinene er hyppig utsatt for angrep.

En brannmur er laget for å filtrere datatrafikk inn og ut fra maskinen. I realiteten så er standardoppsettet ofte at all trafikk ut av maskinen er tillatt, mens trafikk inn på maskinen er strengt avgrenset. Har du noen gang opplevd at du får pinget maskinen, koblet opp, men at selve overføringen stopper opp? Dette er det brannmuren som er skyld i på grunn av dårlig filtrering av trafikken. Det har seg slik at en god del brannmurer ikke tillater å viderebefordre ICMP-pakker. Selv om vi kan skjønne grunnen til dette ved å se på ICMP-DoS angrepene over her i del 2, så har ICMP flere viktige funksjoner. ICMP annonserer feil i nettet relatert til TCP/UDP og gir også beskjed om Host/Network unreachable. I tillegg annonserer også ICMP metningssituasjoner i nettet, såkalt "Source Quence" fra ruterene. Hvis slike meldinger blir filtrert bort av brannmuren så kan vi skjønne hvorfor vi blir sittende med en blank side i nettleseren vår.

ICMP er i tillegg nytting ved feilsøking. Man kan sende et ICMP Echo Request/Reply for å finne ut om en ekstern maskin kommuniserer, populært kalt "ping". Man kan også annonsere time-out situasjoner som Time to Live (traceroute) og avdekke rutingløkker. Viktige data og feilrapporter kommer ikke frem eller blir filtrert bort av brannmuren og vi får uforklarlige feilsituasjoner. Ved slike feil bortfaller også muligheten til feilsøking. Du får kontakt, mens eier av ønsker ressurs sier at alt fungerer helt fint. Hvor skal man da lete for å finne feilen?

En stor sikkerhetsrisiko på en datamaskin kommer fra programvare brukeren selv har installert i god tro. Spesielt gjelder dette gratis programvare som i tillegg til den annonserte funksjonaliteten ofte kommer med "trojaner". En personlig brannmur kan hjelpe til med å beskytte mot slike, på den måten at den blokkerer trojaneren når den forsøker å lage oppkoblinger. Hvorfor ikke bare unngå slik programvare? Man skal ikke åpne ting man ikke vet hva er, uansett!

Mange bruker fildelingstjenester (p2p) for å finne musikk og filmer på nettet. I slike programmer kringkaster man sin egen IP-adresse, og da kan man risikere å bli portscannet av crackere som skjønner at på denne ip-adressen så står det en kraftig pc med rask tilkobling til internett. En slik maskin er et interessant mål fordi den kan brukes til blant annet å angripe andre maskiner.

Vi kom opp med et par spørsmål:

- Hvis det ikke skjer noe som helst med datamaskinen, hvordan er det da et angrep?
- Hvis vi heller ikke merker at vi blir angrepet, hva er da problemet?
- Er å kaste snøball et angrep?

Svaret på at vi ikke merker noe kan være at vi har for dårlig beskyttelse på maskinen. Det er godt mulig at noen ondsinnede greier å skaffe seg personlig informasjon fra vår maskin uten at vi selv

oppdager det. Hvis vi omdefinerer "angrep" til å være "ondsinnet handling" får vi også svaret på det første spørsmålet. Nemlig at såkalte angrep ikke alltid er utført for at vi SKAL oppdage noe, men heller at folk kan snike seg usett inn på maskiner og stjele ting og tang.

En brannmur har tilgangs-kontroll (sperre porter, blokkere IP'er osv), men hvis man f.eks. har et virus i en zippet fil og sender filen via epost kommer den automatisk gjennom fordi brannmuren selvfølgelig ikke sperrer porter som brukes til epost.

En løsning for selskaper der mange ansatte sjekker epost i arbeidstiden er å kjøre antivirus scanner på epost serverne og ikke i klientene, men det oppstår lett problemer med dette. Mange ansatte i et selskap har forskjellige leverandører av epost. Noen har webmail løsninger og går derfor ikke gjennom sitt selskaps epost server, men direkte via internett. Da vil det være liten vits i slike løsninger.

Vi ser i UNINETT's presentasjon er at de nevner hastighets begrensninger, men dette gjelder da kun for nettleverandørene og andre som har linjer med stor kapasitet (Vi tolker det slik siden det nevnes 10Gbps++) og ikke sluttbrukerne. Når vi kjører linjehastighetstester for ADSL får vi jo som oftest den farten vi skal ha likevel med de små hastighetene som tilbys. Alle sluttbrukere bør uansett ha en brannmur uavhengig av hva UNINETT gjør/mener.

Konklusjonen må bli at de fleste risikoer reduseres kraftig ved bruk av en brannmur. Selv om man aldri kan bli 100% beskyttet mot angrep, så vil en personlig brannmur ha den samme avskrekkende effekten som en innbruddsalarm. Den vil ikke stoppe de som er fast bestemt på å bryte seg inn, men de fleste vil gå videre til enklere ofre.

Del 4: Risikovurdering: Video-overvåkning av PC-lab ved HSM

Etter en kikk i Datatilsynets veileder(se kildehenvisning) for bruk av kameraovervåkning kom vi fram til at HSM har et saklig behov for å bruke slik overvåkning fordi det i dette tilfellet gjelder sikring mot innbrudd, tyveri og bevis til straffesaker.

Kartlegging av opplysninger vi mener er berørt av overvåkningstjenesten.

Informasjonstype Formål	Sensitive opplysninger	Omfang	Sikkerhetsbehov
Ansikter kan bli gjenkjent på videoen Identifisering av personer	Ja	Mange. Kommer litt an på hvor mange personer som får plass på bildet/videoen	Integritet,Konfidensialitet
Vaner/uvaner til folk Ingen spesielle	Nei, men det kan være ubehagelig for vedkommende	Varierende, kameraet observerer mange personer i løpet av en skoledag.	Integritet

Skala for letthet(med stigende vanskelighetsgrad)
(Vi bruker her skala for letthet siden vi ikke har noen statistikk å basere vurderingen på.)

1 Sikkerhetstiltak er ikke etablert. "Alle" kan få tilgang til videoene uten hjelpemidler.
2 Sikkerhetstiltak har liten virkning. Personer trenger små ressurser for å omgå tiltakene.
3 Sikkerhetstiltak er etablert i forhold til sikkerhetsbehovet og fungerer etter hensikten. Eget personell kan omgå disse tiltakene med små ressurser
4 Bare personell hos IT-senteret med god kjennskap til tiltakene kan få tilgang til videoene eller bryte seg inn på websiden. Det kreves mye ressurser.

Skala for konsekvens(med økende konsekvens):

- Liten (1), hendelsen kan medføre økonomisk tap eller anseelse.
- Moderat (2), hendelsen kan medføre betydelig økonomisk tap, men som kan gjenopprettes.
- Stor (3), hendelsen kan medføre tap av liv eller helse, eller stort økonomisk tap eller tap av anseelse og integritet.
- Katastrofal (4), hendelsen kan føre til tap av liv og/eller enormt økonomisk tap.

Her er en tabell over akseptabel risiko(letthet * konsekvens). De grå rutene representerer uakseptabel risiko. Disse tallene bruker vi senere i risikovurderingen for å se om hendelser havner utenfor eller innenfor akseptabelt risikonivå.

L\K	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

Hendelse/trussel Videobåndet blir stjålet (tilsiktet, permanent utlevering).			Nr. 1
KONSEKVENSVURDERING Beskrivelse: Høyskolen får ikke tatt vedkommende som stjeler PC'er og/eller annet datautstyr.			
Katastrofal	Stor	**(Moderat)**	Liten
SANNSYNLIGHETSVURDERING Årsaker: Området hvor videobåndene befinner seg kan være for dårlig sikret. Dører er kanskje ikke låst og rommet er kanskje ikke overvåket. Personer kommer seg derfor inn hvor videomaskinen med videobåndene er og tar videobåndene.			
1	2	**(3)**	4
RISIKOVURDERING Risiko for denne hendelsen er : 3 * 2 = 6. Dette faller innenfor grensen for akseptabel risiko.			
RISIKOREDUSERENDE TILTAK Man bør sikre rommet hvor videobåndene befinner seg. Helst ved å låse døren og la bare ansvarlig personell få tilgang til nøkkelen. Man bør også sjekke rommet med jevne mellomrom likevel for å sjekke om noe likevel har foregått i rommet.			

Hendelse/trussel Noen greier å komme seg inn på nettstedet på en uhederlig måte (tilsiktet, permanent utlevering).	Nr. 2
KONSEKVENSVURDERING Beskrivelse: Om en person skulle greie å komme seg inn på denne siden, kan personen også	

distribuere videoen videre på nettet.			
Katastrofal	**(Stor)**	Moderat	Liten
SANNSYNLIGHETSVURDERING Årsaker: Brukernavn og/eller passord kan være for dårlig gjennomtenkt slik at det blir en enkel sak å gjette riktig. Det er kanskje ikke brukt tilstrekkelig antall tegn (eller forskjellige typer tegn: tall,bokstaver,@$£€ osv)i passordet, så hvis en kjører en løkke eller lignende for å finne passordet blir det funnet relativt raskt. Kanskje sender siden passord til serveren i klartekst og noen greier å fange pakkene og få passordet derfra. Kanskje brukeren har lagret passordet på sin lokale maskin og noen stjeler den maskinen, eventuelt bare bryter seg inn på den.			
1	**(2)**	3	4
RISIKOVURDERING Risiko for denne hendelsen er : 3 * 2 = 6. Dette faller innenfor grensen for akseptabel risiko.			
RISIKOREDUSERENDE TILTAK Passordet bør velges med helst 8 tilfeldige tegn eller mer for å gjøre det mest mulig vanskelig å knekke. Passordet bør også memoriseres av vedkommende som har det slik at ingen finner notater der passordet er nedskrevet. Man kan også sende passordet kryptert så det blir tilnærmet umulig å sniffe dette.			

Hendelse/trussel Videobånd med tjuven(e)på blir overskrevet ved et uhell (utilsiktet, permanent endring, sporbar).			Nr. 3
KONSEKVENSVURDERING Beskrivelse: En ansatt på IT-senteret er uheldig og overskriver et bånd som inneholder gode bilder av tyven(e). Heldigvis er det 2 kameraer per rom og derfor også 2 bånd. Det andre båndet har muligens ikke fullt så gode bilder av tyven.			
Katastrofal	Stor	**(Moderat)**	Liten
SANNSYNLIGHETSVURDERING Årsaker: Den ansatte kan være trøtt etter en lang dag og trykker på "record" i stedet for "eject". Eventuelt kan båndet også bli ødelagt pga at båndet var i en dårlig forfatning før det ble tatt i bruk. Personalet har kanskje for mye å gjøre og blir stresset og gjør så noe galt.			
1	2	**(3)**	4
RISIKOVURDERING Risiko for denne hendelsen er : 2 * 3 = 6. Dette faller innenfor grensen for akseptabel risiko.			
RISIKOREDUSERENDE TILTAK Det er lurt å ha 2 kameraer slik at man unngår å tape alt av beviser hvis noe slikt skulle skje. Man bør også se på rutiner om hvor lenge en ansatt bør jobbe. Man bør også ha rutiner for håndtering av videobånd slik at denne situasjonen ikke oppstår.			

Hendelse/trussel	Nr. 4

Kameraet blir stjålet (tilsiktet, utilgjengelighet - tidsavbrudd(til et nytt kommer på plass)).	

KONSEKVENSVURDERING
Beskrivelse: En frekk tyv kommer inn og ser kameraet i taket. Tyven vurderer saken og tenker at kameraet kan være verdt litt penger også. Kameraet blir tatt og tyven forsvinner.

Katastrofal	Stor	Moderat	**(Liten)**

SANNSYNLIGHETSVURDERING
Årsaker: Kanskje vedkommende vet om kameraet og vet at det koster en del penger. Tyven er kanskje dum og tror han/hun ikke blir tatt hvis kameraet forsvinner.

1	2	**(3)**	4

RISIKOVURDERING
Risiko for denne hendelsen er : 1 * 3 = 3. Dette faller innenfor grensen for akseptabel risiko.

RISIKOREDUSERENDE TILTAK
Det vil være vanskelig å sikre et slikt tyveri. Kanskje kameraet kan monteres et stykke inn i veggen slik at bare linsen stikker utenfor. Det vil da være umulig å røske ut kameraet med bare hendene.

Hendelse/trussel Noen greier å lure et kamera på et eller annet vis (tilsiktet, har lite med personopplysninger å gjøre, men er en meget uønsket trussel).	Nr. 5

KONSEKVENSVURDERING
Beskrivelse: En person kommer i mørket og hiver en jakke over kameraet. Går ut i fra at det andre kameraet ikke filmer infrarødt og at kameraet er posisjonert slik at det er mulig å kaste jakke over. Så blir en PC stjålet.

Katastrofal	Stor	**(Moderat)**	Liten

SANNSYNLIGHETSVURDERING
Årsaker: En tyv har vært inne i laben tidligere og sett at kameraet stikker ut fra veggen. Tyven planlegger da å snike seg inn og kaste en jakke eller lignende over kameraet. Kameraet blir da uskadeliggjort. Det er godt mulig dette skjer siden det er så mange som ferdes i disse rommene. Men på den annen side, hvis tyven har sett at det er 2 kameraer i rommet, er det ikke sikkert tyven tror dette vil lykkes.

1	**(2)**	3	4

RISIKOVURDERING
Risiko for denne hendelsen er : 2 * 2 = 4. Dette faller innenfor grensen for akseptabel risiko.

RISIKOREDUSERENDE TILTAK
Det vil være mer avskrekkende å ha 2 kameraer i rommet enn bare 1. Det vil være vanskelig å uskadeliggjøre 2 kameraer samtidig. Hvis en person gjør noe med det ene kameraet vil det andre likevel fange opp det som skjer. Ihvertfall inntil noe skjer med det andre også.

Hendelse/trussel En person som er ansvarlig for overvåkningen kvitter seg med videobåndene på grunn av at tjuven var en han/hun kjenner (tilsiktet, endring- sporbar).	Nr. 6

KONSEKVENSVURDERING Beskrivelse: En ansvarlig person kommer over videoen og ser at det er noen vedkommende kjenner og erstatter videobåndene for det gjeldende tidsrommet med en tom video.			
Katastrofal	Stor	**(Moderat)**	Liten
SANNSYNLIGHETSVURDERING Årsaker: Den ansvarlige personen vil ikke at den kjente personen skal bli tatt for tjuveri, særlig siden de kjenner hverandre.			
1	2	**(3)**	4
RISIKOVURDERING Risiko for denne hendelsen er : 2 * 3 = 6. Dette faller innenfor grensen for akseptabel risiko.			
RISIKOREDUSERENDE TILTAK Hvis rommet hvor videoen er oppbevart også var overvåket og båndet for denne overvåkningen var plassert i et sikkert rom vil det være mindre sjanse for at noe slikt vil kunne skje. Dette ville også raskt bli litt for mye overvåkning. Det beste ville være om de ansvarlige var til å stole på med tanke på slikt. Det blir her etiske tanker i bildet. Men det som kunne avskrekke dette må være å i verste fall miste jobben sin. Og det ville jo være dumt om båndet skulle dukke opp i ettertid eller noen oppdager at det opprinnelige båndet er borte, f.eks. at det opprinnelige båndet er merket på en slik måte at merkingen ikke kan fjernes.			

Hendelse/trussel Man kan bli urettmessig beskyldt for et tyveri som kan være begått av andre (utilsiktet, Utlevering-kan tilbakeføres).			Nr. 7
KONSEKVENSVURDERING Beskrivelse: Tyver har vært på laben og stjålet datautstyr. En uskyldig person kommer inn senere og ser at noe har blitt stjålet. Personen blir nervøs og stikker av. Dessverre kom det med på et kamera at denne personen sprang ut av rommet. De(n) ansvarlige på IT-senteret ser bare denne personen på videoen og beskylder derfor denne personen for tyveri.			
Katastrofal	Stor	**(Moderat)**	Liten
SANNSYNLIGHETSVURDERING Årsaker: IT-ansvarlige er kanskje uoppmerksom på hva som faktisk skjer på videoen og oppfatter hendelsen som annerledes enn den var i virkeligheten. Det kan skyldes dårlig bildekvalitet eller at det er mørkt i rommet.			
1	2	3	**(4)**
RISIKOVURDERING Risiko for denne hendelsen er : 2 * 4 = 8. Dette faller utenfor grensen for akseptabel risiko.			
RISIKOREDUSERENDE TILTAK Hvis dette var om natten ville det være vanskelig for kameraene å få med det som faktisk skjedde. Det ville da ha hjulpet om kameraene kunne bytte modus til infrarød når det blir mørkt i rommet.			

OPPSUMMERING AV VURDERINGEN

I denne risikovurderingen ble det avdekket én hendelse som hadde høyere risiko enn det akseptable risikonivået. For å få denne hendelsen inn i det akseptable området trengs det sikkerhetstiltak, enten

for å redusere konsekvensene av eller sannsynligheten for uønskede hendelser. Vi har inkludert noen risikoreduserende tiltak i vurderingen vår. Vurderingsgrunnlaget for valg av tiltak kommer an på om man vil at hendelsen:

- unngås
- avskrekkes
- hindres
- isoleres
- oppdages

eller om tiltaket har som formål å gjenopprette tilstanden før hendelsen inntraff.
Det er også viktig å ta hensyn til kriterier for kvalitet. Det er aktuelt å sette krav til- og vurdere funksjon, styrke og konsistens, og om tiltakene er fullstendige og dekkende. Ut over dette må det vurderes om tiltakene er kostoptimale og praktiske, og om det er mulig å verifisere at de fir tilsiktet effekt.

En ettertanke: Et annet område som kunne vært overvåket er området hvor man setter fra seg sykler. Dette er et område hvor mange ferdes og stjeling av sykler lett kan forekomme. Ikke alle låser syklene og det er ikke mulig å vite om folk faktisk eier syklene når man ser folk sette seg på dem og sykle av gårde.

Kilder:

Risikovurdering av informasjonssystem (Datatilsynet).
Tilgjengelig fra URL:
http://www.datatilsynet.no/upload/Dokumenter/infosik/veiledere/Risikovurdering_TV-506_02.pdf [sist sett: 25.04.05]

Kameraovervåking – en veileder (Datatilsynet)
Tilgjengelig fra URL:
http://www.datatilsynet.no/upload/Dokumenter/publikasjoner/brosjyrer/kamoverv.pdf [sist sett: 26.04.05]

Utdelt kopi av eksempler på risikovurdering(utdrag fra M&R Fylkeskommune og Fiber i Hardanger) [utdelt 26.04.05]

4. IN530: RUTEPLANLEGGING OG KORTESTE VEI MELLOM ADDRESSER AV EIVIND ANDERS BERG, ANDERS BJARNE SKJELTEN GJENDEM OG LENE THERESE ØSTBY

Kontakt: Eivind Anders Berg (eivind.a.berg@himolde.no), Anders Bjarne Skjelten Gjendem (anders@gjendem.net) og Lene Therese Østby (lene.t.ostby@himolde.no)

Bakgrunn

Programmet finner korteste vei mellom et antall adresser, og er primært tenkt brukt i et beslutningsstøttesystem som benyttes til ruteplanlegging. Prosjektet vil også kunne bli brukt som basis for mastergradsoppgaver om ekvilibriums-modeller. Eksisterende løsninger har ikke vært effektive og fleksible nok for ønsket bruk, og det var derfor ønskelig å utvikle et program som vil gjøre det mulig å på sikt frigjøre seg fra kommersielle verktøy, samt gjøre det enklere å implementere egne premisser.

Oppdragsgiver og faglig veileder:

Arne Løkketangen (arne.lokketangen@himolde.no)
Professor i Informatikk

Interessenter:

Odd Larsen (odd.larsen@himolde.no)
Professor i Transportøkonomi

Johan Oppen (johan.oppen@himolde.no)
Stipendiat

Oppgavebeskrivelse

Bakgrunnen for oppgaven er Dr. Stud. Johan Oppens arbeider innen ruteplanlegging for Norsk Fagråd for Kjøtt. I dette prosjektet skal han utvikle prototypverktøy for å kunne planlegge inntransport av dyr til to utvalgte slakterier (Gilde på Rudshøgda og Fatland på Jæren). I denne sammenheng trenges reisetid mellom alle aktuelle produsenter i en ukes perspektiv. Typiske antall kunder vil være "noen hundre" (mindre enn tusen). Reisetidsberegningen skal typisk integreres i et beslutningsstøttesystem, slik at reisetidene blir beregnet ut ifra "ukens leverandører". Responstiden er således av en viss viktighet. (Firmaet GeoData har p.t. store problemer med å skaffe tilveie enkel avstandsinformasjon for 400 kunder. De estimerer at 800 kunder vil ta 100 timer). Det vil sikkert også være lurt å utvikle en enklere applikasjon først, som finner korteste reisetid mellom to adresser, lignende det for eksempel Gule Sider tilbyr.

Nødvendig kartinformasjon ligger i ELVEG databasen i Sosi-format. Denne er altfor finmasket for vårt formål, slik at noe preprosessering antageligvis bør foretas. Til visningsformål er tenkt brukt verktøyet MapInfo. Adresseinfo ligger også i ELVEG.

(Odd Larsen kjenner en Tom Hamre, ex TØI, som har jobbet med preprosessering av ELVEG. Det er sikkert nyttig å snakke med ham). Det hadde vært kjekt om man i utregningen av reisetid også kunne ta hensyn til ferjer og deres avgangstider. Ferjerutene ligger i ELVEG, men ikke avgangstidene. Professor i transportøkonomi Odd Larsen er også interessert i at en slik oppgave blir utført, da dette vil kunne danne basis for senere oppgaver som kan utvide dette arbeidet til ekvilibriums-modeller. Man kan da frigjøre seg fra bindingene til de kommersielle verktøyene man bruker i dag, og lettere kunne implementere egne premisser.

Mål og eksisterende løsninger

Av eksisterende løsninger på dette området er det særlig to som er interessante for oss: Geodatas løsning, og applikasjonen Emme som brukes ved institutt for økonomi. Emme er, i følge professor Odd Larsen, svært rask, men til gjengjeld svært lite fleksibel – for hver type informasjon man er interessert i om en rute, må beregningen kjøres på nytt. Geodatas løsning er muligens noe mer fleksibel, men bruker forferdelig lang tid – de estimerer 100 timer på ruter mellom 800 adresser.

Det primære målet for oss har derfor vært å lage noe som er raskere enn Geodata, og fortrinnsvis noe mer fleksibelt enn Emme – noe vi delvis har klart.

Arbeidsmetoder

Utviklingsspråk

Den første prototypen ble kodet i Visual Basic .Net. Dette ble valgt fordi det er lett å lage grafiske brukergrensesnitt i Visual Basic, og det er et språk vi kjenner rimelig godt. Tidlig i fasen ble det også vurdert som språk på resten av prosjektet, men med tanke på hvor lite vi egentlig har brukt språket i forhold til andre alternativ ble det forkastet. Det ble foreslått å bruke C eller C++ med tanke på hastigheten, siden dette var viktig for prosjektet vårt. Ulempen med disse språkene er at vi har lite eller ingen kjennskap til dem fra før, og vi følte at én persons C-kunnskaper fra ett kurs ikke var nok til å velge det. Alternativet ville vært at vi hadde brukt hele semesteret på å lære oss brukbar C/C++ - for å lære et språk godt trenger man normalt et par år. Da stod vi igjen med Visual Basic.Net eller Java. Java ble valgt av den grunn at det er det språket vi har brukt aller mest i løpet av utdanningen, blant annet i relevante fag som IN 295 – Algoritmer og datastrukturer og IN 212 – Objektorientert utvikling.

En stor fordel med Java (samt Visual Basic .Net) er at koden kjører i et såkalt ”managed” miljø. Det betyr i praksis at man for eksempel ikke får problemer med pekere som forsøker å lese minne på ulovlige steder og fører til at maskinen og/eller programmer henger seg, i motsetning til hva som er mulig med C og tildels C++. Det kan være svært frustrerende og vanskelig å finne ut av slike feil. Andre fordeler er at språket er relativt enkelt, syntaksmessig likner det en del på C/C++. Klassebiblioteket er også svært omfattende og meget godt dokumentert, det gjør det enkelt å kode, og man slipper i mange tilfeller å finne opp hjulet på nytt. Det skulle også vise seg at Java Swing ble viktig i utviklingen. Sånn sett fikk vi også lært mye nytt, siden programmering av brukergrensesnitt i Java ikke er et område det blir fokusert mye på i andre kurs.

Java har hatt et rykte på seg for å være veldig tregt, og dermed et dårlig valg med tanke på den type prosjekt vi har valgt. I praksis er dette blitt ganske bra, kraftigere maskinvare gjør sitt til at dette ikke er merkbart i samme grad som det var før. Den koden som behøver hastighet er heller ikke spesielt

omfattende, og valg av gode datastrukturer har vist seg å ha en atskillig større innvirkning på hastigheten enn valg av språk. Tilgangen på gode utviklingsmiljø har også vært avgjørende for valget. En annen positiv bieffekt av å bruke Java, er portabilitet mellom flere operativsystemer. Vi har i visse situasjoner hatt behov for å få kjørt deler av programmet under Linux, og dette er ikke noe problem med Java.

Verktøy

Vi har valgt å utvikle prosjektet i Eclipse – et open-source, gratis utviklingsmiljø som fritt kan lastes ned fra http://www.eclipse.org/. Så hvorfor bruke Eclipse og ikke Netbeans eller Sun One Studio som allerede er tilgjengelig på skolen?

Selv om Eclipse er mest kjent som en Java IDE (Integrated Development Environment), er det hovedsaklig en plattform. Eclipse støtter bruken av jsp, xml, html og annet som gjør utviklingsprosessen mye mer fleksibel. Det er også et antall utvidelser tilgjengelig, det kan for eksempel nevnes Visual Editor – et verktøy for å utvikle grafiske Java-grensesnitt i Eclipse. I tillegg kan det synes som at Eclipse er fremtiden [36]. Store selskaper står bak plattformen, først og fremst IBM som finansierer det hele.

Det grafiske grensesnittet er ekstremt fleksibelt og har få feil – for eksempel fungerer debugging utmerket i Eclipse, mens det i Sun One Studio i tidligere prosjekter har vært tilnærmet umulig. Man kan enkelt hoppe mellom debug-vinduet med vanlig debugging, trace code og oversikt over variabler, og java-vinduet med hierarkisk oversikt over klassene, selve kodevinduet, metodene og konsollen med mer. Man kan også endre og fjerne vindu man ikke vil ha med, så mulighetene er mange. Et annet viktig poeng er at Eclipse har støtte for flere skjermer, noe de andre verktøyene så vidt vi er kjent med ikke har [36]. Kort sagt har vi tilgang til svært mange ulike løsninger i grensesnittet.

Et annet kjennetegn ved Eclipse er mangelen på en ”build all”-knapp – Eclipse autokompilerer når man lagrer eller kjører programmet. Koden er med andre ord alltid kompilert. Den innebygde støtten for refactoring er også uvurderlig i et litt større prosjekt – se Refactoring i Eclipse. Eclipse kjører også selv om koden inneholder syntaks-feil. Når en metode som inneholder feil blir kalt, hopper programmet til debuggeren og lar oss rette opp feilen og fortsette å kjøre programmet som om ingenting har skjedd. Automatisk kodefullføring og innebygd visning av Javadoc når dette er tilgjengelig er en selvfølge [36].

Den største enkeltstående faktoren for vårt valg av utviklingsmiljø var at Eclipse støtter CVS. Dette gjør det svært enkelt for flere å jobbe på samme prosjektet – se CVS.

Vi valgte å bruke testversjonene av Eclipse gjennom hele semesteret, på dette tidspunktet er enda ikke Eclipse 3.1 sluppet. Hovedgrunnen til dette var støtte for Java 1.5, som vi i starten benyttet flittig inntil vi fikk problemer med å kjøre Profiler (ytelsesanalyser) på koden. For å få dette til å fungere ordentlig, ble vi nødt til å skrive om en del steder og gå over til Java 1.4 syntaks, i stedet for den nye syntaksen for f.eks. Collection-klassene.

I deler av prosjektet benyttet vi oss av AppPerfect sin DevSuite [37]. Denne integreres i Eclipse, og gir oss muligheter til å se hvor mange ganger en funksjon kalles, prosent av tiden den bruker og mye mer. Vi har fått erfare at man som regel forsøker å optimere koden på de stedene det ikke trengs. Etter å ha brukt DevSuite og etter beste evne forsøkt å gjøre endringer i koden, oppnådde vi for eksempel mangedoblet hastighet på innlesning og parsing av geometri-dataene. Der har vi nok

fremdeles en god del å gå på, men vi synes fordelen med det meget fleksible og utvidbare systemet vi har for innlesning oppveier tapet av noe større hastighet.

Vi nevnte tidligere at Eclipse også har en plugin for å lage grafiske grensesnitt, nemlig Visual Editor [38]. Denne ble brukt i startfasen av grensesnittutviklingen, og er et godt utgangspunkt – spesielt for mindre vinduer uten for mange spesialkomponenter. For vårt bruk ble den etter hvert overflødig, men å se på hvordan koden den genererte for vinduene ble, var lærerikt og ble benyttet videre. Dette førte til at vi fikk et mye mer robust grensesnitt enn vi ville hatt ellers, og har nok spart oss for betydelige problemer. Grensesnitt har det vært veldig lite fokusert på i Java-fagene, og ville uten Visual Editor til dels måtte læres fra scratch.

Versjonskontroll

Vi bestemte oss for å bruke versjonskontrollsystemer for koden. Siden prosjektgruppen ikke har felles fag og forelesninger dette semesteret ville det vært mye ekstraarbeid med å koordinere kodingen, sørge for at man til enhver tid koder på nyeste versjon og at man unngår dobbeltarbeid. Når alle utviklerne oppdaterer sine lokale kopier fra versjonskontrollserveren før de starter, og etter hver velfungerende endring eller feilfiks sørger for å oppdatere denne mot serveren, blir denne koordineringen i stor grad unødvendig. Ved å ta i bruk TODO og FIXME kommentartypene i Eclipse, får man også en egen oversikt over hva som gjenstår av arbeid, hvor det er kjente feil som noen må ta en kikk på osv.

Valget av versjonskontrollsystem stod mellom SubVersion [39]og CVS (Concurrent Versions System) [40]. CVS ble valgt hovedsaklig fordi den fremdeles er mest utbredt, og danner mer eller mindre grunnlaget for SubVersion. SubVersion er av de fleste sett på som arvtakeren til CVS og utbedrer en del mangler CVS har, men vi ønsket å bruke den klassiske varianten først – SubVersion kommer nok til sin rett i senere prosjekter. CVS-støtten i Eclipse er meget bra: det er innebygde verktøy for å sammenligne og slå sammen to forskjellige versjoner av samme fil, og de fleste andre funksjonene et utviklermiljø ønsker seg for å kunne jobbe effektivt sammen på samme prosjekt. En annen stor fordel er at man kan eksperimentere med koden for å se om en endring er til det bedre – hvis den ikke er det er det fort gjort å overskrive med den som er den nyeste på serveren, eller alternativt lagre din versjon som en egen ”branch” som kan hentes frem igjen senere, og som de andre utviklerne ikke uten videre får fra serveren.

Når versjonskontrollsystemet var på plass, var det også interessant å kunne følge fremgangen i utviklingen visuelt. Det finnes flere alternative programmer som genererer slik statistikk ved å gå igjennom loggene på serveren. Den vi endte opp med å bruke var StatCVS [41]. Denne plotter grafer over kodelinjer, kan vise informasjon om hvilken utvikler som har jobbet mest med hvilke deler, nøyaktig hva som er gjort osv. I aktivitetsloggen er det et eksempel på en slik graf.

StatCVS kan igjen kobles mot et par andre systemer. For å få tilgang via internett på all koden i prosjektet brukte vi også et program som heter CVSweb [42]. Det fine med dette systemet er at det er lett å sammenligne flere utgaver av samme fil i prosjektet, og få dette for eksempel fargekodet. Dette er også støttet direkte i Eclipse, men er fremdeles interessant i andre sammenhenger, for eksempel for utenforstående brukere/utviklere som har kommentarer eller ønsker å sjekke noe uten å måtte få tilgang til å hente ut hele prosjektet fra CVS-serveren.

I programmet ble det også brukt en database til adressedata, se for øvrig eget avsnitt om databasen, samt installasjonsveiledningen. Vi brukte hovedsakelig PostgreSQL [43] som databasesystem, men andre databaser med JDBC-støtte skal være mulig å bruke. Valget falt på PostgreSQL fordi det har

mange avanserte funksjoner, for eksempel for analyse av spørringene, se egen seksjon om optimering av databasen. PostgreSQL er også et åpent kildekode prosjekt, slik som de fleste andre programmene vi har brukt i utviklingen er. MySQL er et greit alternativ, men mangler en del avanserte funksjoner. Disse bruker vi ikke nødvendigvis nå, men de kan være aktuelle med tanke på fremtidig videreutvikling, og PostgreSQL var derfor et naturlig valg.

Refactoring

Refactoring er prosessen med å forandre software på en måte som ikke forandrer kodens oppførsel utad, men forbedrer den interne strukturen [17]. Dette er en metode som er spesielt nyttig i prosjekter som i liten grad planlegges fullstendig på forhånd, men "blir til" underveis – svært mye brukt i for eksempel Extreme Programming. Ved å bruke refactoring-metoder får man ryddet opp i koden og forbedret strukturen med relativt enkle grep, og ettersom Eclipse har støtte for refactoring ble disse metodene hyppig benyttet underveis i prosjektet.

Når man har jobbet på den måten vi har, kommer behovet for reorganisering av deler av programmet rett som det er. Da oppstår det ofte et behov for å endre navn på klasser og pakker. Spesielt nyttig var funksjonene for å endre signaturen til en funksjon, og automatisk få oppdatert den de stedene funksjonen brukes. Det skjer også relativt ofte at det oppstår et behov for å flytte en klasse eller pakke til et mer naturlig sted. En slik jobb ville fort tatt en halv time med arbeid i mange tilfeller hvis jobben skulle vært gjort for hånd, og er i tillegg en særdeles kjedelig jobb. Den innebygde støtten for å få innkapslet variabler i klassene ved å gjøre de private, og etter behov legge til get/set-metoder har også vært nyttig. Ingen av disse eksemplene er noen komplisert affære å ordne for hånd, men når prosjektet blir stort nok, er refactoring et nyttig verktøy for å spare tid på små operasjoner som skal gjøres mange ganger, og som dermed kaster vekk tid som kunne vært brukt på noe mer produktivt – og det er noe av poenget med refactoring.

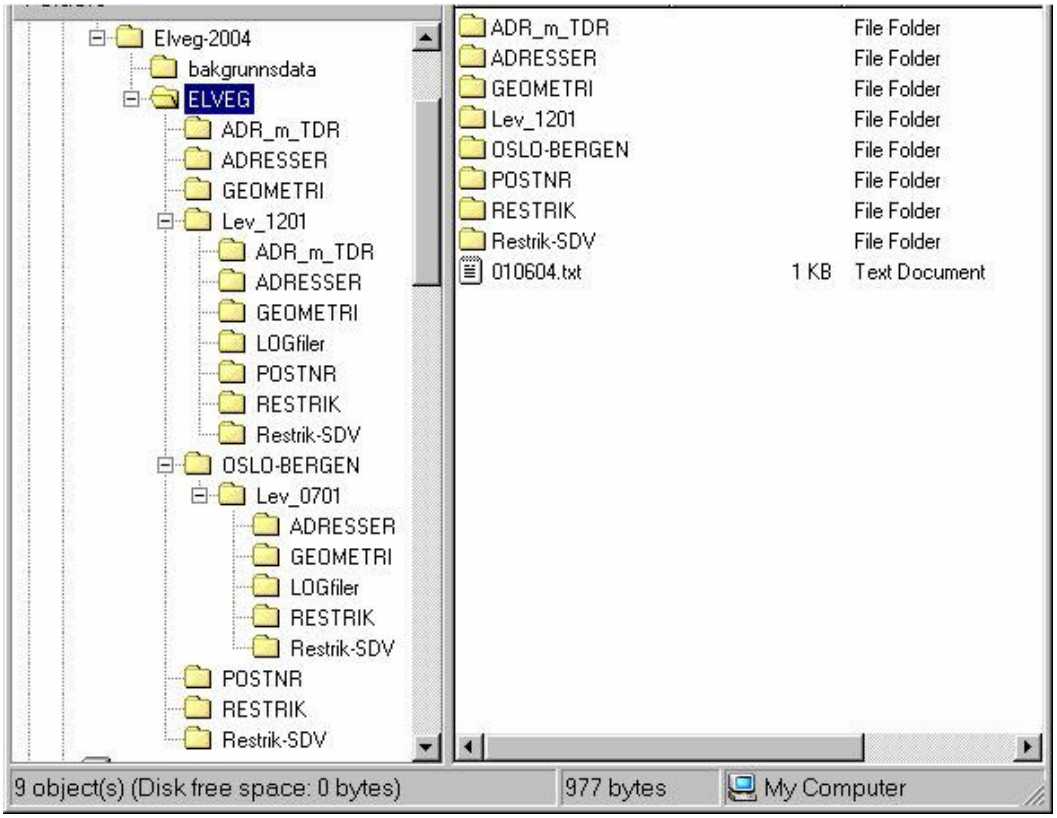
Problemer

Sosi-formatet

ElVeg-databasen er lagret i Sosi-formatet, og dette har til tider skapt en del problemer – formatet var til dels svært dårlig forklart, og har krevd stor grad av selvstudium og oppfinnsomhet.

Selve ElVeg-databasen ble levert på en cd med 609 mb med komprimerte tekstfiler som inneholder informasjon om alle 434 kommuner i Norge. Det fulgte ikke med noen forklaring eller brukerveiledning til databasen, så vi har vært nødt til å finne ut av ting ved hjelp av beskrivelser av Sosi-formatet på internett [21], [22], [23], [24]. I tillegg er selve CD'en svært rotete, og inneholder mye redundant data – det virker som om nye versjoner er lagt til uten at noen har tatt seg bryet med å fjerne de gamle versjonene. For eksempel inneholder mappen Lev_1201 all informasjon om 199 av kommunene – men disse dataene er nyere, og skal erstatte andre data. Mappene Adr_m_Tdr og Adresser inneholder grovt sett det samme – Adr_m_Tdr inneholder data for Bergen, Oslo og Trondheim, Adresser inneholder resten av landet. I tillegg finnes en mappe Oslo-Bergen med adresseinformasjon om disse to byene, og denne mappen inneholder også egne mapper for Geometri, Restrik og Restrik-SDV.

Screenshot fra mappestruktur i ElVeg-databasen



Standarden til Sosi-formatet [25] har vært til dels dårlig forklart – enkelte "forklaringer" er svært forvirrende og tyder på at de som normalt jobber med disse dataene enten har grunnleggende kunnskaper om Sosi fra før, eller tilgang til folk med kompetanse på området. Et eksempel er tvangspunkt, som beskrives på følgende måte i standarden: "Tvangspunkt kan legges inn i et vegkryss med kjent meterverdi (vegreferanse). Det innebærer at den inngitte meterverdi fryses til dette punktet, og at de øvrige meterverdier beregnes ved å interpolere over strekningen mellom tvangspunkter eller hovedparsell-deler med kjent vegident". Vi har ennå ikke funnet ut hva dette innebærer. Et annet eksempel er filen 010604.txt, som forteller at "i forbindelse med harmoniseringsarbeidet hos SVV/SK – medfører at et antall svingekonnekteringslenker ikke har fått tildelt fart og akseltrykk i txt-tabellene. Årsak er at disse ikke har attributten ...VPA, meter til og fra. Ref.til mail fra Kjetil Rønningen SVV 12.11.02." Noen videre informasjon om hva for eksempel svingekonnekteringslenker er, finnes ikke. Sosi-formatet har med andre ord ikke vært det enkleste å forholde seg til.

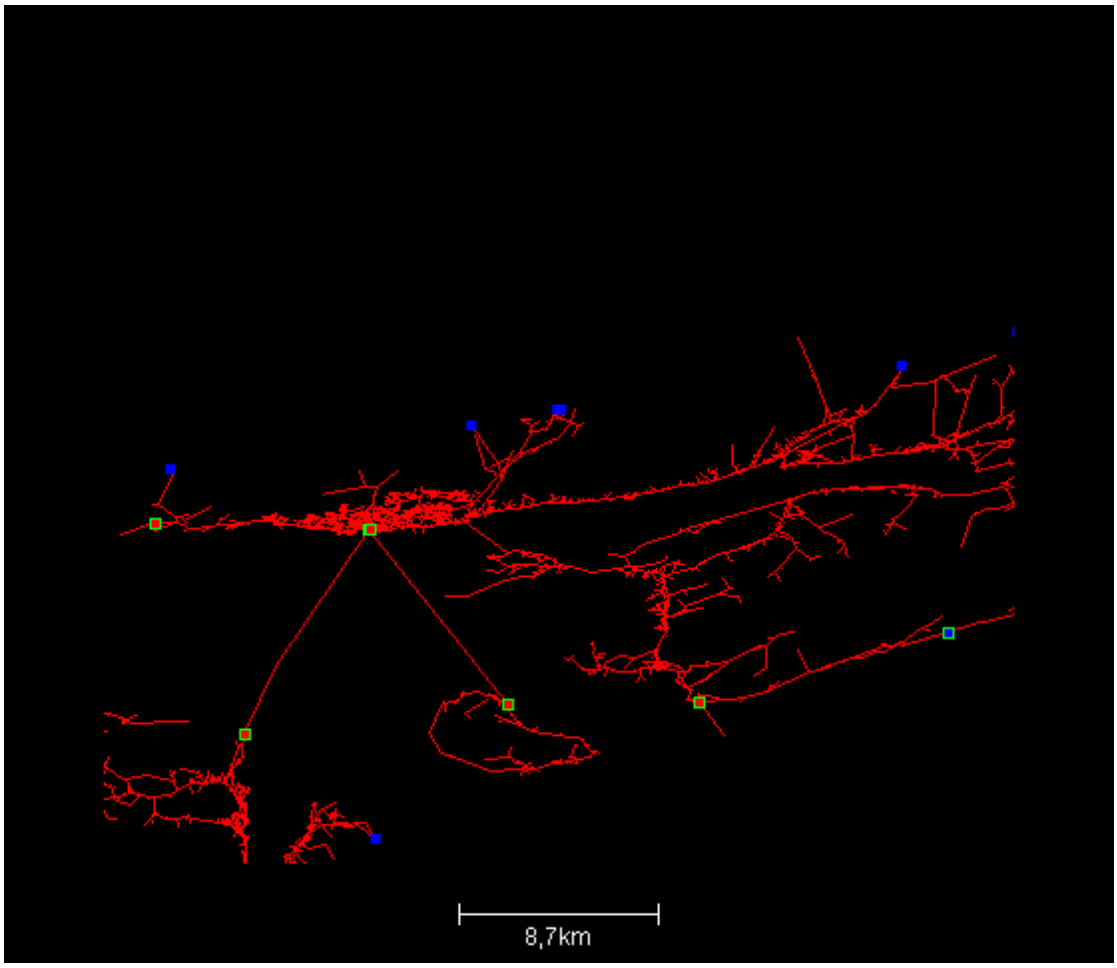
Vi har gjort noen antagelser der det manglet fullstendige data:

- i restriksjons-katalogen finnes det verdier som heter So/Vi/Tel. Vi antar at dette står for sommer, vinter og tele.
- enkelte kommuner (f.eks Sandnes) har flere veger uten kjøreretning. Vi tolker dette som at begge kjøreretninger er tillatt.
- td-nummer er et løpenummer og identifikatoren på en veglenke. Det var lenge uklart om disse numrene faktisk var unike eller ikke, siden vi fant motsigende informasjon om dette. Det ser allikevel ut til at disse i våre data er unike for hver kommune, og vi har brukt td-nummerne som identifikator på veglenkene.

- Restrik og Restrik-SDV inneholder den samme informasjonen om restriksjoner som fart og akselast på vegene – vi har derfor bare brukt et av formatene.

Veglenkene (kurver) skapte en del problemer. Vi trodde at en veglenke, ettersom det så ut slik i standarden, hadde ett startpunktkoordinat som kom etter en ”NØ”/”NØH” (Nord, Øst, Høyde) og ett tilsvarende sluttpunktkoordinat etter ett siste ”NØ”/”NØH”. Vår antagelse viste seg å ikke være riktig overalt, siden det kan dukke opp en ny ”NØ”/”NØH” innimellom de andre kurvekoordinatene, for å vise at de neste koordinatene har/ikke har høyde angitt. Dette førte til at parsingen av dataene ble ”forskjøvet”, som igjen førte til at når vi tegnet opp Molde kommune fikk vi plutselig flere veier mellom Molde sentrum og Hjelset, vei over Romsdalsfjorden til Otterøya, og direkte vei fra Hjelset til Nordbyen. Det gikk mye tid på debugging av slike feil for å passe på at alt vi leste inn fra cd'en var riktig i forhold til virkeligheten. På grunn av slike problemer valgte vi tidlig å lage et brukergrensesnitt for å finne ut om dataene vi fikk var fornuftige. Uten et visuelt grensesnitt for å kontrollere at innlest data var korrekt, kunne ting sett dramatisk annerledes ut uten at vi var klar over dette. De enorme mengdene data gjør det i stor grad upraktisk å debugge på vanlig måte, selv om Eclipse sin støtte for ”Conditional Breakpoints” gjør jobben noe lettere hvis man vet litt om hvor feilen oppstår.

For å finne korteste vei gjennom flere kommuner, måtte vi knytte sammen kommunene. Det viste seg at ikke alle kommunegrensene faktisk var markert som kommunegrenser – et eksempel er mellom Molde og Nesset. Der er punktet i Molde kommune markert som kommunegrense, mens samme punkt i Nesset ikke er markert som kommunegrense. I tillegg er det flere kommunegrenser som i praksis ligger på fergestrekninger. Disse punktene er av en eller annen grunn ikke markert som kommunegrense i Sosi-formatet.



Skjermbildet over viser kommunene Molde, Vestnes (nede til venstre) og deler av Nesset. Vi ser at det ikke er noen markering i punktet over fjorden mellom Molde og Vestnes der hvor det er en svak knekk, ca. midt på. Punktet i Nesset som ikke var korrekt markert er det blå punktet med grønn firkant rundt ute til høyre. Dette bildet er tatt etter at vi fikset problemet ved å gjøre en grundigere sammenslåing av kommunene, og det er derfor markert som tilkoblet. De andre blå punktene er kommunegrenser uten noen tilkobling på den andre siden.

Problemer som dette gjorde at vi ble nødt til å forsøke å koble sammen alle punktene i de kommunene som skal kobles sammen på samme måte som vi også knytter sammen de øvrige knutepunktene i kommunene – se Prosessering av data. Dette tar naturligvis noe ekstra tid og minne, men vi snakker heldigvis ikke om mer enn et par sekunder ettersom dette gjøres ved hjelp av en hashmap.

Use cases

Use Case 1: Adressesøk

Primær aktør: Beslutningstaker.

Interessenter og deres interesser:
Beslutningstaker ønsker å søke etter adresser.

Forutsetninger:
Beslutningstaker har valgt tab'en "Finn rute" i programmet.

Suksessgarantier:
Adressene blir funnet og listet ut.

Hovedsuksess-scenario:
1. Beslutningstakeren skriver inn ønsket adresseinformasjon. Gatenavn, sted, husnummer eller poststed i angitte tekstbokser.
2. Beslutningstakeren trykker på knappen "Søk".
3. Systemet lister opp alle adressene som ble funnet.

Variasjoner:
1a) Beslutningstakeren skriver ikke inn noe.
1a1) Systemet gir tilbakemelding om at ingenting ble funnet.
1a2) Beslutningstakeren gir systemet påkrevd informasjon.
3a) Ingen adresser ble funnet.
3a1) Beslutningstakeren gjennomfører et nytt søk.

Use Case 2: Velg adresser

Primær aktør: Beslutningstaker.

Interessenter og deres interesser:
Beslutningstaker ønsker å velge adresser.

Forutsetninger:
Beslutningstaker har gjennomført et adressesøk.

Suksessgarantier:
Beslutningstaker får valgt ønskede adresser.

Hovedsuksess-scenario:
1. Beslutningstakeren velger ønskede adresser i listen.
2. Beslutningstakeren trykker på knappen "Legg til valgte".
3. Systemet viser valgte adresser i ny liste.

Variasjoner:
1a) Beslutningstakeren vil velge alle adressene i listen.
1a1) Beslutningstakeren trykker da på knappen "Legg til alle".
1b) Beslutningstakeren vil ikke ha noen av adressene i listen.
1b1) Beslutningstakeren foretar da et nytt søk etter Use Case 1.

2a) Ingen adresser var valgt.
2a1) Beslutningstakeren velger ønskede adresser og trykker på knappen "Legg til valgte".
3a) Beslutningstakeren forteller systemet at han vil endre valget.

3a1) Beslutningstakeren velger fra den nye listen de adressene han ikke vil ha og trykker på knappen "Fjern Valgte".
3a2) Beslutningstakeren vil ikke ha noen av adressene og trykker på knappen "Fjern alle".
3a3) Systemet viser oppdatert liste.

Use Case 3: Finn rute

Primær aktør: Beslutningstaker.

Interessenter og deres interesser:
Beslutningstaker vil finne en rute mellom to eller flere punkter

Forutsetninger:
Beslutningstaker har en liste med ønskede adresser.

Suksessgarantier:
Ruten blir funnet og systemet viser korteste kjørevei mellom de ulike adressene.

Hovedsuksess-scenario:
1. Beslutningstakeren velger ønsket algoritme fra nedtrekksboksen.
2. Beslutningstakeren trykker på knappen "Finn rute".
3. Systemet viser på kart korteste vei fra alle adressene til alle adressene.

Variasjoner:
2a) Beslutningstakeren hadde valgt mindre enn to adresser.
2a1) Beslutningstakeren velger flere adresser og trykker på knappen "Finn rute".

Use Case 4: Visualisering

Primær aktør: Beslutningstaker.

Interessenter og deres interesser:
Beslutningstaker vil tilegne seg ulik informasjon på bakgrunn av hans interesser for et rutesøk.

Forutsetninger:
Beslutningstaker har funnet en gyldig rute som er visualisert på kart.

Suksessgarantier:
Systemet viser hva beslutningstakeren ønsker å se.

Hovedsuksess-scenario:
1. Beslutningstakeren krysser av et valg i en avkrysningsboks i menyen.
2. Systemet viser det valgte alternativet på kartet.

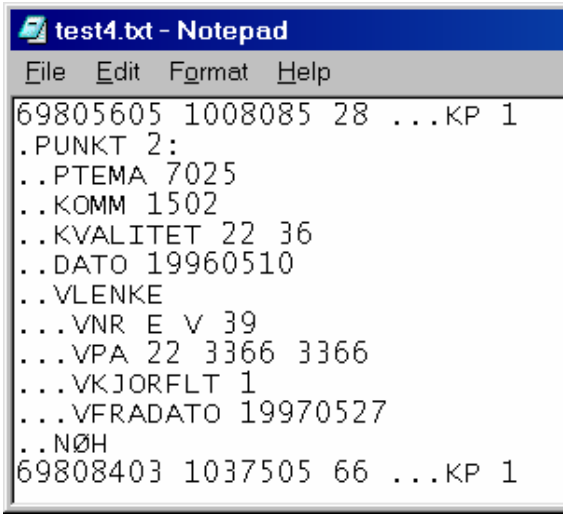
Variasjoner:
2a) Systemet viser ikke valget til brukeren på kartet.
2a1) Beslutningstakeren forstørrer kartet ved å trykke på det eller markere området hvor han vil ha det forstørret om han har krysset av et valg som ikke vises før målestokken på kartet er på en viss størrelse.
2a1a) Systemet viser valget til beslutningstakeren.

2a2) Beslutningstakeren trykker på sitt valg i menyen og trykker på knappen "Opp" for å øke prioriteten på sitt valg.
2a2a) Systemet viser beslutningstakerens valg hvis prioriteten er høy nok.

Prosessering av data

Parsing

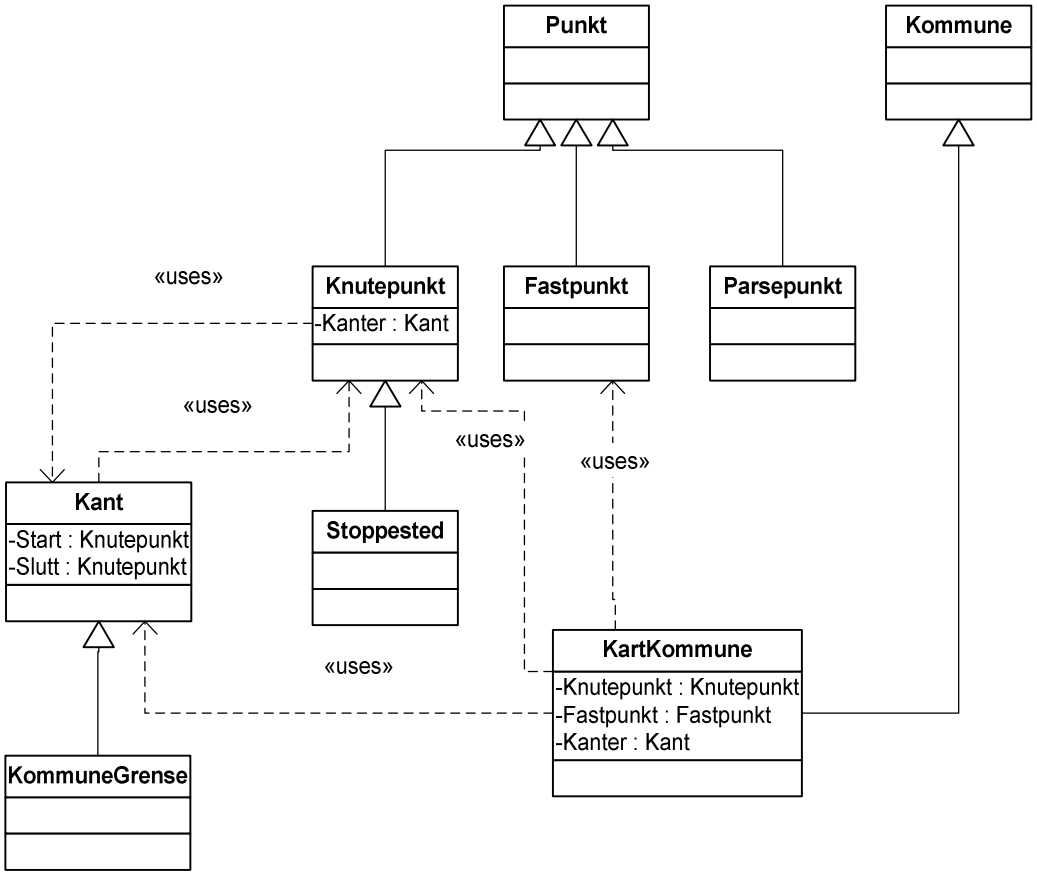
Informasjonen vi trengte var i Sosi-formatet, og det var naturlig å starte med å lese inn de dataene vi vurderte som nødvendig. Dette var geometri-, adresse- og restriksjonsdata. Dataene var lagret i enkle tekstdokumenter, og det var derfor naturlig å parse dem ved hjelp av blant annet en StringTokenizer. Nedenfor vises et eksempel på hvordan dataene er lagret i Sosi-formatet:



Det har i et visst omfang vært benyttet regulære uttrykk (Regular Expressions) for å gjenkjenne mønster i dataene. Hovedsakelig gjelder dette start på seksjoner i Sosi-filene, hvor de ofte er definert med en fast tekst, som ”Punkt” etterfulgt av et tall. I eksempelet over ”.PUNKT 2:”. For å på en sikker måte gjenkjenne starten på seksjoner, har vi derfor laget en del regulære uttrykk som vi kan teste mot. For å gjøre dette på raskest mulig måte, er disse uttrykkene pre-kompilert av Pattern-klassen. Det er derfor en relativt billig operasjon, siden Pattern-klassen slipper å tolke både mønsteret og det den skal sammenligne mot hver gang.

Et annet viktig område dette er brukt på er for å gjenkjenne koordinater. Parsepunkt-klassen har 4 forskjellige mønster den er i stand til å tolke. Når teksten vi forsøker å parse matcher det regulære uttrykket, kan vi trygt splitte det opp uten å bekymre oss for å sjekke datatyper og verdier så nøye som normalt, siden mønsteret allerede har kontrollert dette.

Klassediagram – geometridata



Over er et klassediagram som viser en oversikt over hvordan håndteringen av geometridataene er oppbygd.

Beskrivelse av geometridata

Punkt er en generell klasse som tar vare på koordinater som er lagret i Sosi-formatet med to eller tre variabler. Nord, øst og i mange tilfeller også høyde.

Parsepunkt tar et punkt beskrevet som en tekststreng på Sosi-formatet og bruker regulære uttrykk for å finne ut hvordan punktets koordinater skal behandles. I tekstfilene kan det virke litt tilfeldig om et punkt har to eller tre koordinater, og da må vi være sikker på at vi kan håndtere innlesningen på en sikker og feilfri måte.

Fastpunkt bestemmer hvilken type punkt vi har å gjøre med. Et punkt kan være et tvangspunkt, kai eller brygge, fysisk stengsel, veibom eller kommunegrense. Disse punktene har en konstant i Sosi-formatet slik at det er enkelt å finne ut hva som er hva ved innlesningen.

Knutepunkt inneholder en liste med kanter som går ut fra knutepunktet. Kant inneholder to Knutepunkt, start og slutt som forteller hvor starten og slutten på kanten er. En kant er en veglenke (en veg mellom to kryss) med til dels veldig mye informasjon som vegtype, tdnummer (unikt for hver veglenke i en kommune), lengde, fartsgrense. I tillegg inneholder de også informasjon om

kjøreretning, om det er innkjøring forbudt, maksimal vekt for kjøretøy på vegen, maksimal lengde med mer.

Stoppested inneholder en bygning. Dette er start og sluttpunkt for en rute som algoritmen kjører. Denne klassen er da på en måte et mellomledd mellom punktene og adressene, da vi lett kan få tak i koordinatene til alle bygningene via denne klassen.

Kommunegrense er til for å koble kommunene sammen. Den inneholder kun en constructor som sier hvilken kant som er en kommunegrense.

Kommune inneholder grunnleggende informasjon om kommunene. Hver kommune har et navn og et unikt kommunenummer. Tekstfilene i Sosi-formatet er også lagret kommunevis.

Kartkommune inneholder tabeller hvor koordinatene i punktet blir hashet, og dermed identifiserer punktet, det er tre slike tabeller pr. kommune for henholdsvis knutepunkt, kanter. Vi har valgt å bruke hashtabeller fordi det siler ut dubletter på en rask måte. Vi gjør det ved å først sjekke om koordinatet finnes fra før, hvis ikke legges det inn. Alternativt kopieres kantene over i det gamle punktet. For å unngå for mye rehashing har vi analysert dataene for hele Norge, og forsøkt å gi noenlunde optimale verdier. Dette fører til noe overforbruk av minne for de minste kommunene, men sparer til gjengjeld noe tid.

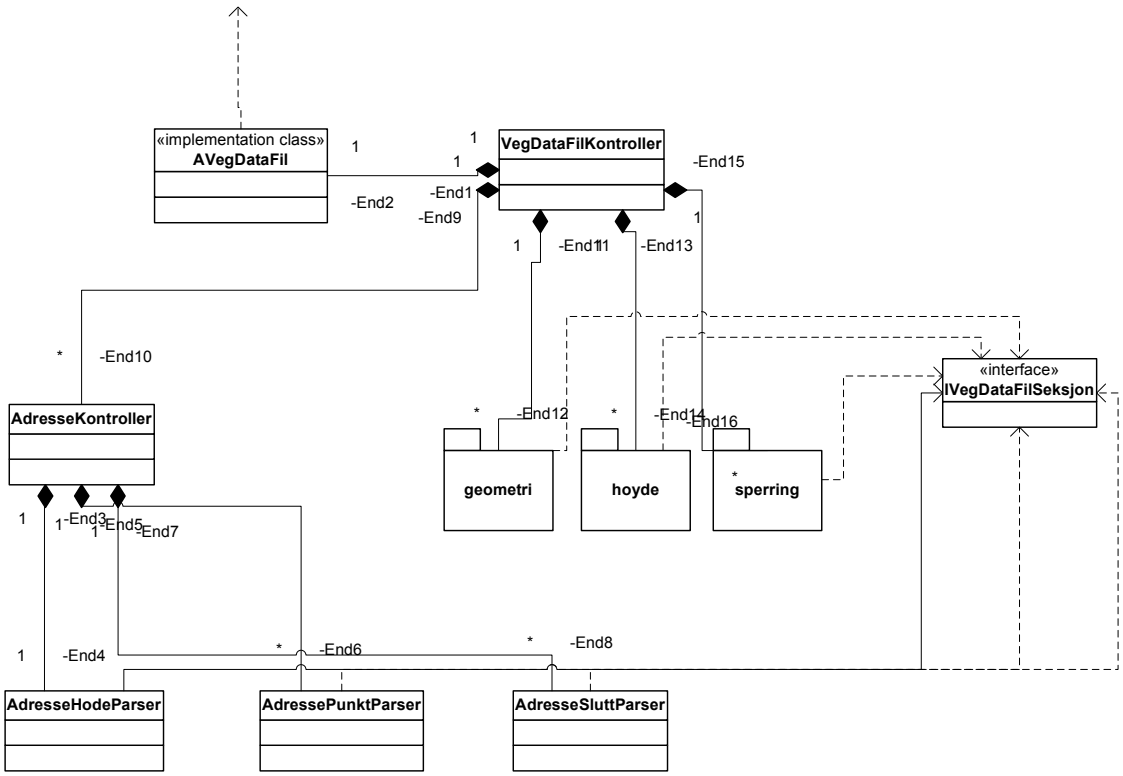
Forenklinger av geometridata

Når det gjelder veglenkene har vi forenklet systemet noe. På en vei er det i snitt et punkt for ca. hver tiende meter. Om vi skulle tatt vare på alle koordinatene fra alle veglenkene ville det da tatt opp mye minne, da det er over 21,5 millioner koordinater registrert i hele Norge. I stedet valgte vi å kun ta med start- og sluttpunktet for en veglenke og heller regne ut avstanden mellom de to punktene.

Avstanden mellom start- og sluttpunktet til en kant er nå utregnet ved at vi går igjennom også alle de andre koordinatene som finnes mellom disse to punktene. For hvert par i lista regner vi så ut den euklidske avstanden, og om det er data om høyde tilgjengelig tar vi også hensyn til dette. Summen av disse euklidske avstandene blir lagret i kanten som lengden. Disse dataene er oppgitt i desimeter, og skal være målt etter senterpunkt på vegbanen, og dermed blir resultatet tett opp til virkeligheten. Etter at dette ble gjort endte vi med ca.1,2 millioner koordinater for hele landet, altså en besparelse på rundt 20 millioner koordinater.

Et annet sted vi har valgt å forenkle systemet noe er kjøreretninger i Kant. Sosi-formatet har konstanter for dette som at ”1357” betyr at det er fire filer i den ene retningen og ”1234” betyr at det er to filer hver vei. Det er også informasjon i form av bokstaver, som angir diverse spesielle felt for avkjøring og lignende. Etter å ha studert listen over mulige verdier konkluderte vi med at henholdsvis partall og heltall var avgjørende for retningen. Vi bestemte oss for at vi ikke trengte all denne detaljerte informasjonen og har valgt å bruke tre konstanter, mot, med eller begge retninger.

Klassediagram – innlesning av data



Over er et klassediagram som viser en oversikt over klassene som har med innlesning av data å gjøre.

Innlesning av data

Klassene AdresseKontroller, AdresseHodeParser, AdressePunktParser og AdresseSluttParser ligger i en pakke adresser, i likhet med pakkene geometri, hoyde og sperring med flere. Disse pakkene inneholder tilsvarende klasser som adresse-klassene, men vi mente det ikke var hensiktsmessig å inkludere alt i klassediagrammet.

Sosi-filene er delt opp i ulike seksjoner. De tre klassene nederst i modellen vil håndtere sin egen seksjon av innlesningen. Typisk er at AdresseHodeParser behandler den øverste seksjonen. Den består typisk av versjonsnummer, tegnsett, eier, dato, hvilket område det er med mer. AdresseSluttParser tar seg av slutten av filene, mens AdressePunktParser leser alle adressene. Ikke alle Sosi-filene er like og dermed vil ikke alle trenge tre forskjellige klasser for å håndtere innlesningen på best mulig måte. For eksempel sperring som blant annet inneholder innlesning av fartsgrense på veglenker har ikke noen definert slutt-seksjon.

AdresseKontroller kontrollerer hvilken seksjon som skal lese hvilken data i filen. Videre gir den oppgaven til den første klassen som kan håndtere seksjonen. Denne håndtereren legger så inn de data den finner i kommunen, før den returnerer og det hele starter på nytt.

VegDataFilKontroller tar imot en liste med kommuner og den ønskede datatypen. Om en kommune ikke er lest fra før , eller om kommunen er registrert som ødelagt på grunn av for eksempel fjerning av løvnoder i grafen etter algoritmen, vil den etterspurte typen data (Geometri, adresse eller alt) leses inn og legges inn i kommunen.

Database

Hovedsakelig kommer datagrunnlaget i programmet fra filer i et spesielt filsystem. Dette hadde også vært en mulighet med tanke på adressene, siden de også er levert på samme format som resten av dataene. Adressedatabasen brukes til å søke opp en gitt adresse, og finne ut hvilken veglenke, i en kommune en adresse er tilknyttet. I tillegg inneholder den informasjon om koordinaten til huset (estimert senterpunkt).

Det var flere ulemper med å gjøre det på samme måten som resten av systemet:

- Store mengder data å prosessere for å finne en gitt adresse, dermed kan hastighet og minne bli et problem. Minimum lese adressedata om en kommune lineært en gang for hver oppstart eller søk, alt ettersom innlesningstid eller minneforbruk var det viktigste.
- Vi måtte laget funksjoner for å søke etter gitte adresser, og gjerne med wildcards.

Fordeler med å bruke samme måte som resten av dataene:

- Konsistens i innlesningen, i forhold til de øvrige dataene.
- Enklere oppsett/installasjon, alt på et sted.
- Felles grensesnitt mot datagrunnlaget
- Slipper å sette opp databaseserver
- Slipper å lage program for å lese adressene inn i databasen.

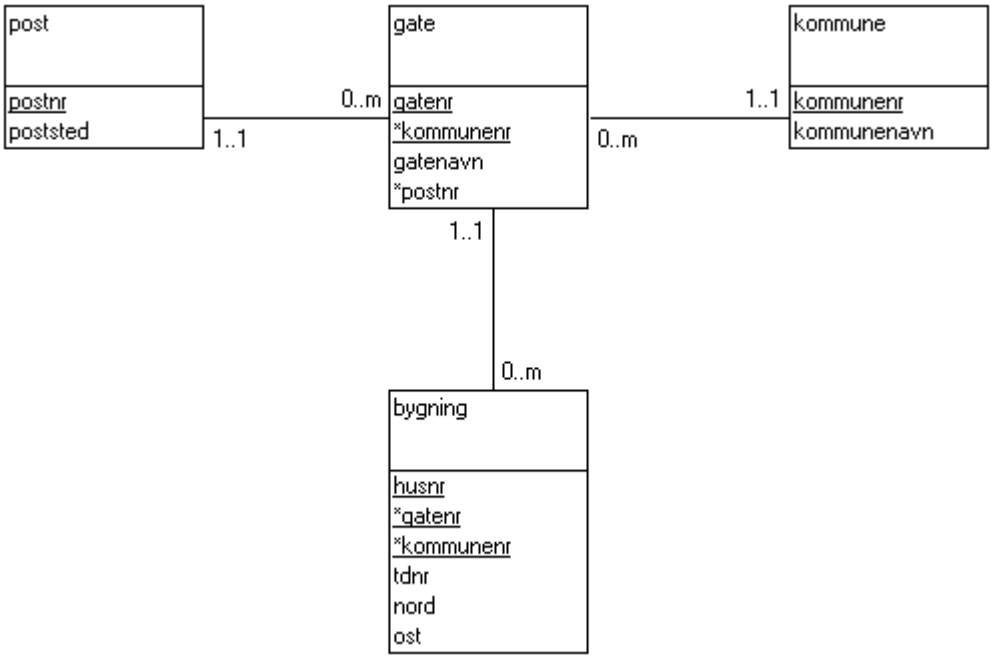
Fordeler med å bruke database:

- Fleksible søk
- Hastighet på søk
- Enkelt å bruke for utvikleren
- Databasen er normalt sparsom med minne, pga. indekser
- Kan kjøres på annen dedikert maskin
- Slipper normalt å lese inn adressedata

Vi fant ut at ulempene med å gjøre det på samme måte som resten av programmet var såpass store, med tanke på både tidsbruk og minne, at de oppveide fordelene. Effektiv søking i store datamengder, av den typen adressedatabasen nå gjør, er en av de viktigste grunner til at databaser brukes. Med tanke på den økte kompleksiteten i bakenden til programmet, og grensesnittene var det nok ikke et spesielt klokt valg. Ytelsen totalt sett er dog med stor sannsynlighet mye bedre enn hva den ville vært med dette integrert på samme måte som resten.

Siden vi har brukt JDBC (Java Database Connectivity), som er et felles grensesnitt mot databaser som støtter dette, er det enkelt å bytte ut databasesystemet med et eksisterende om det er ønskelig, såfremt disse har en JDBC-driver tilgjengelig. Programmet vårt har hovedsakelig benyttet PostgreSQL sin driver, men også MySQL sin har vært testet. Også Oracle har drivere tilgjengelig. Vi vurderte derfor å bruke skolens Oracle-database til prosjektet, men fordi den versjonen som var installert tilsynelatende ikke var oppdatert, var det ikke drivere tilgjengelig som kunne brukes i prosjektet på dette tidspunktet.

Databasemodellen:



Adressedatabasen består kun av de 4 entitetene vist over. Disse er:

- Post
    - Inneholder postnummer, og navn på poststedet.
        - Postnr: 4 siffer, men lagret som tekst pga. 0123, hvor 0 må beholdes.
        - Poststed: 30 bokstaver.

- Kommune
    - Inneholder kommunenummer og navn på kommunen.
        - Kommunenr: 4 siffer, men lagret som tekst pga. 0301, hvor 0 må beholdes.
        - Kommunenavn: 30 bokstaver

- Gate
    - Inneholder navn på en gate. Denne gaten ligger i en kommune, og har ett postnummer. I tillegg har de et gatenummer som er unikt for hver kommune.
        - Gatenr: Heltall
        - Gatenavn: 30 bokstaver
        - Fremmednøkler: postnr, kommunenr

- Bygning
    - Inneholder data om en bygning/adresse. Hver bygning har et husnummer. Dette i kombinasjon med gatenummeret og kommunenummeret, gir oss en unik identifikasjon på en bygning. Den inneholder også et Td-nummer i mange tilfeller, som forteller oss hvilken veglenke bygningen er knyttet til i denne kommunen. Dette brukes for å knytte bygningen inn i vegnettet. En del bygninger mangler dette nummeret, disse har verdi 0. Disse blir etter beste evne forsøkt innknyttet i vegnettet basert på de to siste verdiene, nord og øst koordinaten til bygningen.
        - Husnr: Heltall, forenklet i vårt system i forhold til Sosi.
        - Tdnr: Heltall.

- Nord: Heltall
- Øst: Heltall
- Fremmednøkler: gatenr, kommunenr

Grunnen til at vi ikke har opprettet egne primærnøkler i tabellene, og spesifisert de nåværende som unike kombinasjoner, er hovedsakelig for å gjøre innlesningsprogrammet enklere. På den måten slipper vi å bruke egenopprettede primærnøkler fra gate-entiteten som fremmednøkler i bygning, og sparer oss et oppslag for å finne disse når data om en bygning skal legges inn. Innlesningsprogrammet er selv ansvarlig for at disse dataene er korrekte, og på plass når de trengs under innlesningen.

Forbedringspotensiale

Det kan tenkes at fleksibiliteten i kartprogrammet kan økes ved at også øvrige kartdata blir lagret i en database. Hvordan dette er med tanke på ytelsen har vi ikke testet, men med et nytt adapter for datainnlesning skulle det absolutt være gjennomførbart å innføre dette som en hoveddatakilde i programmet. I denne sammenhengen er PostgreSQL igjen en aktuell kandidat som databasesystem, siden den støtter geometriske datatyper som standard.

Optimering av søk i databasen

Det første trinnet som bør gjøres etter at dataene er lagt inn i databasen, eller det har foregått andre større endringer i tabellene er å kjøre "VACUUM ANALYZE". Vacuum går først over alle tabellene og sørger for å slå sammen ledig plass som måtte være til overs etter sletting av rader. For oss er ikke det noe stort poeng før det har skjedd større endringer.

Analyze-delen av kommandoen sørger for å oppdatere statistikk om de forskjellige tabellene. Dette gjør at query-planneren til PostgreSQL kan gjøre bedre valg med hensyn til hvilke algoritmer som skal brukes når den skal løse spørringene. Statistikken inneholder antall rader i de forskjellige tabellene, samt hvor mange sider på disken dataene er fordelt over. I tillegg er det statistikk for hver kolonne i hver tabell som blant annet inneholder statistikk om hvilke verdier som er mest vanlige i kolonnen, og hvor mange forskjellige verdier det finnes. Disse dataene blir ikke automatisk oppdatert ved normal bruk, og det kan derfor være greit å kjøre disse manuelt med jevne mellomrom.

Hvis det ikke ble opprettet indekser på fremmednøkler under opprettelsen av tabellene, slik det ble anbefalt, er det nå tid for å legge inn disse. Disse finnes i "adressesql.txt", under seksjonen "Indekser på fremmednøkler".

PostgreSQL har flere typer indekser som kan brukes. R-tre-indekser er uaktuelle siden vi ikke bruker den typen sammenligninger i spørringene som R-tree yter bra på "geometriske" (spatial) data. Dette kunne vært aktuelt om vi hadde all kartdata registrert i databasen, men det er ikke tilfellet. En mer aktuell type indeks er Hash. Ifølge manualen er ikke ytelsen til en Hash-indeks noe bedre enn et B-tre i praksis, og det tar mye lengre tid å opprette og vedlikeholde denne typen indeks. Brukeren av Hash-indekser er ikke anbefallt på dette tidspunktet av PostgreSQL. Det finnes også en type indeks som kalles GiST (Generalized Search Tree), dette er ikke en spesiell type indeks, men et rammeverk for å bygge ut databasen med mer spesialiserte og tilpassede indekser. Det er også noen begrensninger med hensyn til noen datatyper, og flerbrukermiljø. Vi har ikke sett noe nærmere på disse. Vi har derfor falt tilbake på den mest vanlige typen indekser, B-trær. B-trær er standardvalget i PostgreSQL hvis ingenting annet blir spesifisert. Query planneren til PostgreSQL tar i bruk denne typen trær i de vanligste sammenligningene, <, <=, =, => og > og i vårt tilfelle er det = som blir brukt. I visse

situasjoner kan også disse indeksene brukes ved søk etter deltreff. Dette er en valgmulighet i programmet vårt. Dette blir dog ikke utnyttet, siden vi ikke har gjort det slik at deltreff kun kan ha wildcard til slutt i kravet. B-trær vil kunne brukes i tilfeller hvor man søker etter "gatenavn ~ '^GLOM'" men ikke "gatenavn ~ 'GLOM'". Vi bruker den siste varianten. Dette kan være aktuelt å endre hvis det viser seg nødvendig på et senere tidspunkt, og eventuelt la brukeren selv få taste inn wildcard i grensesnittet, eller velge det fra en meny. (Begynner med, slutter med osv).

Det er ikke aktuelt å opprette multikolonneindekser med de spørringene vi utfører.

Det blir automatisk generert unique-indekser på primærnøkler, så i de tilfellene er det unøndvendig å opprette nye. Ingen av de søkbare kolonnene i våre tabeller er unike. Det finnes flere kommuner med samme navn, men i forskjellige fylker, for eksempel Nes, Os og Bø. Det samme gjelder også poststed, hvor mange postnummer for eksempel har poststed Molde.

Neste trinn er nå å legge inn indekser på alle kolonnene det er aktuelt å søke i. Disse ligger under "Indekser på søkbare kolonner" i samme fil.

Hvis det skal kjøres større oppdateringer mot databasen, bør indeksene midlertidig slåes av, eller slettes, slik at databasesystemet slipper å vedlikeholde indeksene under oppdateringene. Dette kan gjøre at ytelsen synker en del under disse operasjonene.

Databaseytelse

For å beregne kjøretid og optimere søkene i databasen har vi brukt PostgreSQL sin innebygde kommando "EXPLAIN ANALYZE". Denne i kombinasjon med eksempelspørringer på virkelig data, gir oss ett estimat på tidsbruk i millisekund og kostnad i antall diskaksesser. Kostnaden som blir angitt av explain-kommandoen er estimert antall diskaksesser for å hente sider fra disken med nødvendig data. I tillegg til diskaksessene teller det også med en liten kostnad som varierer med antall resultatrader. (0.01 * antall rader). Explain-kommandoen forteller oss i hvilken rekkefølge og hvordan query-planleggeren vil utføre spørringen.

På disse første resultatene er det kun kjørt Explain, uten analyze - men de gir fremdeles et godt inntrykk av hvor stor forbedring vi kan forvente oss når det gjelder ytelsen på spørringene. Alle testene har brukt samme spørring. Denne er den enkleste vi regner med blir kjørt, og er et enkelt søk etter gatenavn, med maksimalt 100 resultater returnert (dette er standard i programmet).

Kommandoen som ble kjørt:

```
explain SELECT k.kommunenr, k.kommunenavn, p.postnr, p.poststed, g.gatenr, g.gatenavn, b.husnr,
b.tdnr, b.nord, b.ost
FROM bygning b, gate g, post p, kommune k
WHERE p.postnr = g.postnr
AND k.kommunenr = g.kommunenr
AND b.gatenr = g.gatenr
AND b.kommunenr = g.kommunenr
AND g.gatenavn = 'GLOMSTUVEGEN'
ORDER BY g.gatenavn, b.husnr
LIMIT 100;
```

Det første resultatet er tatt rett etter innlegging av data, uten andre indekser enn de som automatisk er opprettet på primærnøklene i de fire tabellene. Det var heller ikke kjørt "VACUUM ANALYZE" før dette resultatet.

Resultatet av planen sin første linje kan leses som følger:
Hvilken operasjon som utføres
Det har kostet oss 410977,74 diskaksesser før resultatet kan begrenses ned til de 100 radene vi har sagt vi maksimalt vil ha tilbake fra spørringen.
Kostnaden etter at Limit-kommandoen også har kjørt. Som vi ser er limit en billig operasjon i dette tilfellet, den har knapt økt kostnaden i det hele tatt.
Estimert antall rader denne operasjonen vil returnere.
Det siste resultatet er bredden på resultatet i bytes.

Som vi ser av planen under, er den estimerte totalte kostnaden i antall diskaksesser for å hente sider på ca 411 000 stykker. Statistikk fra databasen viser at tabellen bygning bruker ca. 12700 sider, og det er den desidert største tabellen med i overkant av 1,7 millioner rader. Til sammenligning tar gate-tabellen ca 750, post-tabellen 20 og kommune-tabellen 3.

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
Limit  (cost=410977.74..410977.99 rows=100 width=241)
Sort  (cost=410977.74..411028.90 rows=20463 width=241)
Sort Key: g.gatenavn, b.husnr
Merge Join  (cost=394712.33..407871.75 rows=20463 width=241)
Merge Cond: (("outer".kommunenr = "inner".kommunenr) AND ("outer".gatenr = "inner".gatenr))
Sort  (cost=2128.83..2130.02 rows=474 width=241)
Sort Key: g.kommunenr, g.gatenr
Hash Join  (cost=1961.62..2107.77 rows=474 width=241)
Hash Cond: ("outer".kommunenr = "inner".kommunenr)
Merge Join  (cost=1953.19..2092.23 rows=474 width=162)
Merge Cond: ("outer".postnr = "inner".postnr)
Index Scan using post_pkey on post p  (cost=0.00..124.35 rows=3032 width=79)
Sort  (cost=1953.19..1954.38 rows=474 width=99)
Sort Key: g.postnr
Seq Scan on gate g  (cost=0.00..1932.12 rows=474 width=99)
Filter: ((gatenavn)::text = 'GLOMSTUVEGEN'::text)
Hash  (cost=7.34..7.34 rows=434 width=79)
Seq Scan on kommune k  (cost=0.00..7.34 rows=434 width=79)
Sort  (cost=392583.49..396900.57 rows=1726832 width=36)
Sort Key: b.kommunenr, b.gatenr
Seq Scan on bygning b  (cost=0.00..29966.32 rows=1726832 width=36)
(21 rows)
```

Etter at indekser på fremmednøkler ble lagt på relasjonene mellom alle de fire tabellene fikk vi en markant forbedring totalt sett fra 411 000 aksesser før til et estimat på 88 510 nå. Denne forbedringen kommer av at vi nå sparer oss flere steg. Blant annet sparer vi 2 sorteringer, en sekvensiell lesing av bygning-tabellen, og en stor sammenslåing basert på kommunenr og gatenr.

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
```

```
Limit  (cost=88509.04..88509.29 rows=100 width=241)
Sort  (cost=88509.04..88560.20 rows=20463 width=241)
Sort Key: g.gatenavn, b.husnr
Nested Loop  (cost=1961.62..85403.05 rows=20463 width=241)
Hash Join  (cost=1961.62..2107.77 rows=474 width=241)
Hash Cond: ("outer".kommunenr = "inner".kommunenr)
Merge Join  (cost=1953.19..2092.23 rows=474 width=162)
Merge Cond: ("outer".postnr = "inner".postnr)
Index Scan using post_pkey on post p  (cost=0.00..124.35 rows=3032 width=79)
Sort  (cost=1953.19..1954.38 rows=474 width=99)
Sort Key: g.postnr
Seq Scan on gate g  (cost=0.00..1932.12 rows=474 width=99)
Filter: ((gatenavn)::text = 'GLOMSTUVEGEN'::text)
Hash  (cost=7.34..7.34 rows=434 width=79)
Seq Scan on kommune k  (cost=0.00..7.34 rows=434 width=79)
Index Scan using fk1_gate_bygning on bygning b  (cost=0.00..175.08 rows=43 width=36)
Index Cond: ((b.gatenr = "outer".gatenr) AND (b.kommunenr = "outer".kommunenr))
(17 rows)
```

Neste trinn som ble utført var "VACUUM ANALYZE", og det burde nok vært gjort tidligere for best sammenligning. Dette resultatet viser kostnaden etter at planleggeren har fått oppdaterte statistiske data om tabellene i databasen. Dette gjør at den kan ta bedre avgjørelser når det gjelder hvilke teknikker den skal benytte for å få en best mulig utføring av spørringen. Vi kan for eksempel se at den nå har valgt å ikke utføre noen hashing.
Kostnaden har gått ytterligere ned, og er nå estimert til 2170 mot 88 510 før. Nå er også estimatet på antall rader i resultatet et annet enn det LIMIT var satt til i spørringen. Det faktiske antallet rader denne spørringen returnerer er 57, mot det estimerte 47.

Under er et utvalg med informasjon analysen genererte om tabellen bygning. Tabellen er den desidert største med sine 1,7 millioner rader. Vi ser for eksempel antall reltuples - altså antall rader i tabellen, og antallet sider denne tabellen bruker på disken (relpages).

```
SELECT relname, relkind, reltuples, relpages FROM pg_class where relname LIKE 'bygning';
relname | relkind |  reltuples  | relpages
---------+---------+-------------+----------
bygning | r       | 1.72683e+06 |    12698
```

Under er statistikk om de forskjellige kolonnene i tabellen bygning. Første kolonne er kolonnenavn, andre er antallet forskjellige verdier denne kolonnen inneholder. Tredje og siste kolonne er de mest vanlige verdiene i denne kolonnen. Vi ser at for husnr er dette som forventet de lave tallene.

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'bygning';
attname  | n_distinct |                                most_common_vals
-----------+------------+-------------------------------------------------------------------------------------------
husnr     |        374 | {1,2,3,5,8,4,11,7,6,16}
gatenr    |       2411 | {1,18,14,10,37,4,21,5,25,77}
kommunenr |        417 | {1201,0301,1601,1103,0106,0219,0412,0709,0105,1102}
tdnr      |       4787 | {0,100746,100564,100573,100852,101405,100019,100236,100266,100281}
```

```
nord      |  -0.191086 |
{65571270,65663330,65722000,65972510,66463300,66514830,66518210,66520960,67348680,692890
20}
ost       |  -0.156417 | {-378410,-
331400,538730,720750,1900570,2575720,2586130,2672410,2679970,2714720}

QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Limit  (cost=2169.67..2169.79 rows=47 width=69)
Sort  (cost=2169.67..2169.79 rows=47 width=69)
Sort Key: g.gatenavn, b.husnr
Nested Loop  (cost=1932.42..2168.37 rows=47 width=69)
Nested Loop  (cost=1932.42..2041.03 rows=15 width=61)
Merge Join  (cost=1932.42..1952.13 rows=15 width=50)
Merge Cond: ("outer".kommunenr = "inner".kommunenr)
Index Scan using kommune_pkey on kommune k  (cost=0.00..18.40 rows=434 width=19)
Sort  (cost=1932.42..1932.46 rows=15 width=31)
Sort Key: g.kommunenr
Seq Scan on gate g  (cost=0.00..1932.12 rows=15 width=31)
Filter: ((gatenavn)::text = 'GLOMSTUVEGEN'::text)
Index Scan using post_pkey on post p  (cost=0.00..5.91 rows=1 width=19)
Index Cond: (p.postnr = "outer".postnr)
Index Scan using fk1_gate_bygning on bygning b  (cost=0.00..8.46 rows=2 width=28)
Index Cond: ((b.gatenr = "outer".gatenr) AND (b.kommunenr = "outer".kommunenr))
(16 rows)
```

Resultatet over viser at nesten hele kostnaden på spørringen kommer fra en sekvensiell lesing av tabellen gate, den er på hele 1932 av 2170. Løsningen her var å legge på ytterligere en indeks på den kolonnen vi sammenlignet på, nemlig gatenavn. Kostnaden på lesingen av tabellen gate er nå gått ned fra 1932 til ca. 60. Den totalte kostnaden på hele spørringen har gått ned tilsvarende, og er nå 313.

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
Limit  (cost=312.84..312.97 rows=52 width=69)
Sort  (cost=312.84..312.97 rows=52 width=69)
Sort Key: g.gatenavn, b.husnr
Nested Loop  (cost=60.70..311.35 rows=52 width=69)
Nested Loop  (cost=60.70..175.26 rows=16 width=61)
Merge Join  (cost=60.70..80.43 rows=16 width=50)
Merge Cond: ("outer".kommunenr = "inner".kommunenr)
Index Scan using kommune_pkey on kommune k  (cost=0.00..18.40 rows=434 width=19)
Sort  (cost=60.70..60.74 rows=16 width=31)
Sort Key: g.kommunenr
Index Scan using idx_gate_gatenavn on gate g  (cost=0.00..60.38 rows=16 width=31)
Index Cond: ((gatenavn)::text = 'GLOMSTUVEGEN'::text)
Index Scan using post_pkey on post p  (cost=0.00..5.91 rows=1 width=19)
Index Cond: (p.postnr = "outer".postnr)
Index Scan using fk1_gate_bygning on bygning b  (cost=0.00..8.48 rows=2 width=28)
Index Cond: ((b.gatenr = "outer".gatenr) AND (b.kommunenr = "outer".kommunenr))
(16 rows)
```

I tillegg har vi her også fått en analyse på tidsbruken ved hjelp av "explain analyze". Her er tilsvarende indekser også lagt inn på alle de andre kolonnene vi søker i, og som ikke hadde indekser fra før (kommunenavn, poststed og husnr). Ingen av disse kolonnene har unike verdier, eller andre spesielle egenskaper som gjør det aktuelt å bruke noe annet enn et standard b-tre som indeks.

Legg merke til den ekstra seksjonen til slutt på resultatlinjene, hvor det nå er med et tidsestimat på utførelsen av spørringen. Som vi ser er resultatet nå marginalt bedre enn forrige plan. Estimatet på tidsbruken er ikke avskrekkende i dette tilfellet, ca 9 ms for å hente ut et utvalg rader etter gatenavn, med data fra 4 tabeller. Selve oppkoblingen mot databasen, og eventuelt å sende søket over internett er betydelig høyere enn tiden det tar å utføre selve spørringen.

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
-----------------------------------
Limit  (cost=300.29..300.42 rows=51 width=69) (actual time=8.069..8.266 rows=57 loops=1)
Sort  (cost=300.29..300.42 rows=51 width=69) (actual time=8.061..8.119 rows=57 loops=1)
Sort Key: g.gatenavn, b.husnr
Nested Loop  (cost=56.38..298.84 rows=51 width=69) (actual time=2.872..4.178 rows=57 loops=1)
Nested Loop  (cost=56.38..165.00 rows=15 width=61) (actual time=2.772..2.797 rows=1 loops=1)
Merge Join  (cost=56.38..76.09 rows=15 width=50) (actual time=2.693..2.711 rows=1 loops=1)
Merge Cond: ("outer".kommunenr = "inner".kommunenr)
Index Scan using kommune_pkey on kommune k  (cost=0.00..18.40 rows=434 width=19) (actual
time=0.042..1.396 rows=260 loops=1)
Sort  (cost=56.38..56.42 rows=15 width=31) (actual time=0.161..0.163 rows=1 loops=1)
Sort Key: g.kommunenr
Index Scan using idx_gate_gatenavn on gate g  (cost=0.00..56.09 rows=15 width=31) (actual
time=0.132..0.141 rows=1 loops=1)
Index Cond: ((gatenavn)::text = 'GLOMSTUVEGEN'::text)
Index Scan using post_pkey on post p  (cost=0.00..5.91 rows=1 width=19) (actual time=0.057..0.061
rows=1 loops=1)
Index Cond: (p.postnr = "outer".postnr)
Index Scan using fk1_gate_bygning on bygning b  (cost=0.00..8.90 rows=2 width=28) (actual
time=0.081..0.803 rows=57 loops=1)
Index Cond: ((b.gatenr = "outer".gatenr) AND (b.kommunenr = "outer".kommunenr))
Total runtime: 8.718 ms
(17 rows)
```

Vi har her vist at ytelsen kan forbedres dramatisk med enkle grep på mange av spørringene. Dessverre utnyttes ikke indeksene i alle situasjonene, hvis resultatsettet av deler av spørringen returnerer store mengder rader, før begrensningen i limit slår til, kan den gjøre valg som setter indeksene ut av funksjon, og utfører et sekvensielt søk i stedet.

Tilgjengelige algoritmer, opptegningstyper og resultathåndterere

I kart.xml kan man angi hvilke algoritmer og opptegningsmuligheter som skal være tilgjengelig for programmet. For både opptegningslagene, algoritmene og resultathåndtererne er det klassenavnet som spesifiseres, inkludert hvor i pakkehirarkiet klassen ligger. Alle lagene som er spesifisert i filen blir etter tur forsøkt instansiert, og hvis opprettelsen er vellykket vil de være tilgjengelig for

programmet. Algoritmene vil være valgbare fra "Finn rute" i programmet. Opptegningslagene listes opp under "Visualisering". Resultathåndtererne er ikke synlige eller valgbare i grensesnittet.

I prosjektbeskrivelsen var det ønsket utregning av reisetid som et mulig resultat. Dette har ikke blitt implementert i prosjektet av oss foreløpig, hovedsaklig på grunn av tidspress mot slutten. I våre samtaler med Johan Oppen, som er primær bruker av systemet, kom det frem at de data han mottok pr. dags dato gjelder avstander. Vi har derfor implementert et enkelt eksempel på hvordan disse dataene kan hentes ut av systemet på forskjellige måter, ved hjelp av resultathåndtererne. Disse håndtererne har tilgang på alle data om ruten, inkludert kryss og veglenker. Det skal derfor være en smal sak å lage resultathåndterere som i stedet for avstand, eller i tillegg for den del, kan regne ut kjøretid. Disse har tilgang både på avstandsdata, hastighet, vegkvalitet mm. der disse data eksisterer, og det skulle derfor være lett å estimere kjøretid.

Data om fergestrekninger eksisterer. Veglenker som er fergestrekninger er markert med en egen kode, samt at fergekaiene er markert som fastpunkt. Det burde derfor være mulig i resultathåndtererne å skrive kode som gjenkjente disse strekningene, basert på for eksempel koordinat eller tdnr (i kombinasjon med den nevnte merkingen) - og så gikk til en ekstern datakilde for å hente data om avgangstidene. Ved å integrere disse to tenkte resultathåndtererne, burde det ønskede resultatet absolutt være gjennomførbart.

Algoritmeklassene må arve fra klassen RuteBeregningAlgoritme, som er en abstrakt klasse som definerer en del av fellesmetodene algoritmene trenger. Denne abstrakte klassen sørger også for å sende beskjed til resten av grensesnittet når algoritmen er kjørt ferdig, samt at den har startet. Det er valgfritt for algoritmene om de også vil sende fremgangsmeldinger, som sørger for å oppdatere progressbaren i grensesnittet. Hvis de ikke sender slike meldinger, vil den vise en kontinuerlig oppdatering frem og tilbake, for å indikere at en beregning pågår. Den abstrakte klassen er også en tråd, slik at alle algoritmer automatisk kjører uavhengig av grensesnittet, og uten at disse selv må sørge for synkronisering og tilbakekall. Det må være minst en algoritme tilgjengelig for programmet.

Den abstrakte klassen sørger for å nullstille grafen mellom hver beregning. Det er også meningen at den skal ta seg av preprosessering og vekting av kantene i grafen. Den har tilgjengelig en metode fjernUviktigeNoder() som fjerner løvnoder fra grafen inntil det ikke er flere igjen. Det er også to abstrakte metoder hvor et navn og en beskrivelse av algoritmen skal returneres. Disse vises i grensesnittet. Beskrivelsen er ment benyttet som en innebygget brukerveiledning, hvor man kan spesifisere når denne algoritmen bør brukes.

Opptegningslagene har et felles grensesnitt som spesifiserer funksjoner for å få tegnet opp et BufferedImage. Det er viktig at dette bildet er av en type som er transparent, siden disse settes sammen etterpå, etter prioritet. Det kan være greit å ta utgangspunkt i klassene lagKanter (tegner opp veger mellom de angitte knutepunktene i en kant) eller lagKnutepunkt (som tegner opp og fargekoder knutepunktene, avhengig av antall kanter) alt ettersom hvilken type opptegning man ønsker å gjøre. Opptegningslagene vi har skrevet i dette prosjektet sørger for å lagre bildet i minne etter at det er opptegnet. Slik forhindrer vi at man må tegne alle lagene på nytt hver gang man endrer prioritering, eller når man skrur av/på et av lagene.

I tillegg til metoden for å få tegnet bildet, er det tilsvarende metoder som for algoritmene som angir navn på laget, og en beskrivelse. Beskrivelsen her er tenkt til å forklare for eksempel fargekoding i kartet. Det er også metoder for å sjekke om laget er påslått, og en metode for å slå det av/på. Det må være minst ett opptegningslag tilgjengelig for programmet.

Resultathåndtererne har også et enkelt felles grensesnitt. Dette grensesnittet spesifiserer to metoder, hvor man kan sende inn ett array med stoppesteder. Disse inneholder normalt ruter som resultathåndtererne kan bruke for å hente ut interessant informasjon. Via stoppestedene og listen med ruter mellom disse kan man få ut komplett vei mellom to stoppesteder av gangen fra start til slutt, inkludert posisjoner, vegkvalitet og lengde osv. Den andre metoden styrer om metoden er slått på eller ikke. Dette gjør at man kan liste opp alle håndtererne i kart.xml, men man trenger ikke nødvendigvis bruke alle samtidig. Fordelen med dette er at det er lett å skru de av på, uten å ha tilgang til å gjøre det via grensesnittet.

I utgangspunktet har vi laget tre relativt enkle eksempler på hvordan dette kan utnyttes. Den ene klassen skriver informasjon til konsollet, og den andre til en fil i brukerens hjemmekatalog. Disse to går igjennom alle rutene, finner start og slutt, og skriver også ut hele veien mellom disse stedene, relativt detaljert. For hver rute skriver den også totalt antall hopp og total lengde på ruten. Den tredje varianten skriver også til fil, men betydelig mindre datamengder. Den skriver ut start, slutt og antall hopp, samt total lengde på ruten. De to handlerne som skriver til fil har definert hvert sitt prefiks på resultatfilene, henholdsvis Rute og EnkelRute. Disse nummereres stigende, og blir skrevet til brukerens katalog.

Hvilke resultathåndterere som er aktivert bestemmes av kart.xml. Alle resultathåndterere som er definert der vil gjøre jobben etter tur. Det er ikke nødvendig å ha noen resultathåndterere om det ikke er ønskelig med annet enn visualisering fra opptegningslagene. De tre eksempelklassene kan potensielt generere enorme mengder data, og det tar gjerne lengre tid enn selve søket. Et eksempel er alle til alle 100 adresser i Oslo, som genererte ca 46 mb resultatdata i vår test.

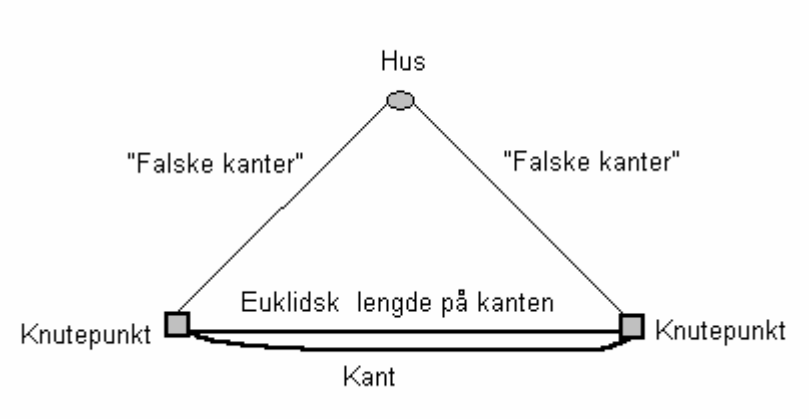
Vær oppmerksom på at rekkefølgen som er angitt i kart.xml normalt tilsvarer rekkefølgen valgene blir utført og/eller vist i grensesnittet.
Kart.xml er en xml-fil med sitt eget spesifiserte format. Dette formatet er dessverre spesifisert internt i xml-fila, siden det viste seg å være vanskelig å få pekt på formatet uten bruk av full filbane til formatet, eller alternativt å legge det på en nettside.

Algoritme

Preprosessering av data for algoritmen

Før algoritmen kan kjøres, er det nødvendig med noe opprydding/preprosessering av grafen. Det første som må gjøres er å knytte adressene til grafen, ettersom de ikke er knyttet til knutepunkter, men et tdnummer – altså en kant. Den opprinnelige planen var at adressene skulle knyttes til nærmeste knutepunkt (egentlig gatekryss), ved å hente ut endepunktene for kanten adressen var knyttet til, beregne euklidsk avstand og velge det nærmeste gatekrysset. Problemet med denne løsningen er at den vil gå på bekostning av optimalitet – vi bestemmer at algoritmen skal starte på et gitt knutepunkt, uten å vite om dette inngår i den optimale, korteste veien. Løsningen ble å opprette en egen klasse Sluttpunkt som arver Knutepunkt, og la adressen være et sluttpunkt – altså en node i grafen. Deretter hentet vi ut gatekryssene fra kanten adressen var knyttet til, og opprettet ”falske” kanter fra sluttpunktet til disse knutepunktene. Dermed blir hver adresse knyttet inn i grafen med to kanter, algoritmen kan startes og avsluttes i disse sluttpunktene, og løsningens optimalitet garanteres ettersom det nå er algoritmen som avgjør via hvilket knutepunkt den korteste veien går. Vekten på disse ”falske” kantene ble først satt til den euklidske avstanden mellom sluttpunktet og det aktuelle knutepunktet, men dette førte i noen tilfeller til at algoritmen tar veien via sluttpunktet fordi lengden på de to ”falske” kantene blir kortere enn den egentlige veien sluttpunktet er knyttet til – veien/kanten er ikke nødvendigvis en rett linje mellom knutepunktene. Dette løste vi ved å gange

opp den euklidske avstanden med forholdet mellom den egentlige veilengden og den euklidske avstanden mellom knutepunktene.



En annen, og muligens bedre, måte dette kunne vært løst på, er å benytte seg av "spesialklassen" Sluttpunkt. Ved å nekte algoritmen å gå via sluttpunkt i korteste vei-beregningen vil dette unngås – algoritmen blir nødt til å bruke den ordentlige veien. Dette vil muligens også være en mer "nøyaktig" måte å gjøre det på, ettersom beregningen vi gjør med ratio osv teoretisk sett kan feile – vi har ingen garanti for at algoritmen aldri går via sluttpunkter når den finner korteste vei.

Et annet problem med å knytte adresser til grafen, er at ikke alle adresser er knyttet til en kant – ca 80 000 adresser mangler tilknytning til et td-nummer. For disse adressene har vi ganske enkelt brukt brute force, loopet gjennom alle knutepunkt i kommunen, og knyttet sluttpunktet til det nærmeste knutepunktet med en "falsk" kant. Dette vil ikke alltid stemme – for eksempel er det mulig at nærmeste knutepunkt ligger i en annen kommune. For disse adressene kan vi derfor ikke garantere optimalitet. Vi valgte å gjøre det slik fordi det var enkelt, det gjelder relativt "få" adresser, og fordi vi hadde viktigere og mer "fornuftige" ting å bruke tiden på. Det kan her tenkes en utvidelse hvor man ut ifra en kommunes fastpunkt av typen kommunegrense, identifiserte nabokommuner, og kontrollerte disse også.

Ved kryssing av kommunegrenser oppstod det et spesialtilfelle fordi knutepunktene som markerer kommunegrensen ikke alltid er knyttet sammen. For å unngå å gjøre endringer i en eller begge av kommunene som var destruktive, eller som innebar mye kopiering og flytting av kanter osv. ble det valgt å lage en egen KommuneGrense-klasse for å knytte sammen kommunegrensepunkt som i realiteten ligger på samme sted, med identiske koordinater. Denne arver Kant-klassen, og den eneste forskjellen er at kantens vekt er satt til 0. Dette gjør at når algoritmene vi har implementert kommer til en kommunegrense, vil de først prioritere å hente ut knutepunktet i den andre enden og sjekke kantene der. Hvis kommunegrensen krysser mellom 3 kommuner, vil den da fortsette også over neste kommunegrense-objekt slik at også den tredje kommunens knutepunkt blir hentet ut tidlig, og dette vil skje før den plukker den vegen videre som nå har lavest vekting.

En effektiv måte å få ned kjøretiden på, er å fjerne overflødige noder i grafen før algoritmen kjøres – kjøretiden går naturlig nok ned når antallet noder minker. Vi har laget en slik funksjon som rydder i grafen – se Algoritme med rydding.

Dijkstras algoritme – korteste vei i veid graf

Vi valgte å bruke Dijkstra fordi det er en enkel algoritme som vi kjenner godt, den kan med enkle grep gjøres mer effektiv, og den er blant de bedre (beste?) til å finne korteste vei fra én til én. Dijkstra var med andre ord et godt utgangspunkt for å lage noe som taklet korteste vei på ca 800 adresser på mindre enn 100 timer.

Dijkstras algoritme bruker labelling eller markering for å løse korteste vei-problemet, en sentral metode i mange korteste vei-algoritmer. Noder markeres som kjent eller ukjent, og algoritmen avsluttes når alle noder er markert som kjent. Outputen fra algoritmen er et korteste vei-tre som går fra en startnode til et sett av kjente noder – dvs noder som har en kjent korteste avstand fra startnoden. Korteste vei-treet konstrueres i steg. Dijkstra er også en grådig algoritme, og løser dermed problemer ved å velge den muligheten som virker best ved hvert steg [1]. En av Dijkstras styrker i forhold til andre korteste vei-algoritmer, er muligheten til å avslutte algoritmen når sluttnoden erklæres kjent – andre algoritmer må ofte finne hele korteste vei-treet før optimale svar kan garanteres. Dijkstra er derfor svært nyttig når man skal finne korteste vei mellom to punkter som er i nærheten av hverandre, eller når grafen/søkeområdet er svært stort [5].

Graf

Grafen består av noder V (vertices) og kanter E (edges) – antall kanter/antall noder-ratio for vår graf er ca 1.14.
Hver node tilsvarer et knutepunkt/gatekryss, og inneholder følgende informasjon:

- x og y koordinater
- liste over kanter ut fra knutepunktet
- kostnadsvariabelen dist, som summerer opp avstanden/kostnaden til kantene som benyttes når man kjører fra startpunktet til det aktuelle knutepunktet. Dist er i tillegg grunnlaget for utvelgelse av neste knutepunkt, og settes lik Integer.MAX_VALUE før første gjennomkjøring av algoritmen
- Knutepunkt path, som inneholder "forelderen" til dette knutepunktet. Variabelen forteller hvor den korteste veien går når algoritmen er ferdig – man kan enkelt loope bakover fra sluttpunktet via path, via path.path osv. til startnoden. Path settes lik null før første gjennomkjøring av algoritmen
- iterasjon, forteller hvilken iterasjon av algoritmen som sist oppdaterte dette knutepunktet. Brukes for å nullstille grafen mer effektivt mellom kjøringer av algoritmen – se Alle til alle
- + noen andre variabler som brukes i opptegning av rute/graf

Hver kant tilsvarer en gate mellom to gatekryss, og inneholder følgende informasjon:

- kostnadsvariabelen lengde/vekting, lengden på kanten. Denne kostnadsvariabelen kan settes til "hva som helst" – variabelen bestemmer om algoritmen skal kjøres på kjøretid, avstand eller andre kriterier. Her kan det også legges til penalties (dersom man beholder den opprinnelige verdien i en egen variabel, slik at man kan beregne korrekt rutelengde uten penalties til slutt), se Forslag til forbedringer av algoritmen
- Knutepunkt start og slutt, gatekryssene kanten knytter sammen
- all informasjon om veien: lengde, veistandard, veitype, antall kjørefelt, kjøreretning osv

Pseudokode for Dijkstras algoritme

```
     void dijkstra (Vertex s)
   {
     Vertex v, w;

/*1*/   s.dist = 0;

/*2*/   for(; ;)
      {
/*3*/     v = smallest unknown distance
vertex;
/*4*/     if (v == null)
/*5*/       break;
/*6*/     v.known = true;

/*7*/     for each w adjacent to v
/*8*/       if (!w.known)
/*9*/         if (v.dist + cvw < w.dist)
          {     //Update w
/*10*/        decrease (w.dist to v.dist +
cvw);
/*11*/      w.path = v;
          }
       }
       }
```

[5]

Initialisering:

- dist settes til Integer.MAX_VALUE
- path settes til null
- known settes til false

Algoritmen tar startnoden som parameter.

1. Startnodens dist settes lik 0.
2. Starter algoritmen – gjentas til alle noder er kjent, alternativt sluttnoden er funnet.
3. Finner neste node som skal behandles – plukker den ukjente noden v som har kortest avstand til startnoden. Ved første iterasjon vil dette være selve startnoden.
6. v settes til kjent – v.dist er nå endelig bestemt og garantert optimal.
7. Henter ut alle naboene til v – for hver nabo
8. ..som ikke allerede er kjent:
9. Sjekk avstanden. W.dist fungerer som en upper bound på avstanden fra startnoden til w - hvis w.dist er større enn v.dist + kostnaden på kanten mellom nodene v og w, finnes det en kortere vei til w via v. Oppdater w.
10. Senk w.dist til v.dist + kostnaden til kanten vw.
11. Sett w.path til v – korteste vei til w går via v. Gjenta 8 – 11 for alle naboer av v – gå til 3, gjenta til alle noder er kjent og/eller sluttnode er kjent.

Datastruktur

Kjøretiden til Dijkstra avhenger fullstendig av hvordan man velger å håndtere nodene. Med en naiv implementering av algoritmen med alle ukjente noder liggende i f.ex. et array, vil kjøretiden bli

$O(|V|^2)$. Det er derfor svært mye tid å spare på å velge en effektiv datastruktur. Følgende operasjoner vil utføres ofte på datastrukturen:

- insert
- delete/remove min
- find
- update

Update-funksjonen blir svært tidkrevende – man må først bruke find for å finne riktig node, avstands- og forgjengerinformasjon i noden oppdateres. Deretter må noden flyttes i datastrukturen – dersom nodene ligger sortert på avstand. Den største flaskehalsen vil allikevel være deleteMin-operasjonen, utvelgelsen av neste node. Brukes et vanlig array må det søkes gjennom hele arrayet etter noden med minst dist V ganger – dermed er det her den største besparelsen vil være ved valg av en annen datastruktur. Vi har prøvd litt forskjellige strukturer – se også Forslag til forbedringer av algoritmen.

Utskrift/lagring av rute

Rutene tas primært vare på etter at algoritmen har kjørt ved å loope gjennom dem bakfra – fra sluttnoden mot startnoden. Deretter kan de skrives direkte ut, som her i en rekursiv metode:

```
/*
 * Print shortest path to v after dijkstra has
run.
 * Assume that the path exists.
 */
void printPath(Vertex v)
{
   if (v.path != null)
   {
      printPath(v.path);
      System.out.print(" to ");
   }
   System.out.print(v);
}
```

[5]

I tilfellet alle til alle ble hver rute opprinnelig lagret i form av en lenket liste, som så ble lagt til i en liste i selve stoppestedet. Når alle rutene er funnet, vil de enkelt kunne tegnes opp/skrives ut ved å loope gjennom alle stoppesteder og hente ut rutene. Problemet med denne løsningen var den lenkede listen – mer om dette under Alle til alle.

Én til én

Første implementasjon av algoritmen var en naiv versjon av Dijkstra, med en sortert vektor som datatstruktur. Algoritmen la inn alle naboer av et knutepunkt, sorterte vektoren og plukket deretter ut knutepunktet med minst dist og erklærte det som kjent. Sparte noe tid på å sortere vektoren rett før neste knutepunkt ble plukket ut, i motsetning til å sortere for hvert nye knutepunkt som ble lagt til. Denne naive versjonen fungerte helt fint, om ikke spesielt kjapt, med en kjøretid tilnærmet $O(|V|^2)$.

I denne første implementasjonen sendte vi også med sluttpunktet som parameter inn i algoritmen. Når et knutepunkt ble erklært kjent, sjekket vi om dette knutepunktet hadde samme x og y-verdier som sluttpunktet – hvis det var tilfelle ble algoritmen avsluttet.

Poenget med denne implementasjonen var å lage noe som fant korteste vei fra én til én. Det var et greit utgangspunkt for å få litt oversikt, vi fikk testet at opptegning av rute fungerte korrekt og at algoritmen fant riktig rute. Vi testet på kommuner vi kjenner godt, som Molde og Aukra – laget også en testkommune bestående av 6 noder og 5 kanter for mer effektiv debugging. Kjøretid fra én til én gjennom Oslo var 3125 ms.
Vi gjorde også et forsøk på å implementere A* algoritmen, for å se hvor mye raskere den var. A* er en variasjon av Dijkstras som søker i retning av sluttnoden for å finne neste node [14] – se også Forslag til forbedring av algoritmen. Algoritmen egner seg derimot ikke til å finne korteste vei for annet enn én til én, så vi valgte å ikke bruke så mye tid på det. Vi har ikke tatt med denne algoritmen i det endelige programmet.

Én til alle

Med Dijkstras er det å finne korteste vei fra én til én og én til alle egentlig to sider av samme sak. Har man kjørt algoritmen på hele søkeområdet til korteste vei-treet er fullstendig, er det i realiteten bare snakk om å plukke ut de interessante knutepunktene og hente ut ruteinformasjonen. Utfordringen var å gjøre algoritmen mer effektiv – en kjøretid på $O(|V|^2)$ er ikke spesielt imponerende.

Vi implementerte en utgave av algoritmen som bruker en binærheap som datatype i stedet for vektoren – binærheapen har en gjennomsnittlig kjøretid på O(log N) for insert og deleteMin. For å takle oppdateringer av dist på knutepunkt som allerede ligger i heapen, brukes en percolateUp-funksjon for å flytte knutepunktene oppover til riktig plass i heapen – denne metoden gir update en kjøretid på O(log N) [5]. Totalt sett reduseres kjøretiden på Dijkstras fra $O(|V|^2)$ til $O(|E| \log |V|)$ ved bruk av binærheapen – en svært merkbar forbedring. Kjøretid på én til én gjennom Oslo ble nå 141 ms. Vi har også sett på noen andre datastrukturer, se Forslag til forbedring av algoritmen.

Sluttpunktene ble behandlet på to forskjellige måter. Den ene muligheten var å sende inn kun startpunktet til algoritmen – denne ble så kjørt på hele søkeområdet, som oftest en kommune eller et fylke, til alle knutepunkter var erklært kjent. Dette fungerer svært bra når man skal finne relativt mange sluttpunkt innen f.ex. et fylke – så lenge søkeområdet er avgrenset og ikke for stort, er det ikke noe problem å la algoritmen gå til den er ”ferdig.”

Den andre muligheten var å sende inn en hashmap med sluttpunkter, og sjekke hvert knutepunkt som ble satt som kjent opp mot hashmap’en. Sluttpunktene blir fjernet ett og ett, og algoritmen avsluttes når hashmap’en med sluttpunkt er tomt. Denne varianten passer best hvis sluttpunktene ligger svært nærme hverandre, søkeområdet er stort, og/eller når det er snakk om et lite antall sluttpunkter.

I begge tilfeller plukkes rutene ut etter at algoritmen har kjørt ved å ”loope” bakover fra hvert sluttpunkt til startpunktet.

Alle til alle

Alle til alle (eller mange til mange) løses ved å ha en for-løkke som looper gjennom alle stoppesteder. ”Nåværende” stoppested settes som start, og alle stoppesteder etter dette legges i en hashmap.

Algoritmen finner korteste vei fra start til alle disse stoppestedene, og rutene tas vare på i det enkelte stoppestedet.

Her har vi gjort en forenkling, og sagt at korteste vei fra A til B er den samme som korteste vei fra B til A. Dette stemmer ikke nødvendigvis i virkeligheten, hovedsakelig på grunn av enveiskjøringer og ulovlige svinger/avkjørsler. Vi har derimot antatt at forskjell i veilengde/kjøretid pga dette vil være svært liten – slike ting vil dessuten forekomme stort sett i byer, og ettersom prosjektet ikke primært dreier seg om bykjøring har vi valgt å sette dette litt til siden. Fordelen er at det halverer antall ruter som må beregnes, ulempen er at optimalitet ikke kan garanteres 100%. Dette er allikevel en smal sak å forandre – ved å ta hensyn til kjøreretning på kantene i algoritmen (informasjonen ligger i kantobjektene), samt å kjøre algoritmen fra hvert sluttpunkt til alle andre sluttpunkt vil "problemet" være løst. En økning i kjøretid må beregnes ved mange adresser, ettersom ruteantallet dobles med denne løsningen.

Når algoritmen kjøres flere ganger etter hverandre, må informasjon om dist, path og known nullstilles i knutepunktene for at algoritmen skal fungere. Vi prøvde først å legge til alle besøkte knutepunkter i en LinkedList, for så å gå igjennom denne og nullstille all informasjon i de besøkte knutepunktene. LinkedList ble foretrukket over for eksempel ArrayList på grunn av muligheten til innsetting av nye knutepunkter i konstant tid, men det viste seg at denne løsningen førte til svært lang kjøretid på algoritmen ettersom opprettelse av listenoder til å putte knutepunktene i krevde mye tid. Vi brukte Oslo som testkommune på dette, og algoritmen brukte nærmere 5 minutter på å finne ruter fra 100 til 100 adresser med denne løsningen:

```
13.mai.2005 12:44:44
kartdata.ruteberegning.RuteBeregningAlgoritme run
INFO: Tid brukt på beregning av kjørerute mellom 100 steder:
289326ms
```

Logg

I stedet innførte vi variabelen iterasjon i knutepunktklassen for å holde orden på hvilken iterasjon av algoritmen som sist endret informasjonen for dette knutepunktet. Iterasjonsnummeret er identisk med telleren i for-løkken algoritmen kjøres fra, og sendes med som et parameter hver gang algoritmen startes med en ny startnode. Når en ny node hentes ut, sammenlignes iterasjonsnummeret i noden med iterasjonsnummeret til algoritmen – er de forskjellige, ble noden sist endret av en tidligere iterasjon av algoritmen, og informasjonen må nullstilles før algoritmen kan fortsette sammenligning.

```
w = k.getNeste(v);

if (w.iterasjon != iterasjon){
      w.dist = RuteBeregningAlgoritme.UKJENT_AVSTAND;
   w.path = null;
   w.known = false;
   w.iterasjon = iterasjon;
}
```

Kildekode

På denne måten slipper vi unna med å oppdatere/nullstille nodene kun når det er nødvendig, og det viste seg å være svært tidsbesparende. Med denne lille endringen gikk tiden på 100 til 100 adresser i Oslo ned fra ca 5 minutter til ca 8 sekund.

```
13.mai.2005 13:26:16
kartdata.ruteberegning.RuteBeregningAlgoritme run
INFO: Tid brukt på beregning av kjørerute mellom 100 steder:
7828ms
```

Logg

Behandling av sluttpunktene ble også her gjort på to måter – vi har brukt en kombinasjon av å kjøre algoritmen til den stopper, og å stoppe algoritmen når alle sluttpunkter er funnet. Tanken var at dersom man skal finne så mange som 800 adresser vil det være greit å kjøre algoritmen på hele søkeområdet. Men med de antagelsene vi har gjort (korteste vei AB = korteste vei BA), vil antallet ruter som skal finnes minske ved hver gjennomkjøring av algoritmen – første iterasjon vil finne 1 til 799, andre vil finne 1 til 798 osv. Derfor var tanken at når antallet sluttpunkter kommer under et visst nivå, vil det være noe tid å spare på å begynne å sjekke sluttpunktene ut av hashmap'en og stoppe algoritmen når alle er funnet.

```
v.known = true;

// hvis få noder igjen stopper algoritmen når alle
sluttpunkter er
// funnet for å spare tid
// FIXME: har dette noen innvirkning? Når må vi stoppe den?
100, 50,
// 20 noder igjen?

  if (v instanceof Stoppested)
  {
     if (antall < 90)
     {
       sluttpunkt.remove(v); //fjerner et sluttpunkt fra
hashmap'en

       if(sluttpunkt.isEmpty())
       {
        //funnet alle sluttpunkt - avbryt
        break;
        }
     }
  }
```

Kildekode

Vi har ikke testet dette særlig grundig, og for at dette skal ha noen effekt bør det sannsynligvis testes en del på grensen for når man begynner å fjerne sluttpunkt. Det er heller ikke sikkert at det vil ha så mye å si – det vil sannsynligvis komme veldig an på søkeområdet og plasseringen av adressene. Vil tippe at det i visse tilfeller kan ha noe effekt, i andre vil det kanskje ikke merkes i det hele tatt.

Som nevnt tidligere ble hver rute først tatt vare på i en LinkedList, som så ble lagt til i det enkelte knutepunktet. Dette fungerer, men etter å ha forbedret kjøretiden så drastisk ved å ikke bruke LinkedList til å nullstille noder begynte vi å bli noe "mistenksomme" til hele datastrukturen, og gjorde et forsøk på å erstatte den med en ArrayList. Dette førte til nok en forbedring i kjøretid – fra ca 8 sekund til ca 6 sekund på 100 til 100 i Oslo. Flere testresultater viste at kjøretiden gjennomsnittlig forbedret seg med 20% ved bruk av ArrayList i stedet for LinkedList. Vi har derfor lært at selv om innsettingstiden er angitt som konstant, slik som i LinkedList [5], kan den konstante tiden fremdeles variere betydelig når man kommer opp i de antallene objekter vi har operert med. LinkedList er definitivt et tilfelle hvor dette slår inn, sett i forhold til ArrayList, som i praksis har vist seg å være raskere.

> The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.
>
> [49]

Algoritme med rydding

I tillegg til den "vanlige" algoritmen med binærheap, lagde vi en versjon som rydder i grafen før algoritmen kjøres. Algoritmen er nøyaktig den samme som før, men etter at stoppestedene er knyttet inn i grafen går vi igjennom hele grafen og fjerner alle løvnoder – det vil si noder som kun har én kant som knytter dem til grafen. Det sjekkes at stoppestedene ikke er knyttet til disse kantene. Denne oppryddingen gjør at antall noder i grafen reduseres med ca 30%, og går ikke utover optimaliteten til rutene. Kjøretiden reduseres tilsvarende. I denne oppryddingsfunksjonen har vi også vurdert å fjerne noder som kun har to kanter – dette vil lage lengre veistykker, og ytterligere redusere antall noder algoritmen må søke gjennom. Når vi kjører algoritmen med denne oppryddingen på 100 til 100 adresser i Oslo, får vi følgende resultater:

```
31.mai.2005 10:49:48
kartdata.ruteberegning.RuteBeregningAlgoritme
fjernUviktigeNoder
INFO: Knutepunkt for rydding i kommune: Oslo (0301)
var: 33175 na: 22806 endring: 10369 (31% reduksjon)
31.mai.2005 10:49:52
kartdata.ruteberegning.RuteBeregningAlgoritme run
INFO: Tid brukt pa beregning av kjorerute mellom 100 steder:
4219ms
```

Logg

Forbedringen i kjøretid fra 6 sekunder til 4.2 sekunder er ca 30% - med andre ord proporsjonalt med reduksjon i antall noder.

Testområder

Oslo kommune – 33 175 knutepunkt, 40 545 kanter
Rogaland fylke – 27 kommuner, 72 977 knutepunkt, 81 240 kanter
Hedmark og Oppland fylke – 48 kommuner, 178 202 knutepunkt, 192 921 kanter

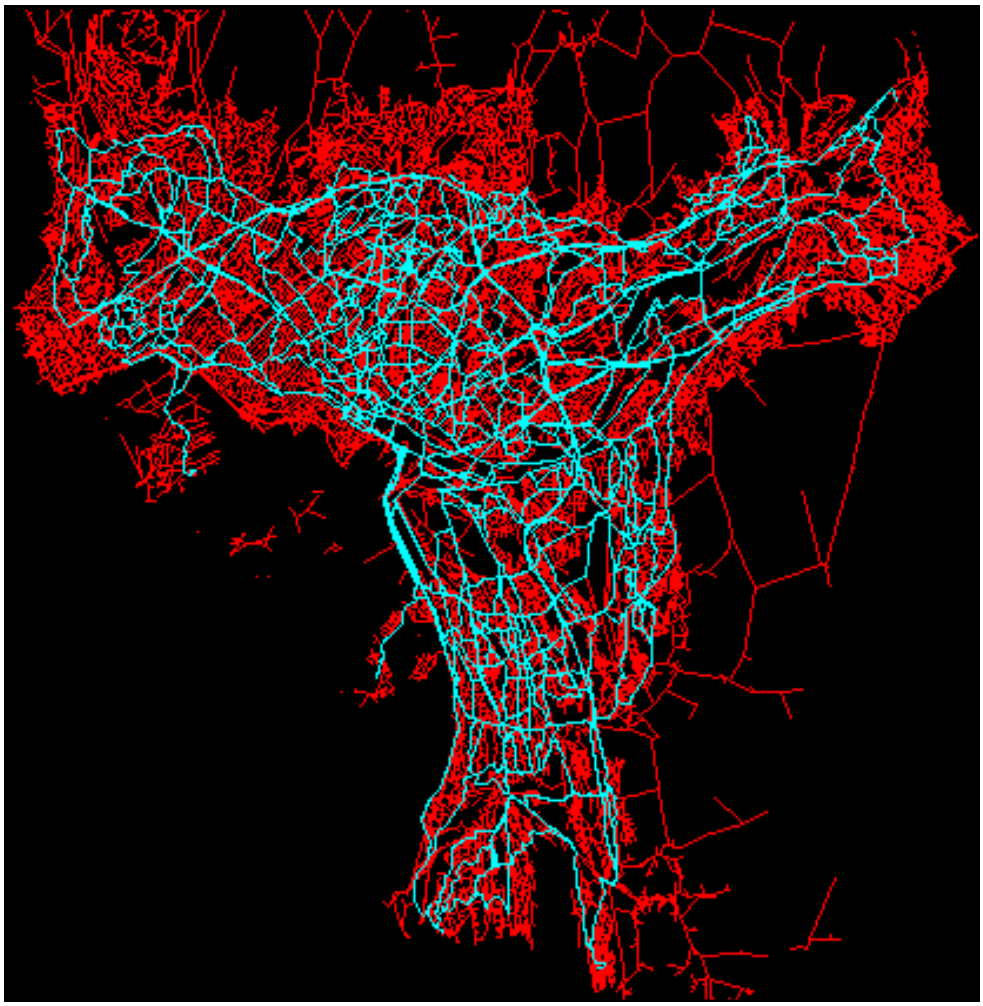
Resultater

De fleste "små" testene ble gjennomført på skolens maskiner med 512 mb RAM – for tester på Oppland og Hedmark måtte vi ty til Anders' maskin med 1 gb RAM. Etter hvert har vi testet mer og mer med 1 gb RAM, blant annet for å se hvor stor innvirkning RAM har på kjøretiden til også de "små" tilfellene.

Testadresser ble gjerne plukket ut på gatenummer i en kommune – i Oslo har vi for eksempel brukt gatenummer 15. Dette gir oss tilfeldig valgte adresser med god spredning innen en kommune.
100 til 100 adresser gjennom Oslo
Dijkstra med binærheap, nullstilling av noder vha LinkedList: 4.8 min
Dijkstra med binærheap, nullstilling av noder uten LinkedList, ruter lagret i LinkedList: ca 8 s
Dijkstra med binærheap, nullstilling av noder uten LinkedList, ruter lagret i ArrayList: ca 6 s

100 til 100 adresser gjennom Oslo – 1 gb RAM
Dijkstra med binærheap, nullstilling av noder uten LinkedList, ruter lagret i ArrayList: 5,8 s
Dijkstra med binærheap, ruter lagret i ArrayList, med rydding i grafen: 4,2 sekund

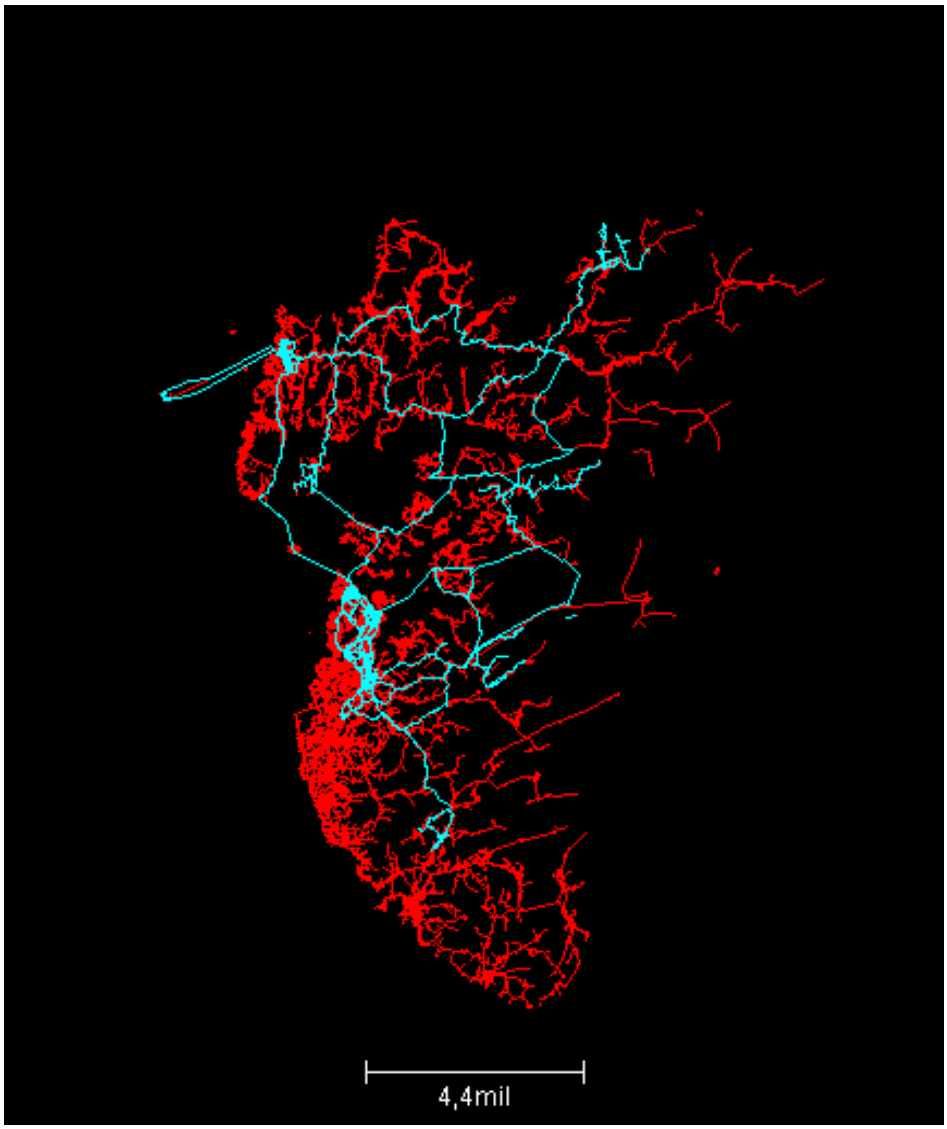
Opptegning av ruter – 100 til 100 adresser i Oslo



200 til 200 adresser i Oslo – 1 gb RAM
Dijkstra med binærheap, ruter lagret i LinkedList: 12.9 sekunder
Dijkstra med binærheap, ruter lagret i ArrayList: 11.5 sekunder

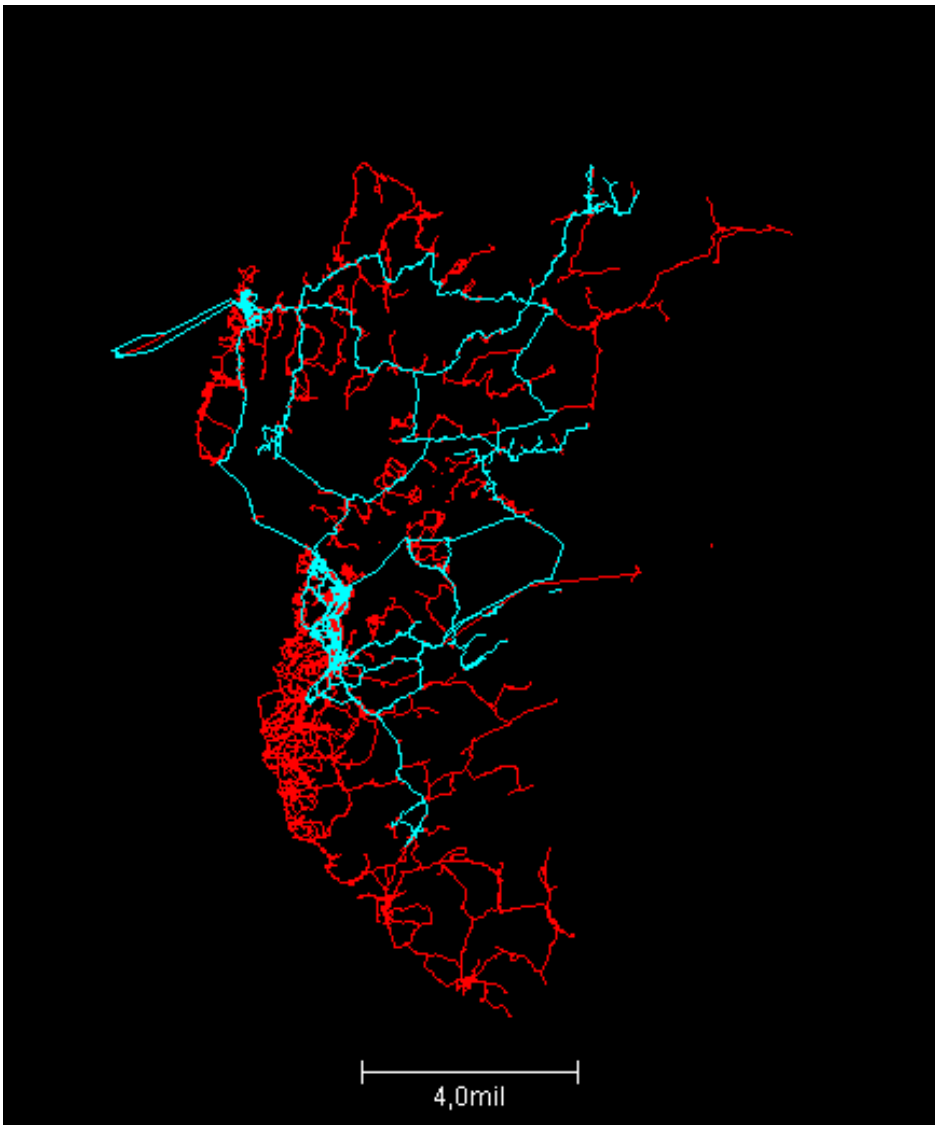
200 til 200 adresser i Rogaland – 1 gb RAM
Dijkstra med binærheap, ruter lagret i ArrayList, uten opprydding: 19.5 sekunder
Dijkstra med binærheap, ruter lagret i ArrayList, med opprydding: 11.4 sekunder

Opptegning av ruter – 200 til 200 i Rogaland uten opprydding av grafen



Her sees det tydelig hvor stor effekt det kan ha å kjøre rydding i grafen før algoritmen kjøres – forskjellen mellom disse to grafene er enorm. Innsparingen i kjøretid er også svært høy, fra 19.5 sekunder uten opprydding, til 11.4 sekunder med opprydding. Vi antar det er til dels svært mye mer å hente på å kjøre opprydding av grafen på "landet" i forhold til i bykommunene, ettersom det sannsynligvis er mye færre løvnoder i en by. Som vi ser blir veinettverket i Rogaland betydelig forenklet.

Opptegning av ruter – 200 til 200 i Rogaland, med opprydding av grafen



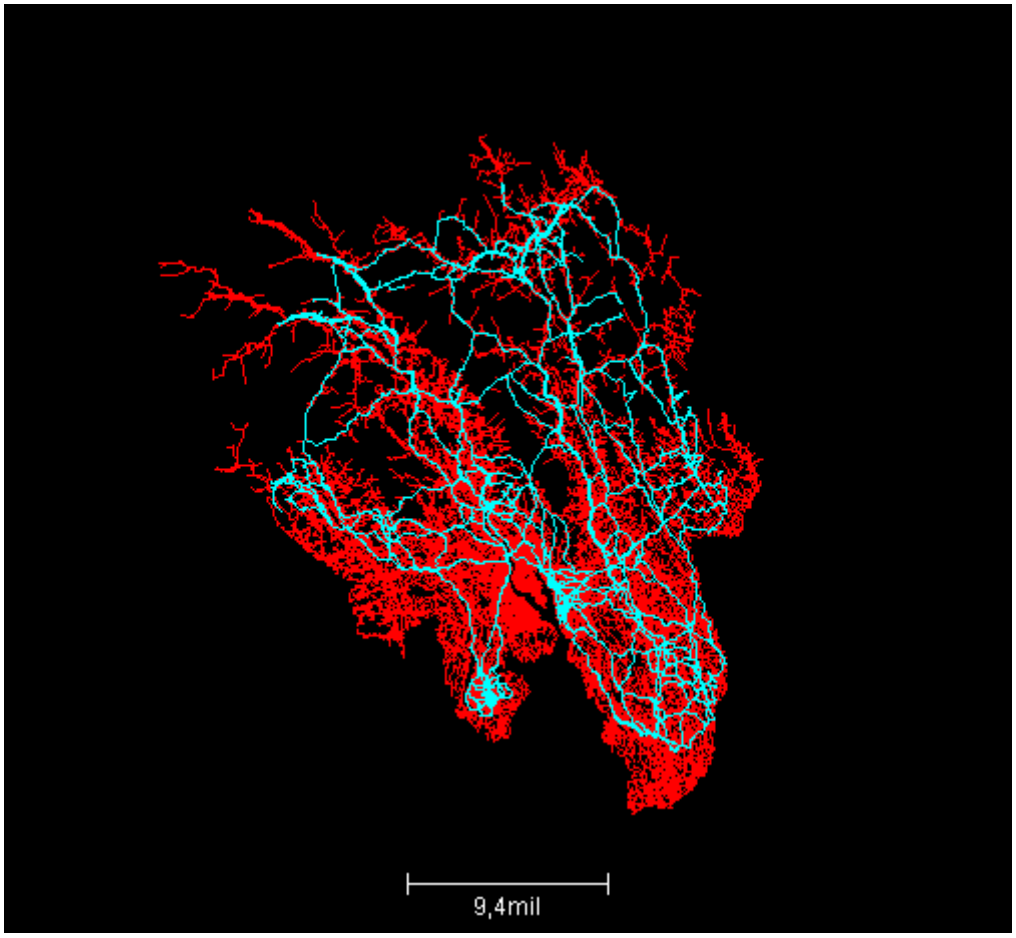
400 til 400 adresser i Oslo – 1 gb RAM
Dijkstra med binærheap, ruter lagret i LinkedList: 30.9 sekunder
Dijkstra med binærheap, ruter lagret i ArrayList: 25.5 sekunder

400 til 400 adresser i Oppland og Hedmark – 1 gb RAM
Dijkstra med binærheap, ruter lagret i ArrayList: 122 sekunder, ca 2 minutter

787 til 787 adresser i Oppland og Hedmark – 4 gb RAM, 3.2ghz p4
Dijkstra med binærheap, ruter lagret i ArrayList, med opprydding: 103 sekunder, ca 1.7 min

843 til 843 adresser i Oppland og Hedmark – 4 gb RAM, 3.2ghz p4
Dijkstra med binærheap, ruter lagret i ArrayList, uten opprydding: 210 sekunder, ca 3.5 min

Opptegning av ruter – 400 til 400 adresser i Oppland og Hedmark



Forslag til forbedringer av algoritmen

Bruke en mer effektiv datastruktur

Dijkstras er en grådig algoritme, og kjøretiden avhenger sterkt av hvordan man velger å håndtere nodene – utvelgelse av neste node som skal behandles er algoritmens flaskehals. Ved bruk av f.ex. et array med noder som scannes for å finne noden med kortest avstand til startnoden, vil algoritmen ha en kjøretid på $O(|V|^2)$. Dette forbedres til $O(|E| \log |V|)$ ved bruk av en binærheap, som i vår implementasjon [5].

Binærheapen som brukes nå er statisk – størrelsen bestemmes ved opprettelse, og heapen er derfor nødt til å være like stor som worst case antall knutepunkt som kan bli lagt inn i den. Det betyr at heapen må være like stor som antall knutepunkt i søkeområdet – i Rogaland fylkes tilfelle dreier det seg om 70 000 noder. Dette er naturlig nok plasskrevende, og bør kunne forbedres. Et forslag er å se på hvor mange noder som gjennomsnittlig er i heapen og gi heapen en størrelse ut fra dette – sannsynligheten for at alle 70 000 noder ligger i heapen samtidig er ikke-eksisterende, ettersom dette ville bety at alle noder var direkte knyttet til startnoden. Det bør derfor være mulig å beregne en mer fornuftig størrelse på heapen, med den risikoen at heapen i verste tilfelle kan bli full. Skjer det, må en

ny heap opprettes, alle data fra gammel heap kopieres over, og ny buildHeap() kjøres – en svært tidkrevende operasjon.

En bedre løsning kan være å lage en dynamisk heap, hvor de første nivåene av heapen opprettes, deretter blir ett og ett nivå av heapen tilordnet om gangen etter behov [17]. Dette vil sannsynligvis være en god løsning for vår algoritme, ettersom det aldri kjøres buildHeap(), kun insert(). Med denne løsningen vil plassbruken i heapen i verste tilfelle være på 25%, i motsetning til en statisk heap, som har en worst case plassbruk på (1/størrelse) * 100%.

En annen mulig datastruktur er en Fibonacci heap, som gir $O(|E| + |V| \log |V|)$ i kjøretid. Denne heapen har gode teoretiske kjøretider, men også en god del overhead – det er dermed uklart om Fibonacci heapen vil være bedre i praksis enn en binærheap [5], [7]. Testresultater fra Zhan og Noon [6] tyder på at Dijkstra med Fibonacci heap gjør det jevnt over dårligere enn Dijkstra med k-array heap, som har samme kjøretid som Dijkstra med binærheap, på real road networks. Det er allikevel en mulighet for at en Fibonacci heap vil passe bedre til vår type graf enn binærheapen, men dette må testes.

Ifølge flere artikler skal Dijkstras implementert med double buckets som datastruktur være svært effektivt [2], [3], [4], [6]. Algoritmen har en kjøretid på $O(|E| + |V| (\beta + C/\beta))$, [6] med
C, $\max(||e||), \forall e \in E$ – største kantlengde i grafen, og
β, konstant som brukes til å bestemme antall buckets, e.g. høyeste potens av $2 < \sqrt{C}$ [3].
Vi har gjort et forsøk på å implementere dette etter modell fra [3]. Har imidlertid ikke fått denne til å fungere, vil anta at feilen ligger i formlene som regner ut størrelse på/posisjon i buckets (?).

Legge til penalties

Foreløbig er algoritmen lite "intelligent." En idé vi har hatt, men ikke har fått implementert, er å legge til penalties for å styre algoritmen i riktig retning.

For å få et mest mulig fleksibelt system uten å begrense hastigheten på algoritmene, hadde vi planer om å lage ett sett til med adapter. Disse adapterne er tenkt å implementere en metode for å beregne verdien på en kant, ved at den aktuelle kanten sendes inn. Deretter vil hvert adapter evaluere de interne dataene og returnere et resultat. Kanter som ikke er så interessante vil få returnert en høyere vektverdi enn kanter som er foretrukket. For å unngå kanter med negativ vekt, vil det sannsynligvis i praksis være mest aktuelt å kun øke vektverdien, ikke minske den. Summen av alle disse adapterne's resultater vil ende opp som kantens vekt.

Dette kan bl.a. benyttes for å få algoritmen til å prioritere gode veier over dårlige - det vil være rimelig å tro at det er raskere å kjøre et lengre stykke på en europaveg enn det er å kjøre en kortere distanse på dårlige grusveier. På denne måten kan man også markere veier som "ukjørbare" ved å gi dem et høyt antall penalties – dermed vil algoritmen unngå disse veiene så langt som mulig, og holde seg til andre tilsynelatende "kortere" veier. Teoretisk sett kan man med denne metoden få algoritmen til å unngå nesten hva som helst – f.ex. bykjøring, ferger osv.

En annen tanke har vært å gi ruter penalties for antall avkjørsler og lignende – dette vil kanskje være særlig nyttig ved rutefinning gjennom store byer, hvor det vil være fordelaktig å kjøre lengst mulig på hovedveier og ha så få avkjøringer som mulig. Det vil også i teorien gjøre det mulig å prioritere lange veglenker, for å unngå flere kryss enn nødvendig ved å gi høyere vekting til korte veger. I disse tilfellene kan det tenkes å være fornuftig å sjekke om knutepunktene i endene virkelig er kryss, fordi det også finnes knutepunkt ved f.eks. bytte av hastighet, overgang mellom vegkvalitet osv.

Tilsvarende kan det hindres at vi velger ruter hvor lastbilene ikke kan kjøre på grunn av vektrestriksjoner, høyde m.m.

Forskjellige typer avkjørsler vil også ha innvirkning på kjøretid – eksempelvis vil avkjørsler fra åtte-felts veier til to-felts veier ofte ta mer tid enn en avkjørsel fra én fire-felts vei til en annen fire-felts vei. Det kan selvfølgelig diskuteres hvor relevant dette er i Norge, men ved bykjøring bør i det minste kunne tas hensyn til enveiskjøringer og ulovlige svinger [8].
Dette kan gjøres ved å implementere en metode i kant som sjekker kjøreretninger – ideen der er omtrent den samme som kant-klassen sin getNeste()-funksjon. Når man skal traversere en kant, er man i et knutepunkt. Kanten har to knutepunkt, og for å finne ut hvilken som er den neste, må man finne det andre knutepunktet. Tilsvarende metode kan brukes for å finne ut hvilken kjøreretning man kommer inn en kant (veg) fra. Når dette er avgjort, vil man kunne bruke verdiene til kjøreretning, ifbtretning (innkjøringforbudt) og sperrekjøreretning for å avgjøre om dette er en veg som er kjørbar i vår retning. Hvis den ikke er det, må algoritmen prøve å finne en annen mulig rute. Denne funksjonen vil da kunne brukes av algoritmene når de skal velge ut neste kant.

To-/flere-lags søk

Ettersom grafen er svært stor, kan det være en idé å dele den opp i flere lag – for deretter å finne korteste vei mellom grensenodene i hvert lag. Dermed sparer man tid ved å "finsøke" kun når man nærmer seg målet [9], [10], [11]. En idé kan være å la europaveier og evt. riksveier være et eget lag, deretter fylkesveier og kommunale veier. Ved å søke seg "oppover" i lagene bør man kunne spare en del tid – spørsmålet er om dette vil gå på bekostning av optimalitet.

En annen mulighet er å lage en algoritme etter modell av [9], med en hierarkisk graf inndelt i subgrafer. På denne måten kan man forhåndsberegne korteste vei mellom alle kantnoder i de forskjellige subgrafene, slik at det er en relativt enkel operasjon å finne korteste vei. Dersom sluttnoden ikke befinner seg i den aktuelle subgrafen, plukker man den forhåndsberegnede korteste veien til neste subgraf. Dermed unngås "finsøking" før man kommer til den subgrafen som inneholder den aktuelle sluttnoden. Med en god grafinndeling [12] skal dette være svært effektivt, og samtidig fortsatt gi optimale løsninger.

Benytte eksisterende informasjon

Ved hver gjennomkjøring av algoritmen lagrer vi korteste vei fra alle besøkte noder til startnoden. Skal man finne korteste vei fra 1000 til 1000 adresser, vil det sannsynligvis være mye å hente på å ta vare på denne informasjonen på en eller annen måte – slik at man kutter ned på kjøretiden til algoritmen ved å benytte eksisterende informasjon om korteste vei mellom noder. Skal algoritmen finne korteste vei mellom A og F og besøker node C på veien, vil man kanskje allerede vite korteste vei fra C til F – dermed kan algoritmen avsluttes. Vi har ikke så mange gode ideer til nøyaktig hvordan dette skal gjennomføres, men det føles noe bortkastet å regne ut store deler av ruteinformasjonen på nytt for hver gjennomkjøring av algoritmen.

På den annen side – dersom noe slikt gjennomføres, kunne man i teorien regnet ut korteste vei fra alle noder til alle andre noder i nettverket på forhånd, og funnet korteste vei i konstant tid [1] - problemet blir lagringsplass ettersom Rogaland fylke alene har nærmere 70 000 noder. Spørsmålet blir om noe av dette i det hele tatt er gjennomførbart – vi har ikke så mange gode ideer til hvordan det i tilfelle skulle gjøres.

A* algoritmen

A* algoritmen er i praksis bare en variant av Dijkstras – men den bruker gjerne euklidsk avstand i tillegg til kantkostnad for å velge neste node: nodens avstand fra startnoden legges til den euklidske avstanden fra noden til sluttnoden, slik at avstand/kostnadsvariabelen som nodene sorteres etter gir et slags estimat på avstanden fra startnoden til sluttnoden. Algoritmen søker med andre ord i retning av sluttnoden, i motsetning til Dijkstra, som søker i alle retninger og alltid velger noden som er nærmest startnoden som neste node.

A* algoritmen er svært populær i løsning av korteste vei-problemer [13], [14], [15], men egner seg best til å beregne korteste vei mellom to punkter i motsetning til en til alle/alle til alle. Besparelsen ved bruk av A* på en til en i forhold til Dijkstra vil nok "spises opp" dersom algoritmen må kjøres fra en til en 800 ganger – de samme rutene fra 1 til 800 finnes ved én gjennomkjøring av Dijkstras. Konklusjonen blir at A* er god til sitt bruk – men egner seg ikke til vårt.

Visningsverktøy

Det ble tidlig i prosjektet klart at de enorme mengdene data vi stod ovenfor i stor grad måtte debugges visuelt. I praksis er det ikke et gjennomførbart alternativ å debugge kode linje for linje når man snakker om millioner av objekter i koden. Selv om våre utviklingsverktøy støtter såkalte "Conditional breakpoints" (Steder i koden hvor debuggeren stopper hvis ett krav er oppfyllt) er det i de fleste tilfeller raskere, sikrere og mer praktisk å visualisere datastrukturene. Vi hadde to alternativer i så måte. Det første alternativet var å bruke det kommersielle visningsverktøyet MapInfo. Dette verktøyet var nevnt i prosjektbeskrivelsen, og ble levert til oss omtrent samtidig med kartdataene. Det var i utgangspunktet meningen å bruke MapInfo som visningsverktøy for både kartet og våre genererte ruter.

Alternativet til dette var at vi måtte ha laget enda et programtrinn, hvor våre interne strukturer igjen ble utkjørt i et format som MapInfo har mulighet til å vise. I praksis var ikke dette noe godt alternativ, spesielt ikke siden vi allerede hadde erfart med Visual Basic .Net prototypen vår at å tegne opp koordinatene var en relativt triviell jobb. Vi hadde også problemer med å bruke MapInfo til å vise Sosi-formatet. Det fulgte med en omfattende manual til MapInfo, og det er antagelig mulig å gjøre dette på et vis. Vi valgte å ikke bruke noe mer ressurser på den delen av arbeidet og begynte på en løsning som var mer praktisk for vår bruk.

Etter at den nevnte prototypen var laget, konkluderte vi med at vi ville gjøre resten av arbeidet i Java. Det ble derfor opprettet et nytt prosjekt med et grensesnitt basert på Java Swing klassene. Grensesnittet har gradvis fått mer funksjonalitet etterhvert som behovet har meldt seg. Det første trinnet var tilnærmet en ren oversetting av Visual Basic .Net prototypen til en tilsvarende i Java. Det ble allerede på dette tidspunktet tegnet direkte på et bilde i minnet. Etterhvert som behovet for å visualisere ytterligere informasjon økte, meldte det seg problemer med å få vist dataene på fornuftig vis samtidig. Dette problemet ble da løst ved at bildet ble gjort transparent, og hver detaljtype fikk sitt eget lag.

Etter at disse var tegnet uavhengig, ble de lagt oppå hverandre til ett bilde av det delsystemet som håndterer opptegningen. Dette bildet ble så vist i grensesnittet etterpå. Ved å manipulere rekkefølgen på disse lagene, og legge til muligheter for å slå de forskjellige detaljene av og på, hadde vi på en meget enkel måte fått et relativt avansert system for å visualisere de nødvendige data. De forskjellige opptegningslagene får koordinatene til alle punktene de skal tegne ferdig konvertert og skalert ned, slik at det eneste de selv behøver å gjøre er mer eller mindre å tegne sin detaljtype på en fornuftig måte.

Det var i prosjektbeskrivelsen nevnt at systemet på sikt er tenkt integrert i et beslutningsstøttesystem. Siden vi ikke vet noe mer om hvordan dette systemet er tenkt har vi ikke kunnet ta direkte hensyn til spesielle krav fra dette systemet. Vårt eget grensesnitt er i noe varierende grad koblet til de bakenforliggende delene. Grensesnittet fungerer i praksis som en kontroller for fremdriften i beregningen, men for å få en fungerende tilknytning er det hovedsaklig to steder man må kobles til. For å få lest inn kartdata har vi definert et grensesnitt IVegData. I vår implementasjon er det kun en implementasjon av dette grensesnittet, og det er mot et sett med Sosi-filer. I all hovedsak gir dette grensesnittet en liste over tilgjengelige kommuner, og man kan så spørre etter en gitt type data (adresser/geometri/alle), og en liste med kommuner. Disse vil så bli lest inn og koblet sammen til den grafen resten av systemet tar utgangspunkt i.

Kontrolleren vil i vårt tilfelle etter en slik innlesning sørge for at de innleste geometridataene blir visualisert. Dette gjøres via Kart-klassen. Den tilbyr metoder for å tegne opp grafen som et bilde i ønskede størrelser. Det er denne klassen som etter innlesning kontrollerer resten av beregningene. Kart har metoder for å legge inn det vi har kalt Stoppested i grafen. Dette er start- og slutt-adresser for ruteberegningene. Disse metodene tar imot en liste med ønskede stoppesteder, og kart-klassen forsøker etter beste evne å knytte disse adressene inn i grafen, og beregne avstander på oppkjørsler osv. Denne tilknytningen skjer indirekte etter at det er sendt en forespørsel om å finne kjøreruter mellom to eller flere bygninger. Samtidig med dette sendes det også inn den aktuelle algoritmen som skal gjøre selve beregningen. Kart har en liste med tilgjengelige algoritmer og grensesnittet lar i vårt tilfelle brukeren velge en av disse algoritmene, kart mottar så den valgte algoritmen sammen med de aktuelle stedene.

Meldinger mellom de forskjellige delene av programmet sendes ved hjelp av en abonnementsordning. Dette mønsteret, ofte kalt "Observer (Publish - Subscribe)" er et av flere meget vanlige mønster for organisering av programvare. Klassene som trenger å vite om en gitt hendelse registrerer seg som interessenter av en gitt type melding. De må da implementere et gitt grensesnitt for disse hendelsesmeldingene. Hver gang hendelsen skjer, tar klassene selv ansvar for å gjøre nødvendige endringer i sitt eget grensesnitt, eller interne struktur. På den måten er grensesnittet relativt oppdelt, uten for høy grad av kobling mellom de forskjellige delene. Denne måten å organisere det på gjør det relativt enkelt å legge til nye moduler.

Bak kart-klassen er det igjen 3 uavhengige systemer. Disse er alle modellert som adapter til programmet - det vil si at de kan plugges inn i programmet mer eller mindre uten videre. Det ene systemet er som tidligere nevnt de forskjellige opptegningslagene. Disse har et felles grensesnitt med alle metoder programmet vårt behøver for å kommunisere med dem. Fordelen er at disse kan skrives uavhengig av resten av systemet, og plugges rett inn. Vårt program har gjort det slik at alle de 3 typene adapter er definert i kart.xml. Der er det seksjoner for hver av de 3 typene, og man lister opp klassene som skal lastes inn under oppstarten. De klassene som håndterer utskrift av resultat er organisert på samme måte med felles grensesnitt. Algoritmene er organisert litt annerledes, fordi det viste seg at det var praktisk med felles metoder for disse til å sende meldinger til abonnenter om fremgang i beregningen, og ikke minst melding om at beregningen var ferdig. I tillegg kan det være interessant for algoritmene å benytte vår funksjon for å fjerne løvnoder i grafen. Denne er derfor definert i en abstrakt klasse RuteBeregningAlgoritme. De algoritmene vi har skrevet må deretter arve denne klassen, og er nødt til å implementere metoder for selve algoritmen. Denne løsningen gjør at vi kan tvinge alle algoritmene til å kjøre i tråder, uten at de behøver å vite noe om det selv. Dette er mulig å gjøre fordi den abstrakte klassen sine metoder sørger for å sende meldingene om at algoritmen er ferdig. Fordelen er da at vi kjører algoritmene uavhengig av grensesnittet i programmet - det er dermed mulig å bruke det mens algoritmen kjører, og man får vist fremdriften i beregningen

til brukeren. Dette hindrer at man tror programmet har hengt seg, samt at ventetiden ikke føles så lang.

Det fjerde systemet er tidligere nevnt, nemlig innlesning av Sosi-data. Dette har sitt eget grensesnitt, men er i stor grad organisert likt. Den største forskjellen er at vi her ikke har gjort det mulig å definere adapter i en xml-fil, hovedsaklig fordi det gir liten mening. Hensikten med innlesningen er å fylle datastrukturene våre. De er ikke dynamiske på samme måte som opptegningene, og det var dermed ikke noe stort poeng.

Logging

Programmet har innebygget logging av hendelser, feilmeldinger og beskjeder. Programmet bruker den innebygde standardklassen for logging, Logger. Denne klassen har 7 forskjellige nivå for meldingene som skal logges. Den alvorligste er SEVERE, for kritiske/uopprettelige feil. WARNING, for alvorlige feil. INFO, for informasjon fra programmet, for eksempel med tanke på debugging. CONFIG tenkt brukt til statiske konfigurasjonsmeldinger. Dette nivået er ikke brukt hos oss. FINE brukes mye til å gi informasjon om hva som skjer, men relativt grovt. Disse meldingene kommer stort sett etter at programmet har tatt avgjørende valg om fremdrift, for å kunne spore rekkefølgen ting utføres i. I tillegg til disse er det to nivåer FINER og FINEST for enda mer detaljerte beskjed om fremdrift og valg.

Vi har valgt å logge alle meldinger fra programmet i utgangspunktet. Det er flere måter å sortere ut de interessante nivåene for sluttbruker, for eksempel kan XSL brukes (http://www.w3schools.com/xml/xml_xsl.asp) hvis loggene er lagret som XML. For å gjøre loggene mer lettleste, uten å bruke XSL, har vi valgt inntil videre logge til en standard tekstfil ved hjelp av klassen SimpleFormatter. Den og XMLFormatter leveres som standard med Java. SimpleFormatter gir som navnet antyder, en enklere og mindre detaljert logg, men den er til gjengjeld enklere å lese direkte for sluttbruker.

Eksempel på en melding ved bruk av SimpleFormatter:
18.mai.2005 16:45:51 storage.fil.AVegDataFil lesKommuner
FINE: Antall kommunenavn innlest: 434

Vi får her logget tidspunkt for hendelsen. I tillegg forsøker loggeren automatisk å finne ut hvor feilen oppstod. Dette kan i visse tilfeller være noe feil på grunn av at enkelte Just In Time compilere forsøker å spare noen steg, og slår sammen funksjonskall. Dette har ikke vært noe problem for oss under utvikling og testing av programmet. Vi ser her altså at hendelsen skjedde i klassen AVegDataFil under pakken storage.fil. I tillegg har vi fått logget navnet på funksjonen hvor hendelsen oppstod, i dette tilfellet lesKommuner. På den andre linjen står det først angitt hvilket nivå feilmeldingen har, før selve meldingen skrives ut. I situasjoner hvor det logges Exceptions, vil også stacktrace skrives ut i loggen, slik at man ser hendelsesforløpet til den orginale feilen samt feilmeldingen.

Hvis det er ønskelig å endre hvilket minimumsnivå feilene som skal logges har, må man inn å gjøre endringer i koden i enten Veiviseren.java eller DbFyller.java - disse klassene har ansvaret for å sette opp loggingen, samt starte sine respektive programmer. I disse filene er det også mulig å endre hvilken type logg som skal skrives. Vi har valgt å bruke loggrotering på loggene. Programmet vil sørge for at de 7 siste kjøringene sine logger ligger lagret. Hver fil er begrenset oppad til 5mb. Disse

tingene kan endres samme sted. Nyeste loggfil er alltid fil nummer 0. Det er også mulig å logge til flere kilder samtidig hvis dette er ønskelig.

Det er separat logging for henholdsvis Veiviseren og DbFyller, med tilsvarende navn. Hvis programmet startes med konsoll, vil det stå under oppstarten hvor loggen lagres. Dette er satt til brukerkatalogen til den innloggede brukeren. Det vil si C:\Documents and Settings\bruker (Windows) eller /home/bruker (Linux). Hvis programmet startes med konsoll vil også meldinger med nivå INFO og oppover skrives direkte til konsoll i tillegg til loggfilen.

Konklusjon

Oppgaven gikk grovt sett ut på å lage en ruteberegningsapplikasjon, som på sikt skal implementeres i et beslutningsstøttesystem. Dette mener vi å ha fått til på en akseptabel måte. Applikasjonen er ikke perfekt, men skal i det store og hele fungere – det var dessuten en utfordring å måtte avslutte prosjektet fordi det ikke var mer tid igjen; vi har fortsatt massevis av forslag til forbedringer.

Målet var å lage en applikasjon som beregnet ruter mellom 800 adresser på mindre enn 100 timer – dette har vi klart til gangs, med en kjøretid på 210 sekunder på ruter mellom 843 adresser (i Oppland og Hedmark) må vår applikasjon sies å være betydelig raskere enn Geodatas løsning.

Referanser

1 Faramroze Engineer: *Fast shortest path algorithms for large road networks*

2 F. Benjamin Zhan: *Three fastest shortest path algorithms on real road networks: data structures and procedures*

3 Peter Eklund, Steve Kirkby, Simon Pollitt: *A dynamic multi-source Dijkstra's algorithm for vehicle routing*

4 Roozbeh Shad, Hamid Ebadi, Mohsen Ghods: *Evaluation of route finding methods in GIS application*

5 Mark Allen Weiss: *Data structures & algorithm analysis in Java*

6 F. Benjamin Zhan, Charles E. Noon: *Shortest path algorithms: an evaluation using real road networks*

7 Frank Schulz, Dorothea Wagner, Karsten Weihe: *Dijkstra's algorithm on-line: an empirical case study from public railroad transport*

8 Dirk Van Vliet: *Improved shortest path algorithms for transport networks*

9 Sungwon Jung, Sakti Pramanik: *An efficient path computation model for hierarchically structured topographical road maps*

10 Frank Schulz, Dorothea Wagner, Christos Zaroliagis: *Using multi-level graphs for timetable information in railway systems*

11 Patrick Lester: *Two-tiered A* pathfinding*

12 Yun-Wu Huang, Ning Jing, Elke A. Rundensteiner: *Effective graph clustering for path queries in digital map databases*

13 Patrick Lester: *Using binary heaps in A* pathfinding*

14 Jan Husby: *Fastest path problems in dynamic transportation networks*

15 http://www.wiu.edu/users/mflll/351/AStar.html *The A* variation to the shortest path problem*

16 Maurizio Bielli, Azedine Boulmakoul, Hicham Mouncif: *An efficient multimodal path computation for transportation networks*

17 Martin Fowler: *Refactoring Improving the design of existing code*

18 Jesper Bojesen: *Heap implementations and variations* http://www.diku.dk/hjemmesider/ansatte/jyrki/Course/Performance-Engineering-1998/heap_report/
19 http://www.velbon.org/cvs/VisVeg/ *CVS webgrensesnitt*
20 http://www.linux.ie/articles/tutorials/managingaccesswithcvs.php *Managing access with CVS*
21 http://www.statkart.no/standard/Sosi/html/welcome.htm *SOSI-standarden*
22 http://www.statkart.no/standard/Sosi/html_34/adr/adr.htm%20 *SOSI-standarden Databeskrivelse: Adressepunkt*
23 http://www.statkart.no/standard/Sosi/html_34/vbase/vbase.htm *SOSI-standarden Databeskrivelse: Vegnett*
24 http://www.statkart.no/standard/Sosi/html_34/abas/abas.htm *SOSI-standarden Databeskrivelse: Administrative og statistiske inndelinger*
25 http://www.statkart.no/standard/Sosi/html_32/vbas/vbas.htm *Definisjoner av standarder – vegnett*
26 http://www.javaperformancetuning.com/tips/rawtips.shtml *Java performance tuning*
27 http://tutorials.worldwidelearn.com/read/id/216/headline/Java:+Performance+Tuning+and+Memory+Management+Part+4+-+Memory+Utilization *Java: Performance tuning and memory management*
28 http://www.javaworld.com/javaworld/javatips/jw-javatip130.html *Do you know your data size? Don't pay the prize for hidden class fields*
29 http://www.caucho.com/resin-3.0/performance/jvm-tuning.xtp *JVM tuning*
30 http://www.ssb.no/kommuner/ *Statistisk sentralbyrå – kommunedata*
31 http://159.162.103.30/gos/gotogos.jsp *GeoNorge Kart og geografiske tjenester på internett*
32 http://jmechanic.sourceforge.net/ *jMechanic Eclipse Java Profiler*
33 http://junit.org/index.htm *JUnit, testing resources for extreme programming*
34 http://www.gisdevelopment.net/technology/gis/index.htm *GIS Development The geospatial resource portal*
35 http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html *How to use threads*
36 Renaud Waldura: *Ten reasons to use Eclipse* http://renaud.waldura.com/doc/java/eclipse-ten-reasons.shtml
37 http://www.appperfect.com/products/devsuite/jp.html *AppPerfect DevSuite*
38 http://www.eclipse.org/ve/ *Eclipse Visual Editor*
39 http://subversion.tigris.org/ *SubVersion*
40 https://www.cvshome.org/ *CVS – Concurrent Versions System*
41 http://statcvs.sf.net/ *StatCVS*
42 http://www.freebsd.org/projects/cvsweb.html *CVSweb*
43 http://www.postgresql.org/ *PostgreSQL*
44 Craig Larman: *Applying UML and patterns – an introduction to object-oriented analysis and design and the unified process*

Kodereferanser

45 http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=2&t=011278 *Opptegning av retningspiler*
46 Nobuo Tamemasa: *Checkbox tree* http://www.codeguru.com/java/articles/185.shtml
47 http://www.apl.jhu.edu/~hall/java/Swing-Tutorial/Swing-Tutorial-JTree.html *JTree*
48 http://forum.java.sun.com/thread.jspa?threadID=125139&messageID=1245992 *Begrense input i tekstbokser*
49 http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html *ArrayList*