Uppgift 11

Simon Moradbakti

HT 2024

Introduktion

I denna sista rapport ska vi utforska en datastruktur som kallas för graph. Precis som det låter är det alltså noder som är kopplade till varandra med så kallade edges, som man kan ha lärt sig från diskret matematik kursen. Man kan också ha lärt sig om grafer från föregående uppgift om träd, som är en typ av graf. Detta gäller också länkad lista. För att gå från en nod till en annan indirekt kan man använda sig av paths.

A train from Malmö

I denna deluppgift skulle man implementera ett järnvägsnätverkssystem mellan 52 olika svenska städer som sammanlagt ska ha 75 stycken kopplingar eller edges. En koppling går åt båda hållen. En koppling mellan Stockholm och Malmö kan alltså gå från vilket håll som helst. Det första vi skulle göra var att beskriva våra kopplingar och noder och göra detta till en graf. Detta för att hitta den kortaste vägen mellan olika städer i landet.

the graph

För att göra vår graph måste vi ha information om vilka 52 städer det handlar om. Lyckligtvis har man fått en cvs fil med alla städer och kopplingar mellan dem som man kan göra en graf av. Kopplingarna beskrivs som hur lång tid det tar att åka från en destination till en annan i minuter. Självaste grafen designas genom att använda oss av 2 olika klasser, givet i instruktionerna. Dessa är City och Connections. Objektet City har ett namn som en sträng och grannar som är connections i form av en array, det vill säga en array med connections. Klassen Connections har ett objekt som heter Connections, som tar emot en stad och ett avstånd som parametrar. Våra connections ger alltså namnet på staden och hur lång tid det tar att åka till respektive stad. Vi måste även lägga till en metod som lägger till nya connections till en stad. Vi tar en destination som argument som är en av våra 52 städer och hur lång tid det tar att ta sig dit. Vi stoppar sedan in det i vår array med samma arrayer som också leder till samma stad. Kodmässigt som implementerades lösningen såhär:

```
public void addConnection(City destination, Integer distance
    ) {
    Connection connection = new Connection(destination,
        distance);
        for (int i = 0; i < neighbors.length; i++) {
        if (neighbors[i] == null) {
            neighbors[i] = connection;
            return;
        }
    }
}
```

hashing to our rescue

I en tidigare uppgift har vi faktiskt använt oss av hash funktioner. Det man fick då såg ungefär ut såhär:

```
private int hash(String name) {
    int hash = 7;
    for (int i = 0; i < name.length(); i++) {
        hash = (hash * 31 % MOD) + name.charAt(i);
    }
    return hash % MOD;
}
```

Primtal ger bättrehashing utan kollisioner som vi gjorde som konstaterande i den uppgiften.

the map

Vi har fått en hel del kod att gå efter när vi ska bygga våran map. Det enda som fattas att göra är att göra en metod som kallas lookup. Hash funktionen är även i mapen. Lookup metoden har som uppgift att returnera namnet på vår City om det existerar, om det inte gör det ska det skapa det. Vi gör sedan en lookup mellan 2 Cities och gör en connection mellan dem. Detta kan vi använda för att göra den kortaste vägen. Kodmässigt blir det såhär:

```
public City lookup(String name) {
    int index = hash(name);
    for (int i = 0; i < cities.length; i++) {
        int currentIndex = (index + i) % cities.length;
        if (cities[currentIndex] == null) {
            cities[currentIndex] = new City(name);
            return cities[currentIndex];
        }
```

```
        else if (cities[currentIndex].name.equals(name)) {
            return cities[currentIndex];
        }
    }
    return null;
}
```

Shortest path from A to B

Här kommer olika implementeringar för att hitta den kortaste vägen. I denna uppgift genomförde vi en djup först-sökning (DFS) för att hitta den kortaste vägen. Denna implementering är naiv, då DFS inte garanterar att en nod (låt oss kalla den nod 1) besöks före en annan nod (nod 2) när vi startar från startnoden. Detta betyder att nod 1 inte nödvändigtvis är närmare källan. När vi hittar en den kortaste vägen, finns det ingen garanti för att detta verkligen är den kortaste vägen. Vid användning av DFS utforskar vi även alla möjliga vägar innan vi återvänder med den kortaste. Ibland kan vi till och med fastna i en oändlig loop.

a max depth

Det kan vi lösa såhär. Vi använder oss av ett maxvärde för att förhindra att sökningen av en väg leder till en oändlig loop. Till exempel, när vi vill veta vilken den kortaste vägen är från Stockholm till en annan stad, sätter vi ett maxvärde som motsvarar den tid det tar att komma dit. Vi sätter maxvärdet till något värde i minuter. Så vi vill hitta den kortaste vägen, men den måste vara mindre än det värdet i minuter. Om det tar längre tid än så letar vi efter en annan väg.

benchmarks

Max	**från**	**till**	**tid för att hitta (ms)**	**tid (min)**
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	10	273
400	Stockholm	Sundsvall	269	327
600	Stockholm	Umeå	558636	517
600	Göteborg	Sundsvall	436585	515
200	Sundsvall	Umeå	0	190
800	Umeå	Göteborg	27	705
800	Göteborg	Umeå	1	705

Table 1: Benchmark för Naive

Den naiva lösningen var inte speciellt bra. Då den kan fastna i loopar för evigt. Den måste också kontrollera alla noder för att hitta den bästa vägen. Det tar väldigt väldigt lång tid, så pass lång tid att man kanske inte har tålamod nog att vänta.

Max	**från**	**till**	**tid för att hitta (ms)**	**tid (min)**
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	0	273
400	Stockholm	Sundsvall	3	327
600	Stockholm	Umeå	10	517
600	Göteborg	Sundsvall	4	515
200	Sundvall	Umeå	0	190
800	Umeå	Göteborg	1	705
10 000	Malmö	Kiruna	231	1162

Table 2: Benchmark för path

Detta blev bättre då vi kan avsluta en sökning när vi vet att dne inte är kortare än övriga i våra arrayer. Det sparar tid och datakraft. Vi behöver egentligen inte häller max värden då vi aldrig kommer hamna i någon loop här. Denna sista förbättring är maginellet bättre än den föregående. Här väljer vi

Max	**från**	**till**	**tid för att hitta (ms)**	**tid (min)**
200	Malmö	Göteborg	0	153
300	Göteborg	Stockholm	25	211
300	Malmö	Stockholm	0	273
400	Stockholm	Sundsvall	2	327
600	Stockholm	Umeå	12	517
600	Göteborg	Sundsvall	5	515
200	Sundvall	Umeå	0	190
800	Umeå	Göteborg	1	705
800	Göteborg	Umeå	50	705
10 000	Malmö	Kiruna	200	1162

Table 3: Benchmark för path slightly improved

ett max värde och hittar en path. Så om vi hittar en väg från A till B som tar 30 minuter och från B till Z som tar 200 minuter, vet vi att om A är kopplad till flera andra noder som C och D, måste den totala tiden vara kortare än 230 minuter.