Reseplanerare

Denna uppgift demonstrerar centrala delar av imperativ programmering i C genom ett verklighetsanknutet exempel – en enkel reseplanerare. Vi kommer att fokusera på ”modellen”, dvs. sökning efter olika rutter, och inte tänka på gränssnittet.

Partiell lista över innehållet

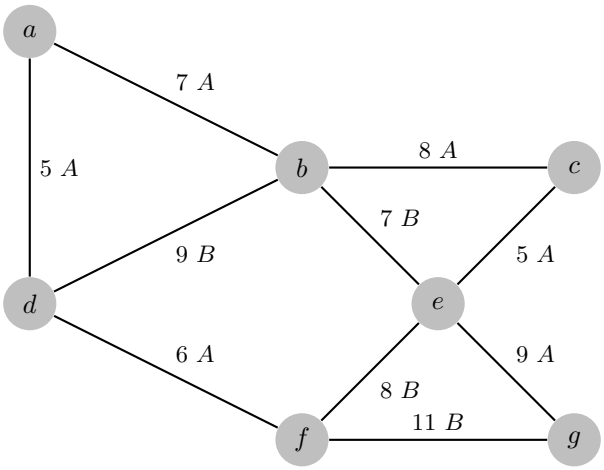
- Dynamiska datastrukturer (grafer och listor)
- Pekare
- Enkel I/O
- `void`-pekare

Introduktion

Hur busslinjer knyter samman olika platser kan enkelt beskrivas som en viktad graf där noderna är platser och bågarnas bär etiketter med linjenamn och vikter som avser restiden mellan platserna. Grafen för ett bussnätverks samtliga linjer tillsammans med en turlista som beskriver deras linjers starttider från sina starthållplatser har vi nu information nog för att besvara frågor som ”Ge mig en komplett lista över alla bussar som passerar Polacksbacken och vilka tider de passerar”, ”När måste jag starta från Polacksbacken för att komma fram till Martallsvägen senast kl. 17.00?”.

För enkelhets skull kan vi förutsätta att alla busslinjer har samma sträckning jämt och samma tider alla dagar.

Nedan visas en graf för ett bussnätverk med stationerna $a - g$ för två busslinjer A och B. För att hålla exemplet enkelt trafikeras varje strecka endast av en busslinje. A går sträckan f, d, a, b, c, e, g och B går sträckan d, b, e, f, g. Vid varje sträckning finns en tid utsatt. Bussbyten är möjliga vid alla stationer utom a och c. Modulo möjligheten att byta buss, om man vill åka från a till e måste man åka a, b, c, e vilket tar $7 + 8 + 5 = 20$ minuter. Om vi tar med bussbyten i beräkningarna och väntetiden t i b är sådan att $t < 6$ minuter skulle det gå fortare att åka a, b, c med byte i b och den totala restiden $7 + 7 + t$. (Rimligtvis finns det alltid en minsta tid för ett bussbyte, säg 2 minuter. Därav skulle följa att resan a, d, f, e med byte i f skulle ta minst 21 minuter om bussarna tajmar perfekt.)



Uppgiften Ni har i denna uppgift en datafil för ett bussnätverk där varje rad innehåller en busslinje, namnet på två hållplatser sammanlänkade av busslinjen, samt tiden det tar att åka mellan dem, t.ex.:

```
110, Polacksbacken, Grindstugan, 2
110, Grindstugan, Uppsala Science Park, 1
...
```

Utöver detta data har ni också en datafil som anger starttiderna för varje busslinje från sina respektive ändhållplatser där varje rad innehåller en busslinje, ändhållplatsen, och en starttid, t.ex.:

```
110, Lingonvägen, 05:00
110, Lingonvägen, 05:30
110, Lingonvägen, 05:55
110, Lingonvägen, 06:20
...
110, Arrheniusplan, 06:50
110, Arrheniusplan, 07:20
110, Arrheniusplan, 07:50
...
```

Ni skall nu:

1. Implementera ett grafbibliotek inklusive sökning efter kortaste vägen mellan två noder. Detta bibliotek skall *helst* vara generellt så tillvida att en det inte är specifikt för bussar, utan en nod innehåller godtycklig information (**void**-pekare) och likaså för bågar. En bra designstrategi är att göra programmet generellt efter att man först gjort en fungerande specifik implementation.

   Det bör bl.a. finnas stöd för att:

   (a) Skapa en ny nod med data
   (b) Koppla samman två noder med data till den skapade bågen
   (c) Ta bort en nod
   (d) Ta bort en båge
   (e) Hitta den kortaste vägen mellan två noder (t.ex. med Dijkstras algoritm[1])

   För enkelhets skull kan man bortse från byten och förutsätta att man vill resa med samma busslinje hela resan. (Men implementera gärna stöd för byte!)

   Om bågar och noder sparar godtyckligt data kommer sökalgoritmen att behöva pekare till funktioner som hjälper till att tolka datat. Man kan t.ex. tänka sig att varje båges **void**-pekare pekar ut en strukt `edgedata` och att användaren av biblioteket implementerar en funktion `getWeightFromEdgeData`:

   ```
   1  struct edgedata {
   2    unsigned short travel_time;
   3    unsigned short bus_line;
   4  };
   5
   6  unsigned short getWeightFromEdgeData(void *d) {
   7    return ((struct edgedata*) d)->travel_time;
   8  }
   ```

   I en implementation där en båges data är känd skulle t.ex. kunna innehålla raden:

   ```
   ... = edge->weight;
   ```

   men om man istället har säg `edge->payload` där `payload` är **void*** blir motsvarande rad:

   ```
   ... = getWeightFromEdgeData(edge->payload);
   ```

   som logiskt sett kan tänkas expandera till

   ```
   ... = ((struct edgedata*) edge->payload)->travel_time;
   ```

   Notera att grafbiblioteket skall tänkas på som ett separat bibliotek (även om du inte går steget ut och bryter ut beroenden till busslinjedomänen med hjälp av **void**-pekare.) Det betyder att biblioteket bör ha egna headerfiler, etc.

2. Implementera stöd för att läsa upp busslinjer från fil (den första datafilen) och med hjälp av biblioteket du skapade ovan bygga en komplett graf över nätverket i datorns minne.

3. Med hjälp av grafen och den andra datafilen skall du nu implementera stöd för enkla sökningar, t.ex.:

[1] Se en algoritmbok eller utgå från `http://sv.wikipedia.org/wiki/Dijkstras_algoritm`.

Möjliga resvägar från X till Y Svaret här bör vara vilka busslinjer jag kan ta modulo eventuella byten (se utökning nedan), samt vilka stopp som är på vägen för dessa busslinjer, och den totala restiden per sträcka.

Åka från X till Y kl. Z Vad har jag för möjligheter? Vilka rutter? Vad får jag för ankomsttider?

Åka från X till Y och vara framme senast kl. Z Vad har jag för möjligheter? Vilka rutter? När måste jag åka med de olika alternativen?

Du får själv bestämma gränssnittet för hur frågor ställs i systemet och på vilket format svaret ges, men det skall vara begripligt för en assistent. Absoluta minimum är en av de ovanstående exempelsökningarna, och i alla fall en idé för hur de andra skulle kunna implementeras.

4. Frivilliga utökningar
   (a) Stöd för olika bytestid vid byten mellan olika bussar
   (b) Stöd för att förändra en rutt (t.ex. ta bort en hållplats) utan att behöva skapa om nätverket