Grafer

Hiba Qutbuddin Habib

Oktober 2022

Introduktion

Grafer är en datastrukur som kommer egentligen från matematik, då man snackar om grafteorier. En graf kan beskrivas som en uppsättning av en ändlig mängd av noder, dessa noder paras ihop med kanter, dessa kanter kan beskriva som direkt väg mellan två noder men också en indirekt väg mellan två olika noder.

Tidigare i kursen arbetade vi med olika graf typer såsom träd samt länkade lista, dock de typer av grafer hade en del olika begränsningar såsom att samtliga noder kan ej ha kanter till vilket andra noder som helst. I denna uppgiften kommer vi att bortse från dessa begränsingar, och vi kommer att arbeta med riktade grafer, alltså om det finns en väg mellan den ena noden till andra så bör det finns också ett väg tillbaka.

Syftet med denna uppgiften är då att förstå hur en graf datastuktur fungerar samt tidkomplexitet för denna algoritmen.

Tåg från Malmö

Som en datagrund för denna uppgiften så kommer vi att använda Sverige järnvägsnät, detta för kunna förstå hur en graf datastruktur tillämpas i verkligheten.

Grafen

För att kunna läsa av Sveriges järnvägnät och kunna använda den datan, så kommer vi att implementera en dubbelriktade graf datastrukur. För att representera en graf så kommer vi att behöva implementera ett antal olika klasser.

Den första klassen är då en klass som representerar en objekt av typen städer, vi kallar den för "city" , den klassen kommer att ha en metod som skapar en förbindelse mellan två olika städer, då en förbindelse kan beskrivas med en namn för stad samt distans mellan städer i minut, representation av klassen "city" ser ut enligt:

```
String name ;
Connection[] neighbours ;
```

```
public City(String name){
    this.name=name;
    this.neighbours=new Connection[4];}
```

För att vi ska kunna representera grannar för en stad som en vektor som innehåller den direkt förbindelse mellan dessa, så behöver vi ytterligare en klass som beskriver vad är en direkt förbindelse, denna klassen kallas för "Connection", denna klassen kan defineras enligt:

```
  City city ;
  Integer dist ;
public Connection(City city, int dist){
    this.city=city;
    this.dist=dist;
    }
```

Kartan

Med hjälp av objektet "city" så kan vi definera en karta. En karta består av en samling antal städer, då vi kommer representera 54 städer samt 75 förbindelse, detta eftersom filen för Sveriges järnvägnät som vi använder, innehåller denna mängd data. För att kunna placera alla dessa städer i en vektor så använder vi oss av en hash funktion, detta för att vi ska inte slösa på utrymmet.

Map klassen kommer att innehålla ytterligare en metod som representerar dem olika noder som beskriver städer när man traversera kartan, eller om ett given stad är inte med i kartan så skapas det en ny stad, denna metoden kallas för "Lookup" .

Kortaste vägen från A till B

Nu när vi har implementerat klart karta så kan vi använda den för att hitta den kortaste väg mellan en given stad till en annan. Vi kommer nöje oss med att implementera en metod som returnerar antal minuter det tar mellan de två städer.

För att hitta den kortaste vägen så implementerar vi två olika metoder, en rätt så enkel och en förbättrade version av den.

Strategin som vi tillämpar kallas för "Depth-First" och den första simpla metoden vi använder kommer att implementeras rekursivt, då man kommer behöva ha en maxvärde för att inte hamna i en oändlig loop samt sökning av den kortaste vägen implementeras enligt:

```
public static Integer shortest(City from, City to,Integer max){
for(int i=0;i<from.neighbours.length; i++){
// Om start noden har grannar så letas rekursivt efter den kortaste vägen
    if(from.neighbours[i] != null){
        Connection conn =from.neighbours[i];
```

```
        Integer dist=shortest(conn.city,to,max-conn.dist);
//Om distance inte lika med null, uppdateras värdet shrt
    if((dist != null) && ((shrt == null) || (shrt>dist + conn.dist)))
        shrt = dist+conn.dist;
        if((shrt != null) && (max>shrt)){
        max=shrt; } } }
        return shrt;]
```

Benchmarks

Med metoden ovan beräknades den kortaste vägen mellan ett antal olika städer, alltså den korstaste antal minuter mellan två givna städer samt tid det tog för programmet för att hitta vägen.

Resultat

Resultat såg ut enligt: Utifrån resultatet ovan ser vi att kod implementering

Från-Till	**Kortste resa(min)**	**Program svarar efter(ms)**
Malmö-Göteborg	153	2
Göteborg-Stockholm	211	4
Malmö-Stockholm	273	2
Stockholm-Sundsvall	327	18
Stockholm-Umeå	517	$6,6 * 10^3$
Göteborg-Sundsvall	538	$56 * 10^3$
Sundsvall-Umeå	190	$2,2 * 10^3$
Umeå-Göteborg	728	1
Göteborg-Umeå	-	gav up, tog lång tid

Table 1: Tid för den kortaste resa, samt programmet svarstid

som vi har är inte den optimala, då vid sökning efter vissa resor så tog den alldels för lång tid för att få ett svar, exempelvis från Göteborg till Sundsvall så tog det nästan en minut(0,94 min) för programmet att returnera ett svar samt när vi sökte efter en resa mellan Göteborg och Umeå så krävde det extremt lång tid vilket gjorde att man tappat tålamod. Om man tänker på en verklig situation så vill man ofta ha en snabb svarstid vid sökning efter en specifik resa, med tänken på att koden ovan kräven jätte lång tid så är det inte så effektivt. Anledning att den kräver så lång tid är då för att metoden vi har sparas inte de städer som man redan har besökt vilket gör då att man hamnar i en loop, med tanken att man kommer förmodlingen besöka en nod mer än en gång, då vid en tidigare gång visste man att man måste välja en annan väg.

Detecktering av loopen

För att metoden vi implementerade ovan ska blir bättre på svarstid så behöver den en liten optimering. "For" loopen vi har ovan är tillräckligt bra dock vi optimerar den så att den ska kunna komma ihåg vilka städer den har besökt och på så sätt så kommer man inte behöva passera de igen vid sökning, för att man ska kunna få ihop detta så behövs det en vektor av typen "City" i den vektor sparas de städer som har redan besöktes, detta kan se ut enligt:

```
for (int i = 0; i < sp; i++) {
      if (path[i] == from)
          return null; }
  path[sp++] = from;
      .....
  path[sp--] = null;
      return shrt;
      .....
```

Med denna lite optimering i koden kommer vi att beräkna den kortaste vägen för städerna ovan för att kolla om vi får något förbättring.

Resultat

Från-Till	**Kortste resa(min)**	**Program svarar efter(ms)**
Malmö-Göteborg	153	0,066
Göteborg-Stockholm	211	0,20
Malmö-Stockholm	273	0.043
Stockholm-Sundsvall	327	1,6
Stockholm-Umeå	517	1,6
Göteborg-Sundsvall	538	1,1
Sundsvall-Umeå	190	2,4
Umeå-Göteborg	728	0,087
Göteborg-Umeå	728	1,1

Table 2: Tid för den kortaste resa, samt programmet svarstid

Resultat ovan visar att den optimering av koden som utfördes har förbättrat svarstid, då man jämför tabell 1 och 2 så syns det en markant minskning i svarstiden. Då optimering som utfördes gör att man inte hamnar i en oändlig loop och då kanske man behöver inte ha ett max värde längre. För vi ska kunna få en inblick över vilket funktion maxvärdet har, så kommer vi att ha ett hög max värde och sen beräkan den kortaste vägen mellan Malmö och kiruna. Samt kommer vi efter att sätta max värdet till null och använda oss av en hittad väg, alltså med hjälp av de direkt riktade vägar, resultatet blir:

Resultat

Från-Till	**Kortste resa(min)**	**Program svarar efter(ms)**
———	**max värde (10000)**	**max värde (null)**
Malmö-Kiruna	1162	99
Malmö-Kiruna	1162	66

Table 3: Tid för den kortaste resa, samt programmet svarstid

Tabellen ovan visar att när vi har max värdet satt till null så får man svar snabbare jämfört om max värdet är hög.

Annat att fundera över

Lösning vi har ovan är en bra lösning för små begränsad kartor, däremot vid sökning efter den kortaste väg för stora kartor så blir den ganska oeffektiv. Google maps är ett exempel på en ganska snabb sökmotor, då vi kan söka efter den kortaste sträckan mellan en del olika städer som ligger väldigt långt bort från varandra såsom Malmö och Aten, en stad ligger i Sverige och den andra i Grekland, Google maps kunde ge svar på den kortaste tåg resan i ungefär 0,02 sekunder vilket är ungefär 20 ms, om vi jämför med koden ovan då det gav svar för sträckan mellan Malmö och Kiruna i 66 ms. Alltså den sista optimerade implmentering vi utförde är bra för vårt fall, dock kanske inte i det verkliga livet än.