TDDC91,TDDE22,725G97 Lektion 3

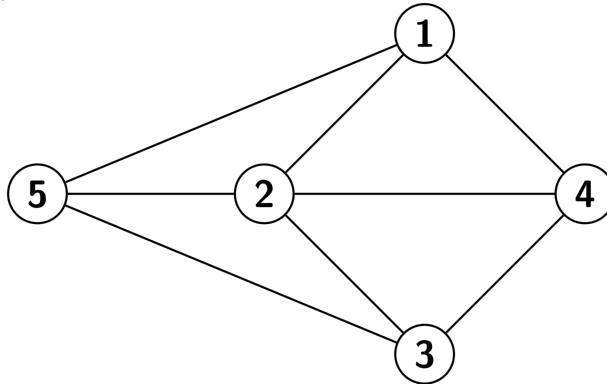
Grafer och Lab4

Magnus Nielsen
magnus.nielsen@liu.se

2 oktober 2018

Enkel oriktad graf

Rita en enkel graf, med vägar av olika grad från ”start-nod” till ”slut-nod”.
Ex:

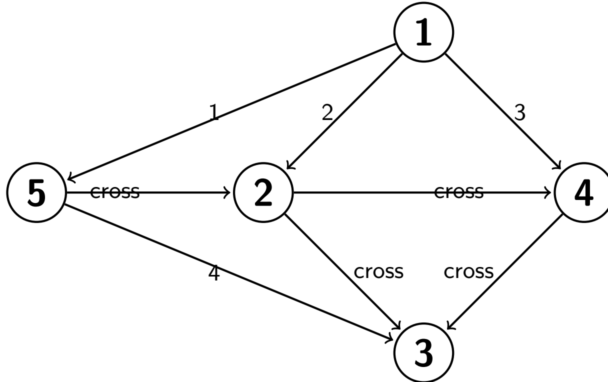


Vi har flera vägar från 1 till 5. Fråga studenterna vilka vägar de kan se. (1-5, 1-2-5, 1-4-2-5, 1-4-3-5, 1-2-3-5).

Förklara, med hjälp av grafen vi nu har ritat, vad kortaste vägen innebär (den väg som traverserar minsta antalet noder) till en slut-nod. 1-5 om vi använder oss av exemplet, alltså inte 1-2-5 till exempel, även om det är en möjlig väg.

Traversera en enkel graf

Använd sagda graf för att, med hjälp av studenterna, genomföra en bredden-först sökning.



Vad är kortaste längsta vägen

Längsta vägen i grafen bör vara 1-5-3. Detta är också kortaste vägen 1-3. Vi skulle kunna ta vägen 1-2-3 (likvärdig, men ogiltig med bredden först) eller 1-2-4-3 exempelvis.

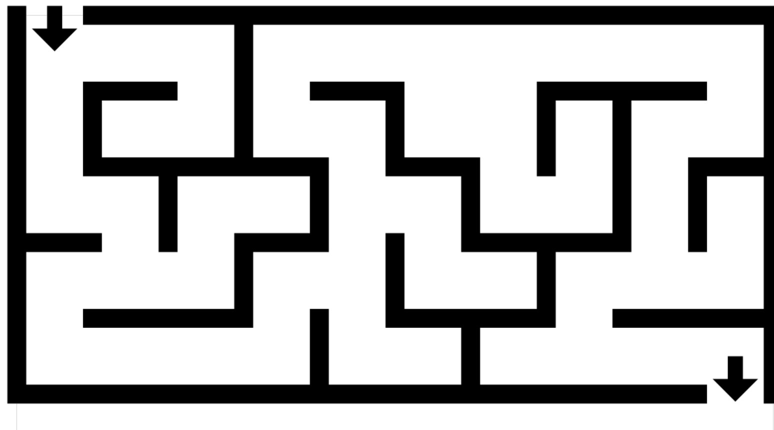
Om vi istället använder 5 som startpunkt (det finns inte nödvändigtvis någon ”start-nod” i en graf) får vi en helt annan längsta väg (5-3-4). Med bredden först sökning kommer det alltid vara sista noden vi anländer till, och vägen är de noder vi traverserar för att komma dit. Beroende på vilken nod vi använder som utgångspunkt kommer resultatet variera. Den kortaste längsta vägen är alltså det kortaste resultatet av bredden först sökning med samtliga startnoder som utgångspunkt!

Praktiska exempel - Kortaste längsta vägen

- Vi tittar på lokaltrafiken i Linköping: vi vill göra en resa från någon slutstation till någon annan slutstation med så få byten och kursändringar som möjligt (kortaste möjliga färdtid, men längsta möjliga sträcka). Bussnätet är vår graf. Jämför gärna med vad vi får med djupet först sökning (vi får en möjlig route till en slutpunkt, men inte garanterat en slutstation och med hög sannolikhet (beroende på grafens utseende) många onödiga byten.
- Vi har en labyrint med flera utgångar. Samtliga utgångar kan även användas som ingångar, men vi får bara gå ut ur labyrinten i den utgång som ligger längst bort från den ingång vi använt, ty resten är vaktade av monster. Vi vill ta reda på vilken ingång vi ska använda för att ha så kort väg som möjligt genom labyrinten.

Ibland måste det inte vara en graf

Många problem kan representeras som en graf för att göra problemet hanterbart i en dator.
Exempel:



Vi har inte en konkret graf, men vi kan konceptuellt föreställa oss labyrinten som en graf där varje korsning är en nod.
Traversera labyrinten tillsammans med studenterna (bredden först). Vi behöver inte traversera hela, bara några steg tills det blir tydligt för studenterna att det faktiskt är samma sak. Gör gärna tydligt vilka ”noder” som ligger i samma ”nivå” av traverseringen (tänk level-order traversering av träd, eller den övertydliga versionen av graf vi använde på tavlan).

Optimering av kod

Hur kan vi förbättra kod generellt?

- Byta representationer (datatyper / klasser som passar syftet bättre)
- Analysera och förbättra algoritmen

Labb 4

- Labb 4 är en optimeringslaboration. Problemet är redan löst - vi måste optimera den existerande lösningen.
- Körtiden måste förbättras ca 10000 gånger.
- Förklara hur allt vi talat om relaterar till labben.
  Använd till exempel (från labbhäftet):
  Sökord: aula labb
  $aula- > gula- > gala- > gama- > jama- > jamb- > jabb- > labb$
  Kedjan är 8 ord. Vi tar oss ett steg i taget genom att byta ett tecken, varje nytt ord måste finnas i ordlistan. Alltså: Varje ord är en nod, och bågarna som ansluter dem är faktumet att man kan nå den nya noden genom att byta ett tecken.
- String är inte muterbar. Skriv upp följande exempel

```
// Del 1: Diskutera vad resultatet är. Förklara vad strängkonkatenering (concat) är vid behov.
String s1 = "my";
s1.concat(" string");
System.out.println(s1);

// Del2: Vi skapar ett helt nytt strängobjekt och tilldelar s1 den nya referensen.
s1 = s1.concat(" string");
System.out.println(s1);

// Del3: Låt studenterna få några minuter att diskutera hur många strängar som skapas
// i tilldelningen nedan (5). Låt dem komma med förslag i grupp, förklara om ingen
// av dem kommer fram till rätt svar (samt vad som skrivs ut).
String s2 = s1.substring(0,1) + "agic" + s1.substring(1,2);
System.out.println(s2);
```

Case 1

Optimera hashtabellen

```
public class HashTable {
    private static final int size = 10;
    private MyQueue<Image>[] hashtable;

    public HashTable() {
        hashtable = new MyQueue<Image>[size];
        for (int i = 0; i<size; i++) {
            hashtable[i] = new MyQueue<Image>();
        }
    }

    public void put(Image newImage) {
        hashtable[hash(newImage.getName())].enqueue(newImage);
    }

    public Image get(String name) {
        Image res;
        Image first = hashtable[hash(name)].dequeue();
        hashtable[hash(name)].enqueue(first);
        while (!hashtable[hash(name)].isEmpty()) {
            Image current = hashtable[hash(name)].dequeue();
            hashtable[hash(name)].enqueue(current);
            if (current.getName().equals(name)) {
                res = current;
                break;
            }
            else if (current == first) {
                res = null;
                break;
            }
        }
        return res;
    }

    private int hash(String key) {
        int i;
        int v = 0;

        for (i = 0; i < key.length(); i++) {
            v += key.charAt(i);
        }
        return v % size;
    }
}
```

Ge gärna studenterna några minuter att diskutera i par eller tre och tre och se vilka optimeringar de kommer fram till. Diskutera i grupp.
Poängtera gärna dessutom att vi, i denna hashtabell, tappar insättningsordningen. Det kan vara relevant i många fall.

Svar till uppgifterna

- Uppgift 1
  Det finns inte en lösning på denna, utan ganska många olika. Ex:
  Sortering: Många kommer säkert att fokusera på sorteringen. Det är dock -mycket- svårt att få till en lika effektiv sorteringsalgoritm som en optimerad bubble-sort eftersom vi sorterar vid insättningsögonblicket och därmed bör aldrig behöva fler än två iterationer av den yttre loopen. Sorteringen kan dock vara relevant att se över om man ändrar till sortering vid get (sortera innan borttagning). Då sparar vi antalet körningar av sort, men kan behöva effektivisera algoritmen (vi måste fundera över hur vi ämnar använda klassen). Detta tjänar vi på om vi sätter in ofta, men tar bort sällan.
  Representation: Vi kan ändra representationen till, exempelvis, ett AVL-träd. Vi effektiviserar insättningen, men tappar när vi hämtar ”sista” elementet.
  Bonus: Om vi har dublettnycklar i listan kommer vi att tappa ordningen vid borttagning av sista elementet (remove tar bort first occurrence av ett element). I fallet med Integer går det bra, eftersom vi bara tittar på referensen, men om vi bygger ut det till att vi tittar på ett ID eller dylikt för ett mer avancerat objekt kommer vi ta bort det första objektet som matchar, istället för det sista.

- Uppgift 2
  Här finns det helt sjuka mängder möjligt arbete. Ex:
  Vi kan ändra representationen av trädet till en lista, array, etc.
  Vi kan göra trädet till självbalanserade (men då behöver vi lyfta ut funktionalitet från Node till Tree.
  Vi kan göra enkla optimeringar, som exempelvis lägga en räknare i Tree som uppdateras vid insättning och borttagning (då behöver vi bekräftelse från Node på att något tagits bort), istället för att rekursera genom hela trädet varje gång vi vill ha antalet element.
  Vi kan -enkelt- optimera remove, då den i nuläget kommer att öka höjden på trädet vid borttagning av noder med två barn.