I120, Høsten 1997: Obligatorisk oppgave

Rekursjon, gjenbruk av kode, dokumentasjon

Frister

Innleveringsfrist: tirsdag 14. oktober, kl.18^{00}.
Innleveringssted: spesielt merket pappeske på terminalrommet i auditoriefløyen.

Formål

Denne øvelsen har som formål å trene dere i rekursiv problemløsing, implementasjon og bruk av gitte programkomponenter. Ved å gjøre den obligatorisk vil vi etterprøve at dere faktisk har gjort dere kjent med disse begrepene og metodene.

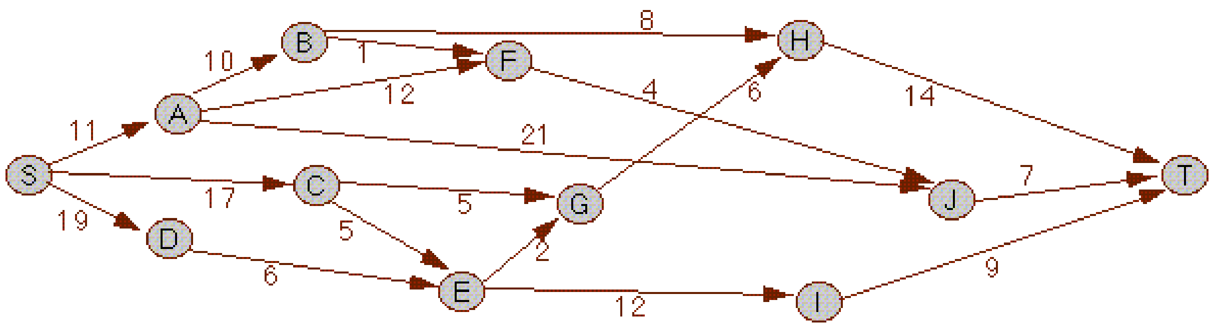
Krav til løsning av oppgaven

1. Fristen overholdes.
2. Oppgaven *skal* løses i grupper på 2 studenter (hverken fler eller færre).
3. Dere skal levere:
   (a) Programlisting (på papir) av løsninger på oppgavene 1-5.
   (b) Dokumentasjon: alle metodene skal dokumenteres, litt om hva metoden gjør, om dens innparametre, resultatet, og hvordan metoden behandler eventuelle eksepsjonelle situasjoner. Denne delen av dokumentasjonen *må* gjøres i javadoc! I tillegg skal en forklaring av datastrukturen leveres.
   (c) Resultatet av kjøring av oppgavene på testdataene henvist til herfra.
   (d) Navn og mailadresse til gruppemedlemmene, og nummeret på øvingsgruppen (man ble tatt opp til, ikke nødvendigvis den man går på) til den av de to gruppemedlemmene hvis etternavn kommer først i alfabetet.
   (e) Kildekoden, datafiler for testkjøring og det ferdige programmet skal finnes (men være utilgjengelig for andre) på undervisningsanlegget. Noen dager etter fristen vil vi be ca. 40% av gruppene om å åpne filene slik at vi kan teste programmene.

Introduksjon til oppgaven

I denne oppgaven skal dere lage to program som løser *'stagecoach'-problemet*, et klassisk problem fra feltet optimering. Bakgrunnen for problemet sies å være en handelsreisende i det ville vesten. Han skulle fra en by til en annen, og hadde valget mellom mange ulike reiseruter som alle var mer eller mindre farlige. Langs veien lå forskjellige småbyer, og mellom disse gikk det postvogner, 'stagecoaches'[1], også kalt diligencer på norsk. Det var omstigning mellom de forskjellige rutene i hver by, og det var også mulig å kjøpe en reiseforsikring som gjaldt enkeltstrekningene mellom hver småby. Forsiktig som denne handelsmannen var, satset han på at jo lavere forsikringspremien var, jo tryggere var reisestrekningen. Han ville derfor velge den reiseruten som totalt sett hadde lavest forsikringspremie, og derfor skulle være den tryggeste. Vi kan representere småbyene som noder (skyggelagte sirkler i figuren nedenfor) i en graf, og forbindelsene mellom byene som kanter (piler i figuren). Til hver kant knytter vi en verdi som representerer forsikringspremien for denne reisestrekningen. Handelsmannen står ved starten i byen merket S (noden lengst til venstre) og skal reise til byen merket T (noden lengst til høyre i grafen).

[1]Se f.eks. westernfilmen 'Stagecoach' for et inntrykk både av selve transportmiddelet og de farer som kunne true. Den finnes i en klassisk filmatisering fra 1939, regissert av John Ford med bl.a. John Wayne i en av rollene. Senere er den filmatisert på ny i 1966 med bl.a. Bing Crosby, og den finnes i en TV-versjon fra 1986 med bl.a. Johnny Cash, Kris Kristofferson og Willie Nelson i sentrale roller.



Oppgaven er å finne kostnaden ved den rimeligste reiseruten, og så skrive ut denne rimeligste ruten. I figuren er kostnaden ved f.eks. ruten S-A-F-J-T 34, ved S-D-E-G-H-T 47. Den rimeligste kostnaden er 33.

Hver by representeres av et objekt av følgende klasse[2] `Node`:

```
class Node {
  public Node(String stedsnavn){
    navn=stedsnavn;
    naboer=new Naboliste();
  }
  public String navn;
  public Naboliste naboer;
}
```

Klassen `Naboliste` er gitt, og *skal* brukes. En nodes naboliste er en slags liste over nodens nabonoder, og kostnadene ved å reise til disse. Klassen er stort sett en forenklet implementasjon av grensesnittet "listeiterator" på s. 153 i læreboken. Den har følgende grensesnitt (se også ~peter/public_html/Oblig1/Naboliste.html):

`public void settinnabo(Node nabo, int kostnad)`

– Setter inn `nabo` som ny nabo til "eiernoden" (dvs. noden til hvilken nabolisten tilhører) med tilhørende kostnad. Hvor den settes inn skulle være uvesentlig.

Vi trenger ikke ta ut naboer, så det er ikke gitt noen metode for fjerning. Når man skal gå gjennom listen av naboer brukes følgende metoder:

`public void settforst()`

– Initierer listeiteratoren, og må alltid kalles før man løper gjennom listen. Tilsvarer metoden `first` på s. 153.

`public Node dennenabo()`

– Gir "aktuell nabo" i retur. Tilsvarer bokens `retrieve`. Gir `null` dersom det ikke er flere naboer i listen.

`public int dennekostnad()`

– Gir kostnad ved å reise fra eiernoden til "aktuell nabo". Kaster `RuntimeException` dersom det ikke finnes flere naboer i listen (det er vel sikrere enn å returnere et heltall?).

`public void nestenabo()`

– Setter "aktuell nabo" til å være etterfølgeren til nåværende "aktuell nabo". Tilsvarer bokens `advance`.

`public boolean flernaboer()`

– Gir `true` dersom aktuell nabo virkelig er en node, `false` ellers.

Denne klassen skal altså benyttes og ligger på ~peter/public_html/Oblig1/Naboliste.java, der også klassen `Node` er definert.

[2] Dersom man ønsker, kan man la alle attributtene i klassen være private, og så la metoder observere verdiene.

Oppgave 1

Første delen går ut på lese inn data fra fil, og sette opp datastrukturen. Todimensjonale array'er skal ikke benyttes. Brukeren av programmet skal bli spurt om filnavnet hvor dataene ligger. En datafil har formatet

```
12
S  A  11
C  E   5
S  D  19
...
```

På den første linjen står et tall som angir antall byer. Deretter følger et antall linjer som navngir byer og gir avstander mellom ulike par av byer. De 4 linjene over kunne vært starten på en fil for eksempelfiguren. Det er viktig å merke seg at input-filene alltid vil være korrekte, dvs. angir løkke-frie grafer og har det angitte antall byer. Bynavnene er tekstrenger som ikke inneholder blanke (eksempelvis kunne vi hatt stedsnavn som *Tombstone* og *Carson_ City*, mens *Bullhead City* ikke er et korrekt stedsnavn). Merk også at ingen stedsnavn vil ha navnet "EOF". Det ligger to datafiler på området *~peter/public_ html/Oblig1/* med navn *byer1.txt* og *byer2.txt* som skal testes.

Oppgave 2

En sti er det vi får når vi krysser null eller flere kanter i rekkefølge (i pilens retning). En løkke (eller sykel) får vi dersom det finnes en sti med minst en kant som starter og slutter i samme node. Vi kan i det følgende anta at avstandsgrafen er løkke-fri. Oppgaven her er å implementere en metode

```
public int minstekostnad (Node fra, Node til)
```

som angir kostnaden av den rimeligste turen fra `fra`-noden til `til`-noden. Man kan anta at både `fra` og `til` virkelig er noder som er i datastrukturen, og at det finnes en sti fra `fra` til `til`.

Metoden skal implementeres som en rekursiv metode (eller ved et kall på en rekursiv metode), og gjerne etter følgende idé:

```
public int minstekostnad (Node X, Node Y)
```

Forkrav: Kostnadsgrafen sykelfri. `X` og `Y` er noder i strukturen.
Resultat: gir kostnaden for den rimeligste reiseruten fra `X` til `Y` dersom den finnes. Ellers kan den passelig returnere `Integer.MAX_VALUE` eller et negativt tall.
Algoritme:
hvis `X`=`Y` så returner 0
ellers hvis ikke `X` ikke har noen naboer, returner en verdi som sier dette (f. eks. `Integer.MAX_VALUE`)
ellers undersøk, for alle nabonoder `Z` av noden `X`:
hva kostnaden `C` ved å reise fra `X` til `Z` er
hva kostnaden `V`=`C+minstekostnad(Z,Y)`, som gir rimeligste reise fra `X` til `Y` via `Z` er;
returner den laveste av alle `V` regnet ut over;

Ideen bak algoritmen er enkel. Hvis handelsmannen står i byen X, og det er den samme byen som Y, så er han fremme, og kostnaden ved å reise til Y er 0. Dersom han ikke er fremme i Y, så sjekker vi kostnaden ved alle alternative reiseruter han har for øyeblikket. Hver av disse reiserutene må gå gjennom en av naboene til X, der den vi undersøker for øyeblikket blir kalt Z. Kostnaden ved å reise via Z består av to deler: kostnaden ved reisen fra X til Z, betegnet C, og den rimeligste kostnaden ved å reise fra Z til Y. Summen av disse delkostnadene betegnes V, og den laveste av disse returneres som den rimeligste kostnaden fra X til Y. Hvis han ikke kan komme videre fra X, en svært uheldig situasjon, så er kostnaden ved å komme seg til målet uendelig stor (≈`Integer.MAX_VALUE`) eller en annen "feilverdi".

Oppgave 3

Selvsagt er det hyggelig for vår handelsreisende venn å vite hvor dyr den billigste veien er, men han ville likt det enda bedre dersom programmet i tillegg skrev ut selve den billigste turen (som i eksempelgrafen er `S-A-B-F-J-T` dersom man vil gå fra `S` til `T`). Dere skal i denne oppgaven programmere en metode

```
public void skrivbilligstevei(Node fra, Node til)
```

som skriver ut (i riktig rekkefølge) den billigste veien (vi antar at det finnes en sti fra `fra` til `til`) fra `fra` til noden `til`. Denne kan passelig kalle en rekursiv metode som generer mulige veier og som oppdaterer noen globale variable i klassen. (Variablene kan f. eks. angi kostnad av aktuell vei, kostnad av beste vei, aktuell vei, og hittil beste vei.)

Oppgave 4

Det er ineffektivt og unødvendig å fortsette å generere veier når den veien vi holder på å lage allerede (før den er ferdig) er dyrere enn hittil billigste totalvei. Legg derfor inn en avskjæring slik at metoden lagd under oppgave 3 ikke bygger videre på veier som umulig kan bli billigere enn hittil billigste vei. Legg også inn en passende invariant som forteller at en slik avskjæring er gjort.

Oppgave 5

Lag et hovedprogram som lar brukeren teste rutinene ved å spørre henne om input-data til rutinene.

Innlesing fra fil: Jeg har utvidet de tidligere oppgitte terminalinnlesingsrutinene til å kunne lese fra fil. Et Lesfrafil objekt konstrueres ved å gi en `InputStream` (f. eks. `new FileInputStream(''data.txt''))` som parameter til konstruktøren. Hvert objekt har metoder `lesord()` og `lesnesteint()` som gir hhv. neste streng og neste heltall i filen. `lesord()` returnerer "EOF" når -1 (filsluttmerke) leses, mens `lesnesteint()` da returnerer -1. Kildekoden ligger på ~peter/public_html/I120/Lesfrafil.java.