# Øving 9 algoritmer og datastrukturer

## Innhold



### Oppgave

Lag et program som både kan finne veien med ALT og med Dijkstras algoritme. Programmet skal også kunne finne de 8 interessepunktene som er nærmest et sted. (Bensinstasjoner, ladestasjoner, kafeer, barer, …) Det gjøres med Dijkstras algoritme.

For å bruke ALT, må dere preprosessere kartet for å finne avstander til/fra landemerkene dere velger. Slike data lagrer dere på en egen fil, så denne prosesseringen *ikke* trenger gjøres hver gang.

Dere får kart over de nordiske landene (Openstreetmap, 2022) med koordinater på hver node. Koordinatene kan brukes til å visualisere reiseruta.

Ta tiden på begge algoritmene, og finn ut hvor mange noder hver av dem trenger å undersøke (plukke ut av prioritetskøen). Presenter reiseruten grafisk.

#### Kriterier for godkjenning

1. Programmet implementerer både ALT og Dijkstras algoritme. Det måler tidsforbruk for begge algoritmene, (å finne veien, ikke alt annet) og teller opp hvor mange noder hver algoritme trenger å plukke ut fra køen. (kan vises med skjermdump)

   Forhåpentligvis ser vi at ALT behandler færre noder – om det også går raskere vil avhenge av implementasjon og reiserute. ALT krever mer minne, det påvirker også tidsbruken.
2. Programmet kan finne de 8 interessepunktene av passende type som er nærmest et sted.

   La programmet finne 8 ladestasjoner nær "Trondheim lufthavn, Værnes". Plott disse på et kart og send inn skjermdump.

   Finn også de 8 drikkestedene som er nærmest "Trondheim torg", vis disse på et kart.

   Finn 8 spisesteder nærmest «Hemsedal», og vis på et kart.

3. Dere implementerer algoritmene selv. (Ikke ferdige opplegg for ruting.)
4. ALT og Dijkstra algoritmene finner samme vei hvis de får samme mål og start, og samme vei og kjøretid som fasit.

   (Teoretisk kan veiene være ulike, men de *må* ha nøyaktig samme lengde. Det er bare *en* avstand som er kortest, og begge algoritmer finner *korteste vei*…)
5. På kartet over Norden, finner algoritmene en hvilken som helst vei på under 15 sekunder. (Gjelder bare kompilerte språk som java/kotlin/C/C++)

   (Det burde være rikelig, mitt javaprogram trenger 1s på Oslo-Stockholm.) Dere trenger ikke regne med tiden for å lese data inn fra fil, som gjerne tar 10–30 sekunder. Ta heller ikke med preprosessering, som typisk tar flere minutter.
6. Programmet forteller hvor lang reiseruta er, i timer, minutter og sekunder. Kan vises med skjermdump.
7. Dere kan vise reiseruta grafisk. Det kan gjøres av programmet, eller med eksterne midler. Men det må være *deres egen* rute som vises. Kan vises med skjermdump.
8. Send inn reisetid og reiserute og andre data for Kårvåg–Gjemnes og Tampere–Ålesund. Husk både Dijkstra og ALT for begge turene.

## ALT-søk på kartdata

Jeg har lastet ned kartmateriale for Norden fra `http://openstreetmap.org`, og trukket ut informasjon som egner seg for denne oppgaven. Veinettet består i utgangspunktet av veldig mange punkter. Men mange av disse punktene brukes bare for å beskrive hvordan veiene går gjennom landskapet. De har jeg forenklet vekk, vi trenger bare vite hvordan kryssene henger sammen. (Hvis dere regner ut avstanden mellom to noder og får et mindre tall enn det som står i filene mine, er det fordi veien imellom har svinger og dermed blir lenger enn en rett linje gjennom landskapet.)

## Kartfiler

### Norden

Nodefila er på 214 MB. Den inneholder 7 509 994 noder. Første linje angir antall noder. Deretter er linjene formatert slik:

```
nodenr breddegrad lengdegrad
```

Feltene er skilt med tabulatortegn. Lenke: `https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/norden/noder.txt`

Utdrag av nodefila, første linje er antall noder:

```
           7509994
0 55.6345298 12.0729630
1 55.6345880 12.0722614
2 55.6346358 12.0705787
3 55.6390613 12.0686169
```

Kantfila er på 425 MB, og inneholder 16 826 594 vektede kanter. Første linje angir antall kanter. Deretter er linjene formattert slik:

```
franode  tilnode kjøretid lengde fartsgrense
```

Lenke: `https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/norden/kanter.txt`

Kjøretiden er i hundredels sekunder, lengden er i meter, fartsgrensen i km/h. Dere trenger ikke egentlig lengde og fartsgrense for å løse oppgaven, de er bare med for å vise datagrunnlaget. Kjøretiden kan brukes som kantvekt i grafen.

Vektingen er i hundredels sekunder, fordi det da går an å bruke int til kantvektene. Med over 16 mill kanter, sparer det en del plass i minnet.

Grupper med interesse for korteste vei uavhengig av fartsgrense, kan også prøve veilengde som kantvekt. Korteste vei å gå/sykle blir anderledes, fordi fartsgrenser ikke er noe tema.

Utdrag fra kantfila, første linje er antall kanter:

```
          16826594
0       1       792     44      20
1       0       792     44      20
1       2       1926    107     20
2       1       1926    107     20
```

I tillegg får dere filen: `https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/norden/interessepkt.txt`. Den er fin å bruke for å finne nodenummeret til kjente kryss som «Prinsenkrysset», eller nodenummeret til stedsnavn som «Trondheim» og «Tampere». Denne fila er bare 5,9 MB. Søk i en editor, eller bruk `grep`-kommandoen for å finne nodenummeret til kjente steder.

Formatet er slik:

```
nodenr kode "Navn på stedet"
```

Nodenr er samme nr. som i nodefila. Navnet er tekst i unicode, utf-8.

Koden angir hva slags interessepunkt dette er:

| Bit | Betyr | Eksempler |
|---|---|---|
| 1 | Stedsnavn | Trondheim, Moholt, … |
| 2 | Bensinstasjon | Shell Herlev |
| 4 | Ladestasjon | Ionity Klett |
| 8 | Spisested | Restauranter, kafeer, puber |
| 16 | Drikkested | Barer, puber, nattklubber |
| 32 | Overnattingssted | Hoteller, moteller, gjestehus |

Et interessepunkt kan være i flere kategorier, da er koden en sum av flere slike verdier. Et sted som serverer både mat og drikke, vil ha kode 8+16=24. Hvis stedet er hotell i tillegg, legger vi på 32 og får 56.
For å sjekke om en kode «er spisested» kan vi teste bit 8 slik:

```
if (kode & 8 == 8) ...
```

Den testen vil slå til for 8 som bare er spisested, 24 som er kombinert spise- og drikkested, og 40 som er hotell med restaurant. Men den vil ikke slå til for 48, som er hotell med bar, men uten restaurant.
Tilsvarende kode for å sjekke om en node er bensinstasjon:

```
if (kode & 2 == 2) ...
```

For å teste typen interessepunkt, bruk & (binary and) og koden for den typen interessepunkt dere ser etter. Binary and gjør det mulig å plukke ut en bestemt type punkt, uavhengig av andre typer som kan være knyttet til punktet.

### Nedlastingsproblemer

Å laste ned noen få hundre MB bør gå fort og greit på våre raske nettverk. Men det er alltid noen som har problemer med å laste ned disse filene, fordi de klikker på dem og prøver å vise dem i nettleseren. Nettlesere liker ikke å tegne 7–16 millioner linjer med tekst! Her er noen andre metoder:

- For linux og mac, bruk kommandoer som denne:

  ```
  wget https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/norden/noder.txt
  ```

- Andre: høyreklikk på lenka, velg «Lagre lenke som…»

Vi har mange vanskelige oppgaver, men en dataingeniør bør *ikke* ha problemer med å laste ned noen hundre MB.

### Island

Islandkartet er mye mindre. (109 911 noder) Det går mye fortere å lese inn, det er praktisk når dere utvikler/debugger programmet. Lenker:

`https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/island/noder.txt`

`https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/island/kanter.txt`
`https://www.idi.ntnu.no/emner/idatt2101/Astjerne/opg/island/interessepkt.txt`

## ALT og preprosessering

For å bruke ALT-algoritmen, må dere velge ut noen landemerker og beregne avstand fra disse nodene til alle andre noder i grafen. Og omvendt, avstand fra alle noder til landemerkene. ALT-algoritmen trenger disse avstandene.

Dere bestemmer selv hvor mange landemerker dere vil ha, og hvor de skal ligge. Minimum er ett landemerke, litt flere (3–5) gir nok bedre resultat. Søkene virker best når nodene ligger omtrent på linje, landemerke–start–mål eller start–mål–landemerke. Derfor bør landemerkene ligge rundt kanten av kartet.

Preprosesserte data blir to todimensjonale tabeller. Den ene indeksen er nodenr, den andre landemerkenr.

| Avstander | | | | |
|---|---|---|---|---|
| Til node | Fra landemerke | | | |
| | 0 | 1 | 2 | ⋯ |
| 0 | 112 | 26393 | 377 | |
| 1 | 115 | 26400 | 378 | |
| 2 | 147 | 26450 | 376 | |
| ⋮ | | | | ⋱ |

| Avstander | | | | |
|---|---|---|---|---|
| Fra node | Til landemerke | | | |
| | 0 | 1 | 2 | ⋯ |
| 0 | 102 | 26383 | 395 | |
| 1 | 105 | 26390 | 396 | |
| 2 | 136 | 26440 | 394 | |
| ⋮ | | | | ⋱ |

Tabellen med avstand *fra* landemerker, fylles ut ved å kjøre Dijkstras algoritme med start i hvert landemerke. Algoritmen kjøres til køen er tom (alle noder funnet). Da har hver node en avstand, som kan brukes til å fylle ut kolonnen for det aktuelle landemerket. Så blir det å kjøre Dijktstras algoritme for neste landemerke, og fylle ut en kolonne til. Slik fortsetter det, til alle landemerker er behandlet. Regn med at dette tar flere minutter. For å vite at programmet ikke henger, kan det være en idé å skrive ut hvilket landemerke som behandles.

Tabellen med avstand *til* landemerker, fylles ut ved å bruke Dijkstras algoritme på den *omvendte grafen.* (Graf med de samme nodene, men alle kantene snudd.) Dette er nødvendig, fordi veinettet har mange enveiskjørte veier og rundkjøringer.

Det er lurt å lagre disse tabellene på en egen fil, så dere ikke trenger å preprosessere hver gang dere skal bruke programmet. Å lese data fra en fil (med fornuftig format), går mye raskere enn preprosesseringen. En fil med 6 mill. int (per landemerke) leses dessuten raskere enn tekstfiler med 6 mill. linjer.

## ALT og distanseestimater

Dijkstras algoritme bruker nodens avstand fra startnoden som prioritet. (Det er derfor Dijkstra-søk sprer seg utover som en sirkel rundt startnoden.)

A* bruker summen av nodens avstand fra startnoden, og estimatet for avstand fra noden til målet, som prioritet. Hvis estimatet er godt, blir det veldig lite søk utenfor den optimale ruta.

ALT er en variant av A* som bruker avstand fra landemerker for å estimere avstand.

Avstand fra en node nr. *n* til målet, finner du slik:

Slå opp avstand fra første landemerke til målet, og trekk fra avstand fra første landemerke til *n*. Hvis tallet blir negativt, bruk 0 i stedet.

Slå deretter opp avstand fra *n* til første landemerke, og trekk fra avstanden fra målet til landemerket.

Det største av disse to tallene, er det beste estimatet første landemerke kan gi oss.

Gjør deretter samme beregning for de andre landemerkene. Det største av alle estimatene, er det beste. Det bruker vi som estimat for nodens avstand til målet.

Denne beregningen må gjøres hver gang ALT oppdager en ny node og legger den i køen. Hvis noden senere får ny avstand må den omprioriteres. Men estimatet for avstand til målet endrer seg ikke, så i slike tilfeller trenger vi ikke gå gjennom alle landemerkene omigjen.

En node bør altså ha tre avstandsfelter: ett felt for avstand til start (som endrer seg underveis), et estimat for avstand til mål (som bare beregnes én gang for hver node som legges i kø), og en sum av de to første som prioritetskøen bruker for å finne beste node. Egentlig trengs ikke sum-feltet, man kan ha en aksessmetode som legger sammen ved behov.

## Tips

Her er det nødvendig å stoppe søket når målnoden plukkes ut av køen. Ellers vil programmet sjekke alle nodene, og det tar *lang* tid.

For å kunne beregne reiseruter så lange som Oslo–Trondheim, er det nødvendig å ha en god prioritetskø. F.eks. heap-basert.

De som bruker java, bør bruke 64-bits java. 32-bits får som regel ikke plass til hele kartet i minnet. Men det er vel ingen grunner til å holde på med 32-bits programmer nå for tiden? 64-bit var «nytt og spennende» da jeg studerte på 90-tallet…

Det er lurt å lage programmet slik at det leser inn kartet én gang, og deretter kan gjøre mange søk. Dette fordi søket typisk bare tar et sekund eller to, men lesing av filene tar opp mot et halvt minutt.

### Vise kart selv

Dere kan bruke ferdige bilbioteker/frameworks for å vise kart, om dere ønsker det. Biblioteker fins for Java, C++ og andre språk. Se f.eks. JMapViewer. Men dere må implementere ALT/Dijkstra-søkene selv, altså *ingen ferdige opplegg for routing!*

| | |
|---|---|
| Generelt | `http://wiki.openstreetmap.org/wiki/Software` |
| Frameworks | `http://wiki.openstreetmap.org/wiki/Frameworks` |
| JMapViewer | `http://wiki.openstreetmap.org/wiki/JMapViewer` |

Med et slikt framework, kan dere beregne en reiserute med egen kode, og deretter la JMapViewer tegne opp ruta på et kart. JMapViewer kan selv laste inn kartblad fra en tile-server på nettet.

### Vise ruta på et webkart/google maps

Finn veien mellom to steder, f.eks. «Prinsenkrysset» og «Moholt». Skriv ut ruta med dette formatet:

```
breddegrad1,lengdegrad1
breddegrad2,lengdegrad2
breddegrad3,lengdegrad3
...
```

Bruk en løkke som begynner med målnoden, og deretter følger forgjengerne hele veien tilbake til startnoden. Skriv ut koordinatene i bredde- og lengdegrader for hver node.

Åpne `https://maps.co/gis/`. (Krever innlogging, men gratis for vår bruk.)

Klikk «Import Data», lim inn koordinatlista, klikk «Plot Map Points» Da får dere vist ruta grafisk på google maps. Sjekk om det er en bra/realistisk rute. Vær klar over at google maps har et annet datagrunnlag enn openstreetmap. Det kan føre til programmet deres plotter en vei som ikke fins på google-kartet. Men riksveinettet bør stemme.

Hvis ruta har flere punkter enn dere får lov å plotte gratis, kutter dere ut annenhvert punkt (eller 2 av 3).

### Vanlige feil, prøv å unngå disse

- Bytte om lengdegrader og breddegrader. Enten ved innlesing eller i utskrift. Reiseruta får riktig form, men plottes 90 grader vridd og langt uti sjøen.
- Avbryte søket når programmet oppdager målnoden. Det er for tidlig! Den første veien til målet trenger ikke være den beste! Avbryt heller søket når målnoden plukkes ut av prioritetskøen, for da vet vi at korteste vei til målet er funnet.
- Rot med prioritetskøer. Når man bruker heap og endrer en nodes avstand, (fant en kortere vei dit), så må noden flyttes i heapen (prioritetsendring). Gjør man ikke dette, kommer nodene ut av heapen i feil rekkefølge, og programmet finner ikke korteste vei. Typisk er at veien blir 10%–20% for lang, og tar rare avstikkere fra hovedveier. Inn på rasteplasser/-bensinstasjoner o.l.

  Java sin PriorityQueue har ingen mulighet for å endre prioritet. Sløve greier, men det kan løses med å ta noden ut av køen og sette den inn på nytt med ny prioritet.
- Korrekt rute, men likevel feil kjøretid

  Når søket er gjort, sitter man med en lang rekke noder. For å finne kjøretiden, er det da noen som lager en løkke som finner lengden på kant1 fra hver av disse nodene. Det må bli feil, for det er ikke alltid den *første* kanten som blir brukt.

  Det er ikke nødvendig med noen slik løkke for å legge sammen avstander. Dijkstra (og ALT) legger jo uansett en kjøretid i hver node, etterhvert som stadig raskere veier oppdages. Denne kjøretiden brukes til å prioritere nodene i køen. Så når Dijkstra/ALT er ferdig, ligger kjøretiden for hele turen i målnoden. Ingen grunn til å beregne dette omigjen!

## Tips for hastighet

- Prioritetskøen bør være en heap eller javas innebygde priorityQueue. Søk i en tabell tar for lang tid til å beregne Trondheim–Oslo. Ihvertfall var det slik sist noen prøvde. Spesielt interesserte må gjerne prøve ut andre køer, f.eks. fibonacci-heap.
- Ikke legg noder i prioritetskøen før de blir funnet, dermed blir det mindre jobb å plukke ut den med minst avstand.
- Noen faller for fristelsen til å bruke metoden «`.contains()`» Men for mange konteinere denne er $O(n)$ og dermed altfor langsom. (Lineært søk igjen, det har vi ikke tid til.) Hvis du trenger å vite om en node er «funnet» eller ikke, bruk en `boolean`, `enum` eller `int`. Slik kan du lagre nodens status i nodeobjektet, og teste status i $O(1)$ tid.
- Nordenkartet kan leses linje for linje på 1 sekund, med Java, objektet BufferedReader og metoden readLine() i løkke. Dessverre er String.split() dårlig implementert, og kan trekke kjøretiden opp i et halvt minutt. Her er en noe raskere hsplit som kutter ned tidsbruken:

```
//Unngå dødstreg String.split()
//finner ordene i "linje", oppdelingen havner i "felt[]"
//Gir "index out of bounds" hvis linja ikke har så mange ord, eller >10 ord.
//unngår regex og unøding bruk av "new"
//Alle ascii-verdier <= mellomrom er blanke/skilletegn
//Alle ascii-verdier > mellomrom former ord.
static String[] felt = new String[10]; //Max 10 felt i en linje
void hsplit(String linje, int antall) {
        int j = 0;
        int lengde = linje.length();
        for (int i = 0; i < antall; ++i) {
                //Hopp over innledende blanke, finn starten på ordet
                while (linje.charAt(j) <= ' ') ++j;
                int ordstart = j;
                //Finn slutten på ordet, hopp over ikke-blanke
                while (j < lengde && linje.charAt(j) > ' ') ++j;
                felt[i] = linje.substring(ordstart, j);
        }
}
```

Arrayet felt[] overskrives ved bruk, så ikke bruk hsplit() før programmet er ferdig med å bruke forrige resultat. Problemet med String.split() er at det bruker regex når man trenger å skille mellom flere typer blanke (mellomrom, tab-tegn, linjeskift). Regex er kraftige saker, men overkill og for tregt til å kalles 16 millioner ganger.