Øving 8 algoritmer og datastrukturer

Innhold

Oppgave

Lag et program som både kan finne veien med A* og med Dijkstras algoritme. Programmet skal også kunne finne de 10 bensinstasjonene eller ladestasjonene som er nærmest et sted. Det gjøres med Dijkstras algoritme.

Spesielt interesserte kan bruke ALT-algoritmen i stedet for A*. Da må de i så fall preprosessere kartet.

Dere får kart over de nordiske landene (Openstreetmap, 2020) med koordinater på hver node, dette kan A*-algoritmen bruke. 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 A* (eller ALT) og Dijkstras algoritme. Det måler tidsforbruk for begge algoritmene, og teller opp hvor mange noder hver algoritme trenger å plukke ut fra køen. (kan vises med skjermdump)

   Forhåpentligvis ser vi at A* behandler færre noder – om det også går raskere vil avhenge av implementasjon og reiserute. A* gjør mer arbeid *per node* på grunn av avstandsberegningene.

2. Programmet kan finne de 10 bensinstasjonene som er nærmest et sted, og de 10 ladestasjonene som er nærmest.

   La programmet finne 10 bensinstasjoner nær "Trondheim lufthavn, Værnes". Plott disse på et kart og send inn skjermdump.

   Finn også de 10 ladestasjonene som er nærmest "Røros hotell".

3. Dere implementerer algoritmene selv.

4. A* og Dijkstra algoritmene finner samme vei hvis de får samme mål og start, og samme vei 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 10 sekunder.

   (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.
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 "Trondheim"–"Helsinki". Husk både Dijksta og A* (eller ALT) for begge turene.

A*-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 181 MB. Den inneholder 6 358 355 noder. Den er formatert slik:

```
nodenr breddegrad lengdegrad
```

Lenke: `http://www.iie.ntnu.no/fag/_alg/Astjerne/opg/norden/noder.txt`

Utdrag av nodefila, første linje er antall noder:

```
           6358355
0 65.6107182 -16.9173984
1 65.6063739 -16.8930365
2 65.6133707 -16.8749985
3 66.0015637 -16.5128338
```

Kantfila er på 361 MB, og inneholder 14 311 839 vektede kanter. Formatet er slik:

```
 franode  tilnode kjøretid lengde fartsgrense
```

Lenke: `http://www.iie.ntnu.no/fag/_alg/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.

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:

```
          14311839
0       1       9007    1251    50
1       0       9007    1251    50
1       2       8424    1170    50
2       1       8424    1170    50
```

I tillegg får dere filen: `http://www.iie.ntnu.no/fag/_alg/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 «Helsinki». Denne fila er bare 2,6 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 er tekst i unicode, utf-8. Koden er 1 for stedsnavn som "Trondheim", 2 for bensinstasjoner som "Shell Herlev", og 4 for ladestasjoner som "Grønn kontakt Bømlo".

Hvis noe er både bensin- og ladestasjon, har det kode 6, fordi 4+2=6.

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 14 millioner linjer med tekst! Her er noen andre metoder:

- For linux og mac, bruk kommandoer som denne:

  ```
  wget http://www.iie.ntnu.no/fag/_alg/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. Det går mye fortere å lese inn, det er praktisk når dere utvikler/debugger programmet. Lenker:

`http://www.iie.ntnu.no/fag/_alg/Astjerne/opg/island/noder.txt`
`http://www.iie.ntnu.no/fag/_alg/Astjerne/opg/island/kanter.txt`
`http://www.iie.ntnu.no/fag/_alg/Astjerne/opg/island/interessepkt.txt`

A* og avstandsberegninger, Haversine-formelen

A* trenger å estimere en kjøretid fra node til målnode. Til det brukere dere bredde- og lengdegradene. Bruk variabler av typen double, for float har ikke nok presisjon for å regne med posisjoner på jorda. Haversine-formelen for avstander mellom to punkter langs jordoverflaten er slik:

$$2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{b_1-b_2}{2}\right) + \cos(b_1)\cos(b_2)\sin^2\left(\frac{l_1-l_2}{2}\right)}\right)$$

Der r er radius for jordkloden, 6371 km. b_1 og b_2 er breddegrad for første og andre punkt, l_1 og l_2 er lengdegradene. $\sin^2(x)$ er en matematisk forkortelse for $\sin(x) * \sin(x)$.

I java bruker dere `Math.asin()`, `Math.sqrt()`, `Math.sin()` og `Math.cos()`.

Når dere har avstanden mellom to punkter, kan dere gi et lavt estimat for kjøretiden ved å regne ut hva den vil bli om det går en rett vei dit med høyeste fartsgrense. (I Norden er 130 høyest.)

I filen er koordinatene angitt med *grader.* Men sinus og cosinus regner med *radianer.* Omregning er slik: en grad er $\frac{\pi}{180}$ radianer. Og andre veien, en radian er $\frac{180}{\pi}$ grader. Og ikke bruk «3,14», det er for unøyaktig. Bruk `Math.PI`.

Bakgrunn for Haversine-formelen

Det fins enklere avstandsformler for lengde/breddegrader. De er matematisk korrekt, men får problemer fordi datamaskiner ikke har uendelig presisjon. Vinkelen fra jordas sentrum til to punkter med noen meters mellomrom er ekstremt smal, Haversine-formelen tar hensyn til slikt. Den ble utviklet i seilskutetiden da de hadde samme problem som nå: begrenset regnekraft (en styrmann med tabeller for logaritmer og sinus) og begrenset presisjon (3–4 sifre). Den gang holdt det med få sifre, for man trengte ikke meterpresisjon. For veinavigasjon trenger vi presisjonen i en double.

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 ett sekund, 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 både for Java og C++. Se f.eks. JMapViewer. Men dere må implementere A*/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 grader, ikke radianer) for hver node.

Åpne `http://www.darrinward.com/lat-long/`

Lim inn koordinatlista, klikk på «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.

Javakode som implementerer avstandsformelen

Her er kode som beregner avstanden mellom to punkter, og hvor lang tid det tar å kjøre den avstanden med høyeste fartsgrense i kartmaterialet. Koden antar at vi har nodeobjektene `n1` og `n2`, som har feltene breddegrad og lengdegrad ferdig omregnet til radianer. For å være raskest mulig, beregnes ikke cosinus til breddegradene, da det kan gjøres på forhånd når programmet leser nodene inn fra filen. Objektene mine har feltet `cos_bredde` til dette formålet. Slike optimaliseringer er en frivillig sak. Kjøretid returneres som en integer, i hundredels sekunder.

```
//Jordens radius er 6371 km, høyeste fartsgrense 130km/t, 3600 sek/time
//For å få hundredels sekunder: 2*6371/130*3600*100 = 35285538.46153846153846153846
int avstand (node n1, node n2) {
  double sin_bredde = Math.sin((n1.breddegrad-n2.breddegrad)/2.0);
  double sin_lengde = Math.sin((n1.lengdegrad-n2.lengdegrad)/2.0);
  return (int) (35285538.46153846153846153846*Math.asin(Math.sqrt(
    sin_bredde*sin_bredde+n1.cos_bredde*n2.cos_bredde*sin_lengde*sin_lengde)));
}
```

Vanlige feil, prøv å unngå disse

- Bytte om lengdegrader og breddegrader. Enten ved innlesing eller i beregningen. Dette kan gi veldig rare reiseruter, f.eks. spiralformede, eller Trondheim–Oslo via Bergen. Husk, lengde- og breddegrader er ikke kartesiske koordinater, avstandene blir ikke de samme om man bytter om på dem.
- Bruke «float» når «double» er nødvendig. «float» har ikke nok presisjon til å skille mellom punkter som bare er noen meter fra hverandre. Kryssene ligger tett i byer. Vinkelen mellom dem, fra jordas sentrum, blir veldig smal. «float» runder da av ned til 0.
- Glemme å regne om mellom «grader» og «radianer». sinus og cosinus på datamaskinen arbeider bare med radianer; en rett vinkel er $1/2\pi$ radianer, som er det samme som 90 grader. Hvis A* i det hele tatt finner frem med en slik feil, kan langturene få noen underlige omveier.
- 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.

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.
- Cosinus til breddegrader kan beregnes under innlesing av data, da blir det færre trigonometriske beregninger under kjøring. Det kan gjøre en viss forskjell.
- Noen faller for fristelsen til å bruke metoden «`.contains()`» Men for mange konteinere denne er $O(n)$ og dermed altfor langsom. 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 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 14 millioner ganger.