Inlämningsuppgift, EDAF30, 2022

1 Anvisningar för redovisning

Inlämningsuppgifterna ska redovisas med en kort rapport och de program som du har skrivit. Gör så här för att lämna in inlämningsuppgiften:

1. Arkivera lösningen (med zip eller tar, men *inte något annat arkivformat* som t ex rar), med rapport (som .pdf) och kompilerbar källkod, Makefile, och eventuella indatafiler. Var noga att inte ta med genererade filer (t ex `.o` eller `.exe`) i arkivet. Koden ska gå att bygga med `g++` eller `clang++` och `make` (samt eventuelt `cmake`).
2. Instruktioner för hur uppgiften ska skickas in publiceras senare.

2 Krav på uppgiften

2.1 Rapport

Uppgiften ska redovisas med en kort (ett par sidor) rapport som översiktligt presenterar din lösning. Följande punkter ska diskuteras:

1. Övergripande design: beskriv klasser och funktioner, och deras relationer till varandra.
2. En kort användarinstruktion: hur bygger och testar man programmet? Försök att paketera så mycket som möjligt med regler i makefilen. Det underlättar både under ert arbete och gör att denna instruktion blir väldigt enkel att skriva.
3. Brister och kommentarer: Finns det något i lösning som du i efterhand anser borde gjorts annorlunda? Andra kommentarer?

Rapporten ska lämnas in som pdf-fil.

2.2 Program

Er lösning ska naturligtvis fungera och lösa den angivna uppgiften. Testning ingår som ett krav i uppgiften, och både hur väl testerna täcker uppgiften, och hur väl programmen fungerar bedöms. Provkör allting ordentligt innan ni lämnar in er lösning.

Exempelprogram och testprogram

Det är ett krav att det ska finnas dels enhetstester för alla ingående delar (klasser och funktioner), och dels ett exempelprogram som visar hur er lösning fungerar genom ett exempel som producerar någon sorts utdata i terminalen, och eventuellt är interaktivt. Testprogrammen ska vara skrivna så att det inte krävs någon manuell kontroll av utdata för att avgöra om testet lyckades eller misslyckades: varje testfall ska – på något sätt – svara "ja" eller "nej". Det ska även finnas ett huvud-testprogram som kör alla enhetstester och tydligt visar vilka som inte uppfylldes.

Generella krav

Programkoden i lösningen ska uppfylla följande krav:

1. Programmet ska ha en vettig design.
2. Klasser och funktioner ska ha tydligt avgränsade uppgifter och ansvarsområden.
3. Minneshantering ska vara korrekt: programmet får inte läcka minne.
4. Programkoden ska vara lätt att följa och förstå.
5. Koden ska vara formatterad på ett sätt som underlättar läsning. Detta innebär en vettig indentering och att raderna inte är för långa.
6. Funktions- och variabelnamn ska vara väl valda och återspegla funktionens eller variabelns innebörd.
7. Programmet ska kompilera med `-Wall -Wextra -Werror -pedantic-errors`

   En tumregel, både för design och läsbarhet, är att en funktion inte får vara längre än 24 rader eller ha fler än 5–10 lokala variabler eller mer än tre indenterings-nivåer. Det finns ibland goda skäl att göra ett undantag från detta, men ofta är det bättre att dela upp en lång, komplex, funktion i några enkla hjälpfunktioner. Varje funktion ska bara göra *en* sak, gör den flera – dela upp den.

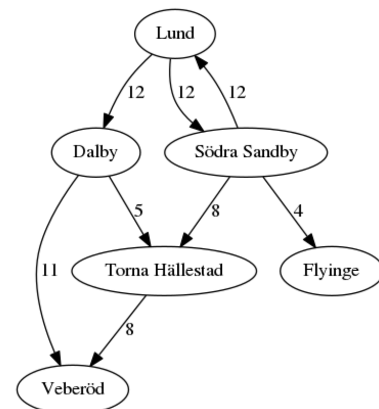
3 Rättning

Vi rättar uppgifterna så snart vi hinner, normalt inom en arbetsvecka räknat från inlämningsdag. När uppgiften är rättad får du besked om uppgiften är godkänd eller ej. Du får en kort sammanfattande kommentar om din lösning samt i de fall uppgiften inte är godkänd kommentarer om vad som behöver förbättras. Om uppgiften inte är godkänd ska du inom rimlig tid lämna in en förbättrad version.

Dijkstras algoritm: kortaste vägen i en graf

1 Bakgrund

I figur 1 till höger visas ett exempel på en datastruktur som kallas *graf*. Grafen består av två slags objekt: *noder* (eng. node) och *bågar* (eng. edge). Varje nod har ett *namn*, som ”Dalby”, och varje båge är märkt med en *längd* (ett heltal, exempelvis 12).



Figur 1: En graf med sex noder och åtta bågar.

Bilden visar en vanlig tillämpning för grafer, nämligen geografisk information. Här visas några orter i Lunds omnejd med de inbördes avstånden (i kilometer) angivna: det är 12 km från Lund till Södra Sandby, och 4 km därifrån till Flyinge. Notera att bågarna är enkelriktade: i vårt exempel finns en båge från Lund till Dalby, men inte tvärtom.

Grafen är alltså ingen komplett karta. I bilden har vi valt att rita noden Flyinge nära Torna Hällestad, men i verkligheten ligger ju Flyinge mer än en mil därifrån (åt Eslöv till).

Vi ska nu införa klasser för noder och bågar.

För noder behöver vi hålla reda på ett namn och ett antal bågar, samt ett *värde*. Nodens värde är ett heltal som vi kommer att få användning av (beskrivs senare).

För bågar håller vi reda på bågens destination (en nod) och längd.

Klasserna `Node` och `Edge` beskriver noder respektive bågar, och har följande specifikationer.

`Node`

```
/** Skapar en nod med givet namn, utan bågar och med maximalt värde. */
Node(const std::string& name);

/** Hämtar nodens namn. */
std::string getName() const;

/** Sätter nodens värde till v. */
void setValue(int v);

/** Hämtar nodens värde. */
int getValue() const;

/** Lägger till en ny båge från denna nod till en given destination.
        Bågen ska ha längden length. */
void addEdge(Node* destination, int length);

/** Hämtar de bågar som utgår från denna nod. */
const std::vector<Edge>& getEdges() const;
```

Edge

```
/** Skapa en ny båge till noden destination, med längden length. */
Edge(Node* destination, int length);

/** Hämtar bågens destination. */
Node* getDestination();

/** Hämtar bågens längd. */
int getLength() const;
```

Exempel: Följande satser bygger upp grafen i figur 1 och skriver ut något om noderna som kan nås från noden med namn ”Dalby”.

```
Node lund{"Lund"};
Node dalby{"Dalby"};
Node sandby{"Sodra Sandby"};
Node hallestad{"Torna Hallestad"};
Node flyinge{"Flyinge"};
Node veberod{"Veberod"};

lund.addEdge(&dalby,12);
lund.addEdge(&sandby,12);
dalby.addEdge(&veberod,11);
dalby.addEdge(&hallestad,5);
sandby.addEdge(&lund,12);
sandby.addEdge(&flyinge,4);
hallestad.addEdge(&veberod,8);

cout << "Anslutningar från " <<dalby.getName() << "(" << dalby.getValue() << ") :\n";
for(auto de : dalby.getEdges()){
    cout << de.getLength() << " to " << de.getDest()->getName() << endl;
}
```

När dessa satser körs fås utskriften

```
Anslutningar från Dalby(2147483647) :
11 to Veberod
5 to Torna Hallestad
```

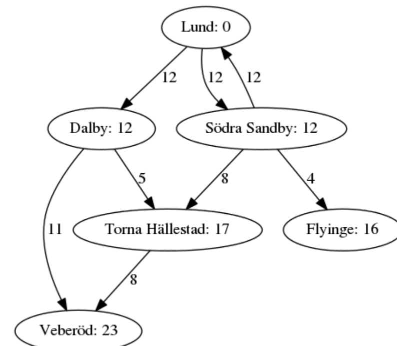
Här ser vi att noderna har värdet `std::numeric_limits<int>::max()` = (i detta fallet) 2147483647.

Kortaste väg mellan två noder

Det är ofta intressant att ta reda på hur lång den kortaste vägen mellan två noder i en graf är. Givet grafen i figur 1 kan vi exempelvis fråga oss hur lång den kortaste vägen från Lund till Veberöd är.

För att lösa detta problem ska vi använda Dijkstras algoritm[1], som vi strax ska titta närmare på. Denna utgår från en given nod *start* och innebär att varje nod N i grafen får ett värde som motsvarar det kortaste avståndet från *start* till N (antingen direkt, eller via en eller fler andra noder).

Om vi använder Dijkstras algoritm med *start* i Lund får noderna värden enligt figur 2. Här har Veberöd fått värdet 23, eftersom kortaste vägen från Lund till Veberöd är $12 + 11 = 23$ (via Dalby).



Figur 2: Grafen i figur 1, med avstånd från Lund utmärkta.

Dijkstras algoritm

Algoritmen utgår från att alla noder inledningsvis har ett värde som är större än alla faktiska avstånd i grafen. För en `int` kan man inkludera `<limits>` och använda `std::numeric_limits<int>::max()` (motsvarar `Integer.MAX_VALUE` i Java).

1. Låt *start* vara noden vi vill räkna avstånd från. Sätt *start*:s värde till 0.
2. Låt S vara en mängd av noder. Inledningsvis ska S innehålla en enda nod, nämligen *start*.
3. Så länge mängden S inte är tom ska följande upprepas:
   a) Ta ut den nod ur S som har lägst värde. Kalla noden n.
   b) Gå igenom de bågar som utgår från n. För var och en av dessa bågar, gör följande:
      - Kalla bågens längd för l och dess destination för d.
      - Låt a vara summan av n:s värde och l.
      - Om a är mindre än d:s värde: ändra d:s värde till a, och lägg in d i mängden S.

Algoritmen bygger på följande idé. Mängden S innehåller de noder som vi funnit en ny väg till. När vi hittat en väg till en nod n, innebär det att vi kanske även hittat ett kortare avstånd a till dess granne d. I så fall stoppas d in i mängden, så att även dess grannar undersöks på samma sätt.

2 Uppgiften

Uppgiften är att implementera klasserna för att representera en graf samt att implementera Dijkstras algoritm för denna graf. Här ges nu en specifikation av kraven för uppgiften, formulerade som ett förslag på arbetsgång.

För att lite närmare specificera klasserna och deras förväntade beteende finns det testfunktioner för dessa. Det rekommenderas att dessa tester används, och anpassas eller utökas där det är motiverat. Ta med era testfall i den kod ni lämnar in.

1. Implementera klassen `Node`. Använd en `std::vector<Edge>` för att hålla reda på bågarna.
2. Implementera klassen `Edge`.

   I filen `test_graph_small.cc` finns funktioner som skapar noder och bågar enligt exemplet ovan. Testa dina `Node` och `Edge`-klasser med dessa. Testprogrammet är skrivet så att det inte ger någon utmatning om testerna går igenom. För att se vad som händer kan du kompilera med makrot `INFO` definierat och kontrollera att utskriften blir:

   ```
   Anslutningar från Dalby: 2147483647:
   11 to Veberod
   5 to Torna Hallestad
   ```

3. I Dijkstras algoritm ska vi använda en mängd med noder. För detta ska vi använda klassen `NodeSet`. Förutom att hålla reda på en mängd noder ska klassen innehålla en funktion för att hitta och ta bort noden med minsta värdet.

   `NodeSet`

   ```
   /** Skapar en tom mängd av noder. */
   NodeSet();

   /** Lägger till noden node till mängden,
       om mängden inte redan innehåller en nod med samma namn. */
   void add(Node* node);
   ```

```
/** Returnerar true om mängden noder är tom. */
bool isEmpty() ;

/** Väljer ut den nod som har lägst värde och returnerar den.
    Den returnerade noden tas bort ur mängden.
    Om mängden är tom returneras null. */
Node* removeMin();
```

Implementera klassen. Använd en lista av typen `std::vector<Node*>` för att hålla reda på noderna. Du kan använda filen `test_nodeset.cc` som exempel på hur klassen ska användas och dess förväntade beteende.

4. Implementera Dijkstras algoritm. Du kan antingen göra det som en fri funktion eller kapsla allting i en klass.

```
/** Dijkstras algoritm. Varje nod som kan nås från start ges ett värde,
    där värdet anger det kortaste avståndet från noden start.
    Alla noder förutsätts, när funktionen anropas, ha ett värde som är
    större än alla faktiska avstånd i grafen.
 */
void dijkstra(Node* start);
```

Ledning: Inuti funktionen `dijkstra` ska du skapa och använda ett `NodeSet`-objekt.

I filen `test_dijkstra.cc` finns ett testfall som bygger en liten graf, anropar en fri funktion `dijkstra` (ändra detta om du lagt funktionen i en klass) och kontrollerar att avstånden från Lund är:

```
Lund: 0
Dalby: 12
Torna Hällestad: 17
Flyinge: 16
Veberöd: 23
```

5. Om vi ska kunna beräkna avstånden i en godtycklig graf måste vi hålla reda på noderna på något annat sätt än genom att ha enskilda variabler för varje nod. Skriv därför en klass `Graph` som håller reda på noderna i en `std::vector<std::unique_ptr<Node>>` (eller en `std::vector<Node*>`, men då måste minneshanteringen skrivas korrekt). För att se vilka funktioner som ska finnas i klassen `Graph` ska du studera filen `test_graph_nofile.cc`. Notera att om du vill köra Dijkstras algoritm flera gånger på samma graf behöver det finnas funktioner så att man kan återställa alla nodernas värde till det initiala maxvärdet.

6. I exemplet i avsnittet Bakgrund har vi deklarerat en variabel per ort, men detta är opraktiskt om vi har många godtyckliga noder och bågar. Därför ska vi nu läsa in information om orterna och avstånden från en fil. Lägg till följande konstruktor i klassen `Graph`:

```
/** Skapar en graf med noder och bågar som läses från strömmen in. */
Graph(std::istream& in);
```

Indata om grafen är en textfil som består av rader, där varje rad representerar en båge mellan två orter och har formen `ort1 : avstånd ort2` och med ett radslut direkt efter andra ortsnamnet[2]. Exempel:

```
Lund: 12 Dalby
Lund: 12 Sodra Sandby
Dalby: 12 Sodra Sandby
Dalby: 5 Torna Hallestad
Dalby: 12 Lund
Dalby: 11 Veberod
Veberod: 11 Dalby
```

7. Lägg till funktionalitet i programmet så att namnet på noderna längs den kortaste vägen skrivs ut. Exempel:

```
Lund Dalby Veberod 23
```

Ledning:

- Lägg till en medlemsvariabel `Node* parent` i klassen `Node`. Lägg till motsvarande set- och get-funktioner.
- Uppdatera `parent` (ska referera till nodens föregångare) varje gång en nod får sitt värde minskat inuti funktionen `dijkstra`.
- Efter anropet av funktionen `dijkstra` kan man utgå från destinationsnoden och via `getParent()` nysta sig bakåt mot startnoden (som har `parent == nullptr`).
- Det är lättast att skriva ut noderna baklänges (från destination till start). Börja med det och se att du får den utskrift du förväntar dig. Sedan kan du använda till exempel en `std::vector<Node*>` för att mellanlagra noderna innan utskrift. För att skriva ut dem i omvänd ordning kan du använda en `std::vector` som en *stack (eller LIFO-kö (last in-first out))* med hjälp av medlemsfunktionerna `push_back()`, `back()` och `pop_back()`, eller helt enkelt iterera över den baklänges med dess `reverse_iterator`.

8. Generalisera din implementation `dijkstra()` så att användaren kan välja vad som menas med "kortaste vägen". Visa att den generaliserade dijkstra-funktionen fungerar genom att skriva två funktioner så att längden på en väg definieras som

   a) vägavstånd (enligt tidigare)
   b) antal passerade orter

   och använd dessa med den generaliserade dijkstra-funktionen så att användaren kan beräkna den väg genom grafen som är a) kortast eller b) passerar minst antal orter.

   Notera att det inte räcker att programmet kan optimera på vägavstånd och antal passerade orter, detta är bara två exempel på möjliga optimeringar, och de ska inte vara hårdkodade i dijkstra-funktionen.

   Skriv även ett testprogram för din generaliserade `dijkstra()` som testar att hitta en väg mellan två orter som passerar så få orter som möjligt.

   Notera att funktionen `dijkstra()` i sig ska vara generell och inte känna till eller vara beroende av kostnadsfunktionen. Det ska vara möjligt att använda samma `dijkstra`-funktion för en godtycklig optimering, till exempel att hitta en väg med minsta antal bokstäver i de passerade orternas namn. Man ska alltså, *utan att ändra på själva `dijkstra`-funktionen*, kunna optimera med en godtycklig kostnadsfunktion.

   Ledning: Det finns olika sätt att göra detta:

   - Lägg till en parameter till funktionen `dijkstra` för att skicka in funktionen som beräknar kostnaden för att följa en viss båge från en viss nod. Det kan t ex vara ett funktionsobjekt som har en `operator()(const Node&, const Edge&)`.
   - Man kan även lösa detta genom att göra funktionen `dijkstra` till en funktionsmall, eller
   - använda en klass som innehåller kostdnadsfunktionen som en medlemsfunktion och skriva en subklass till denna med en annan kostnadsfunktion.

   Redovisa både din ursprungliga och din generaliserade `dijkstra`-implementation i rapporten.

---

[1] Ursprungligen beskriven i E.W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik* **1**, 1959.

[2] Formatet är valt för att enkelt kunna hantera ortsnamn som innehåller mellanslag.