Assignment 9 - Grafer

Jesper Hesselgren

Oktober 24, 2024

Introduktion

I den här rapporten ska vi utforska datastrukturen grafer. En graf består i korthet av en mängd noder som kopplas samman med ett antal bågar, vilka kan bilda vägar mellan noder som inte är direkt sammankopplade. Tidigare har vi studerat länkade listor och binära träd, som också är typer av grafer men med vissa restriktioner. I denna rapport ska vi istället undersöka en graf utan några restriktioner. För att konkretisera detta kommer vi att analysera ett verkligt problem genom att använda oss av järnvägsnätverket i Sverige. Här representeras svenska städer som noder och tåglinjer mellan dem som kanter i grafen. Vår uppgift blir att så snabbt som möjligt hitta den kortaste vägen mellan två städer.

Byggandet av våran graf

Vi har en CSV-fil med 75 rader som vi ska använda för att bygga våran graf. Varje rad i filen består av en "sträcka" som innehåller information om en avresestad, en destinationsstad och tidsåtgången mellan dem. Graf ska var dubbelriktad, vilket innebär att kan vi åka från Stockholm till Södertälje på 21 minuter, ska vi kunna åka åt motsatt håll på samma tid. Nedan kan vi se ett exempel på en rad i CSV-filen.

```
Stockholm,Södertälje,21
```

Vi representerar våra städer och tåglinjer med hjälp av två publika klasser, City och Connection. City-klassen representerar själva staden och har två attribut: ett namn (en String) och en ArrayList av Connection, som innehåller alla tåglinjer (anslutningar) som staden är kopplad till. För att hantera dessa anslutningar har City också metoden addConnection, som låter oss lägga till nya tåglinjer. Metoden tar en annan stad (en City) och tidsåtgången (en int) som parametrar och lägger till en ny Connection i stadens ArrayList. Denna metoden kommer användas när vi ska bygga våran graf senare.

Connection-klassen representerar varje enskild tåglinje mellan två städer och innehåller information om destinationsstaden (en City) och tidsåtgången (en int) för att nå den. Eftersom grafen är dubbelriktad, innebär varje anslutning mellan två städer att en motsvarande Connection skapas i båda riktningar, vilket vi kommer se när vi bygger våran graf.

Kartan över järnverksnätverket- grafen

Nu när vi har ett sätt att repressentera alla våra städer och förbindelser kan vi bygga våran graf. Det gör vi genom klassen `Map` som innehåller en samling av städer. Map klassen är implementerad med hjälp av en hashtabell `map` och en lookup metod `lookUp` som används för att hämta/lägga till städer i hashtabell samt för att initsera sökningen av den kortaste vägen.

Hashtabellen `map` är implementerad med hjälp av buckets för att hantera eventuella kollisioner när vi placerar våra städer i hashtabellen. Varje bucket består av ett `City`-objekt (vår stad) och en referens, `next`, till nästa bucket om en kollision skulle uppstå. På så sätt kan flera städer lagras på samma position om de får samma hashvärde.

Metoden `lockUp` ansvarar för att lägga till eller hämta städer i hashtabellen. När en stad ska läggas till, beräknas först dess hashvärde, vilket avgör vilken plats i tabellen staden placeras på. Om en stad med samma namn redan finns på den platsen returneras istället staden, annars skapas en ny `City`-instans och placeras i en ny bucket på rätt index i hashtabellen.

Vid skapandet av ett `Map`-objekt läses data från en CSV-fil där varje rad representerar en förbindelse mellan två städer. `lockUp` används för att hämta eller skapa de två städerna från varje rad, och därefter skapas en dubbelriktad anslutning mellan dem genom att lägga till en ny `Connection` i respektive stads lista över anslutningar. Nedan kan vi se `map`-
klassens konstruktur där bygger våran graf.

```
public Map(String file) {

map = new Bucket[mod];
// Läser in filen
try (BufferedReader br = new BufferedReader(new FileReader(file))) {
    String line;
    while ((line = br.readLine()) != null) {

        // Läser in en rad i csv-filen och splittar den på ,
        String[] row = line.split(",");

        //Kontrollerar om staden finns. Annars skapas den
```

```
            City one = lockUp(row[0]);
            City two = lockUp(row[1]);
            int distance = Integer.parseInt(row[2]);

            //Lägger till connection åt båda håll
            one.addConnection(two, distance);
            two.addConnection(one, distance);
        }
    } catch (Exception e) {
        System.out.println("file" + file + " not found or corrupt");
    }
    }
```

Kortaste vägen mellan A och B

Vi ska nu titta på två olika sätt att hitta den kortaste vägen mellan två städer, `Naive` och `Paths`. Båda metoderna använder en rekursiv djupet-först-sökning via en metod, `shortest`. Den mer grundläggande `Naive`-metoden saknar loop-detektering, medan `Paths` förbättrar sökningen genom att undvika "oänldiga" loopar.

Naive metoden

Naive-metoden söker efter den kortaste vägen mellan två städer genom att utforska alla möjliga förbindelser med rekursiva anrop av metoden `shortest`. Metoden-`shortest` tar emot tre argument `from`, `to` och `max`, tiden det som längst får ta mellan städerna. För varje förbindelse från startstaden till en angränsande stad minskar den tillåtna restid, `max`, med tidsåtgången för den aktuella förbindelsen. Om restiden tillslut understiger noll, avslutas sökningen längs den vägen eller om vägen tillslut når att `from` == `to` returneras tiden tillbaka .

Paths metoden

Paths-metoden förbättrar Naive-metoden genom att införa loop-detektering. För varje steg i sökningen håller metoden koll på vilka städer som redan har besökts längs den aktuella sökvägen, genom en stack med `City`-objekt. Om en stad redan finns i den nuvarande sökvägen, avbryts den grenen av sökningen för att undvika oändliga loopar. Detta gör sökningen betydligt mer effektiv, då den undviker att undersöka samma väg flera gånger. I och med att vi underviker oänliga loopar behöver vi inte något `max` argument till våran `shortest`-metod.

Nedan kan vi se lite mätningar av dem olika metoderna.

Malmö-Göteborg	0 ms
Göteborg-Stockholm	1 ms
Malmö-Stockholm	2 ms
Stockholm-Sundsvall	19 ms
Stockholm-Umeå	1108 ms
Göteborg-Sundsvall	1507 ms
Sundsvall-Umeå	0 ms
Umeå-Göteborg	0 ms
Göteborg-Umeå	Gave up

Tabell 1: Mätning med Naive

Malmö-Göteborg	79 ms
Göteborg-Stockholm	38 ms
Malmö-Stockholm	70 ms
Stockholm-Sundsvall	59 ms
Stockholm-Umeå	80 ms
Göteborg-Sundsvall	72 ms
Sundsvall-Umeå	187 ms
Umeå-Göteborg	71 ms
Göteborg-Umeå	102 ms

Tabell 2: Mätning med Paths

Sammanfattning/Slutsats

`Naive`-metoden, som saknar loop-detektering, presterar bra för kortare sträckor men har svårigheter vid längre och mer komplexa sökningar. Vi ser till exempel att sökningen för vissa längre rutter, såsom "Göteborg-Umeå," tar så lång tid att vi måste avbryta sökningen ("Gave up"). De långa söktiderna beror på att metoden kan fastna i cykler och undersöka samma vägar flera gånger. De extremt korta eller omedelbara söktiderna för vissa rutter beror på att vi känner till den exakta tiden mellan dessa städer och sätter maxvärdet `max` till denna tid.

`Paths`-metoden, däremot, förbättrar sökningen genom loop-detektering, vilket gör den mer konsekvent och effektiv för längre sträckor. Medan kortare sträckor tar något längre tid än med `Naive`-metoden, så ger `Paths`-metoden betydligt bättre prestanda för längre sträckor. Vi kan se att sökningen mellan exempelvis "Stockholm-Umeå" och "Göteborg-Umeå" är mycket snabbare och undviker de problem som `Naive`-metoden stöter på.

Sammanfattningsvis visar mätningarna att `Paths`-metoden är väl lämpad för större kartor och längre rutter, där loop-detektering kan minska den totala söktiden avsevärt.