Grafer

Mattias Sandberg

26 Okt 2022

Introduktion

I denna uppgift kommer vi att utforska den mer generella datastrukturen grafer. En graf är ett set av noder (hörn) som är ihopkopplade av kanter. Dessa kanter kan vara riktade vilket betyder att även fast A är kopplad till B betyder det inte att B är kopplad till A. Detta system är inte helt nytt då vi redan har jobbat med träd som är en typ av graf med riktade kanter och en rot. Även länkade listor är en graf men självklart av den enklare modellen. I denna uppgift ska vi utforska vad som händer när vi bortser från restriktioner med grafer och låter noder kopplas fritt till andra noder. Även fast detta är en liten ändring i reglerna kring grafer så kommer vi att se att det är extremt tidseffektivt.

Ett tåg från Malmö

För att ha något realistiskt att arbeta med kommer vi använda oss av järnvägssystemet i Sverige. Kartan vi kommer att använda har 52 städer och 75 anslutningar. Anslutningarna är dubbelriktade vilket betyder att man kan åka både fram och tillbaka mellan två städer. Den första uppgiften är att ta informationen till alla anslutningar och göra den till en graf som senare kommer att använda för att hitta den kortaste vägen mellan två olika städer i Sverige.

Grafen

Mappen jag kommer att använda är en csv fil som innehåller anslutningar i formen ”from,to,minutes”. För att representera grafen kommer vi att representera en stad som en struktur med ett namn och det grannstäder som representeras som en array hållande direktanslutningar. För att göra detta så kodar vi två klasser som jag döper till City och Connection där City klassen har en publik metod som adderar nya anslutningar till staden. Metoden ska ta destinationen för staden och distansen som argument. Vår karta kommer att vara en samling av städerna och vi behöver en snabb ”lookup” metod.

Denna lookup metoden används enbart när vi adderar en nya städer till vår karta och när vi korsar kartan för att exempel hitta specifika noder som representerar städer. Poängen med detta är att när vi hittar rätt nod finns all information som vi är ute efter given av City objekt. Nedan kommer ett utdrag från början på min ”lookup” metod.

```
public City lookup(String name){
    Integer index = hash(name);

    while(true) {
        if(cities[index] == null) {
          break;
        }
        if (cities[index].name.equals(name))
            return cities[index];

            index = index + 1 % mod;
    }
    City city = new City(name);
    cities[index] = city;
}
```

Hash till vår räddning

Nu ska vi implementera en hash tabell. Om jag skapar en hash funktion som tar en sträng som argument kan jag skapa ett hash värde genom att addera ihop tecken. När jag väljer mod värde bestämmer jag ett värde som är tillräckligt stort för att tillåta våra städer att spridas ut i en array men fortfarande tillräckligt liten för att inte slösa utrymme. Enligt uppgiften väljer vi 514 vilket är rejält med marginal och kommer låta våra 51 städer få mycket utrymme. Men givetvis kommer vi fortfarande få några kollisioner som behöver kunna hanteras.

Kartan

Genom att implementera följande metoderna ”lookup” och ”hash” och låta ”lookup” metoden returnera staden och namnet om den är befintlig i arrayen eller skapa den som en ny stad om dess namn inte känns igen. För varje rad i kartan ska två städer anslutas till varandra. När alla rader har hanterats i csv filen är kartan redo att användas.

Kortaste vägen från A till B

Uppgiften är nu att implementera ett program som hittar den kortaste vägen mellan två städer. Till en början kommer jag att ignorera hur vägen ser ut

utan bara fokusera på den kortaste vägen och hur många minuter resan kommer att ta. Till en början kommer implementationen vara simpel men med många begränsningar och sedan kommer vi försöka att identifiera problemen och lösa dom.

Djupet först

Strategin för att skapa denna implementation är att använda en djupsökning för den kortaste vägen. Jag skapar en fil Navie.java och lägger till en main metod från PDF: en.

Ett maxdjup

Maxvärdet är satt för att förhindra att sökningen upprepar sig i en oändlig loop. Till exempel om vi ska hitta den kortaste vägen mellan Malmö och Göteborg, måste tiden vara mindre än 300 minuter. Även fast algoritmen kommer att försöka gå fram och tillbaka till lund flera gånger kommer den att tappa 26 minuter varje gång. Efter 11 resor kommer tiden att vara ute och då måste algoritmen försöka ta någon annan väg. På detta sätt kan vi jobba runt detta problem med oändliga loopar. Nedan kommer kodsektioner från Naive programmet:

```
public class Naive{
  private static Integer shortest(City from, City to, Integer max){
    if(max < 0){
      return null;
    }
    if(from == to){
      return 0;
    }
    Integer short = null;

    for(int i = 0; i < from.neighbors.length; i++){
      if(from.neighbors[i] != null) {
        Connection conn = from.neighbors[i];
        Integer dist = shortest(conn.city, to, max - conn.dist);
        if((dist != null) && ((short == null) || (short > dist + conn.dist)))
            short = dist + conn.dist;
        if((short != null) && (max > short)){
          max = short;
        }
      }
    }
    return short
```

Några benchmarks

Det är nu dags för att göra några benchmark där mitt program försöker att hitta den kortaste vägen mellan olika städer. Det som är viktigt är att ge ett tillräckligt högt max-värde för att operationen ska kunna utföras. I grafen nedan representerar jag den kortaste vägen mellan två städer och hur lång tid resan tar samt hur lång tid det tog att exekverar programmet:

Städer	**Restid (min)**	**Exekveringstid (milisekunder)**
Malmö till Helsingborg	48	0
Malmö till Göteborg	153	2
Malmö till Stockholm	273	4
Malmö till Gävle	383	45
Malmö till Umeå	Interrupt	Interrupt
Umeå till Malmö	790	7

Table 1: Tabell över benchmark från Navie

När jag försökte beräkna den kortaste vägen och hur lång tid det tar att resa mellan Malmö till Umeå så stod programmet bara och tuggade men problemet är inte koden utan tidskomplexiteten. Det gick dock bra att hitta den kortaste vägen mellan Umeå och Malmö vilket betyder denna i grafimplementation och grafteori i säg finns det skillnader beroende på åt vilket håll i grafen man går och detta kan få tidskomplexiteten att skjuta i taket vilket jag bedömer att denna uppgift vill visa och att man ska förstå detta.

Upptäcka slingor

Eftersom vi vet att loopar är en del av problemen så ska vi nu försöka att hitta ett sätt att undvika dom. Jag skapar ett nytt program ”Paths.java” som är en kopia av mitt föregående program och låter denna version ha en array som heter ”path” som är tillräckligt stor för att hålla alla vägar. När jag börjar gå ned i grafen för att söka efter den önskvärda vägen, kollar jag först att min stad inte är i min väg. Och om detta skulle vara fallet så avbryter jag sökningen och returnerar null. Nedan kommer kodsektioner från implementationen:

```
public class Paths{
  City[] path;
  int sp;

  public Paths(){
    path = new City[54];
```

```
    sp = 0;
  }
  private Integer shortest(City from, City to, Integer max){
    if((max != null) && (max < 0)){
      return null;
    }
    if(from == to){
      return 0;
    }
    for(int i = 0; i < sp; i++){
      if(path[i] == from)
        return null;
    }
  }
}
```

I denna uppgift kommer vi inte strikt behöva något max-värde eftersom vi inte kommer att hamna i någon oändlig loop men jag har kvar den funktionen då den ändå kan vara hjälpsam. Nu ska jag genomföra nya benchmark med denna nya implementation och jag kommer kunna använda väldigt höga max-värden. Enligt resultatet nedan ser vi att det går alldeles bra att hitta den kortaste vägen från Malmö till Umeå vilket betyder att implementationen är lyckad och att detta tänkesätt med rätt exekvering uppnår ett bra resultat:

Städer	**Restid (min)**	**Exekveringstid (microsekunder)**
Malmö till Gävle	383	110267
Malmö till Umeå	790	376738

Table 2: Tabell över benchmark från Paths

En förbättring som jag kan göra är att använda en hittat väg och sätta en begränsning för framtiden. Detta betyder att om tex A är direkt ansluten till B, C och D och jag försöker att hitta vägen från B till 0 vilket är 460 minuter. Om då avståndet mellan A och B är 60, är en annan väg från A till O kortare än 520 minuter. Om jag gör dessa uppdateringar för max-värdet när jag testar det olika direktanslutningarna så kommer jag inte behöva något initialt max-värde, vilket betyder att den kan vara null tills programmet hittar en väg. Denna nya förbättrade implementationen ska nu benchmarkas och jag kommer bland annat att hitta den kortaste vägen mellan Malmö till Kiruna. Enligt resultaten nedan ser vi att tidseffektiviteten drastiskt har förbättrats och implementationen var gynnsam:

Städer	**Restid (min)**	**Exekveringstid (microsekunder)**
Malmö till Umeå	790	19753
Malmö till Kiruna	1162	91584
Kiruna till Malmö	1162	1384
Kiruna till Stockholm	889	49
Kiruna till Jönköping	1068	913
Kiruna till Halmstad	1145	1368
Kiruna till Lund	1149	1572

Table 3: Tabell över benchmark från förbättrad Paths

Saker att fundera över

Även fast jag har kodat en lösning som fungerar bra för detta exempel så ser vi enligt grafen ovan att tidskomplexiteten är horribel och att exekveringstiden sticker iväg väldigt snabbt i förhållande till avstånd mellan städerna. Den kortaste vägen från Malmö till Athen är enligt Googel maps via kollektivtrafik (Tåg och buss) cirka 3500 timmar. Om det skulle vara möjligt med min implementation att hitta den kortaste vägen skulle det ta extremt lång tid att beräkna den kortaste vägen då det finns så otroligt många olika anslutningar av järnväg i Europa. Problemet med min lösning är att den inte kommer ihåg någon information. När vi söker för den kortaste vägen mellan Malmö och Kiruna kommer algoritmen någon gång att gå förbi Stockholm. Den kommer sedan att hitta den kortaste vägen från Stockholm till Kiruna och använda den data för att beräkna den kortaste vägen från Södertälje till Kiruna. När vi använder denna process är det väldigt värdefullt att komma ihåg den bästa vägen från Stockholm till Kiruna. Om programmet inte kommer ihåg det så kommer vi behöva utforska alla vägar från Stockholm igen.

Om vi kollar på en karta med järnväg som ansluter n städer där varje stad i snitt har två direktanslutningar till andra noder. Den kortaste vägen mellan två städer kan då estimeras grovt till sqrt(n). Med denna minnesimplementation förbättras tidskomplexiteten drastiskt och enligt diskret matematik och dess grafteori så är Tidskomplexiteten för en grafimplementation Ordo O(V+E) där v är antalet hörn i grafen och e antalet kanter i grafen.