Datorlaborationer, Objektorienterad modellering och design (EDAF25)

Datorlaborationerna ger exempel på tillämpningar av det material som behandlas under kursen.

- *Uppgifterna i laborationerna ska lösas i par om två.*
- *Laborationerna är obligatoriska.* Du måste bli godkänd på alla uppgifter för respektive laboration under ordinarie laborationstid. Om du skulle vara sjuk eller ha andra giltiga förhinder vid något laborationstillfälle så måste du anmäla detta till kursansvarig lärare före laborationen (epost-adress finns på kursens hemsida).

  Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den under nästa laborationstillfälle. Det kommer också att anordnas en uppsamlingslaboration i slutet av kursen som erbjuds till dem som haft giltiga skäl för frånvaro på någon laboration eller som varit närvarande men, trots rimliga förberedelser, inte hunnit bli färdig..
- *Laborationerna kräver förberedelser.* I början av varje laboration finns anvisningar om förberedelser under rubrikerna `Läsanvisningar` och `Förberedelser`. Läsanvisningarna anger vilka avsnitt i läroboken, eller i annat material, som är lämpliga att läsa. Under rubriken `Förberedelser` anges vilka av laborationsuppgifterna som *ska* lösas före laborationstillfället. Varje laborationspar ska ha läst igenom de övriga uppgifterna och gärna försökt lösa dem. Det är inget krav att ni kommer med helt färdiga lösningar, men det är erat ansvar att ha förberett er så att ni bedömer att ni hinner bli klara under laborationen. Ni får naturligtvis under dessa förberedelser gärna kontakta kursansvarig lärare om ni stöter på svårigheter.

I källkoden till labbarna används engelska termer för grafbegrepp, t.ex. bågarna i en graf kallas “edges”, en nod kallas “vertex” och grannar kallas “neighbours”.

Innehåll

Förberedelser

Källkod och projektfiler som behövs för att påbörja laborationsuppgifterna finns i den komprimerade projektkatalogen `edaf25-labs.zip` vilken finns att hämta från kursens Canvas-sida.[1] Ladda ner filen `edaf25-labs.zip` och packa upp den på lämplig plats inuti din hemkatalog.

Projektkatalogen innehåller följande viktiga filer/mappar:

- `src/main/java/graph` – mapp som innehåller färdigimplementerade klasser för att representera grafer i Java. Dessa behöver inte ändras men du bör läsa igenom filerna och lära dig hur det fungerar.
- `src/main/java/lab1` osv. – uppgifter att lösas under lab1, lab2, osv.
- `src/test/java/lab1` osv. – test som ska bli gröna när respektive laborationsuppgifter har lösts.
- `gradlew` – skript för att bygga projektet i Unix-miljö (Mac/Linux)
- `gradlew.bat` – batch-fil för att bygga projektet i Windows-mijö
- `build.gradle` – konfiguration för byggsystemet
- `*.txt` – testfiler som används i de senare labbarna

För att arbeta med projektet behövs en programutvecklingsmiljö som Visual Studio Code, Eclipse, eller IntelliJ. Det rekommenderas att använda en av dessa grafiska utvecklingsmiljöer, men för de som vill går det att använda kommandoraden också med hjälp av skriptet `./gradlew`.

Nedan följer instruktioner för er som använder ett av de rekommenderade grafiska utvecklingsmiljöerna. *OBS!* Om du väljer att använda skolans datorer, välj Eclipse!

Eclipse

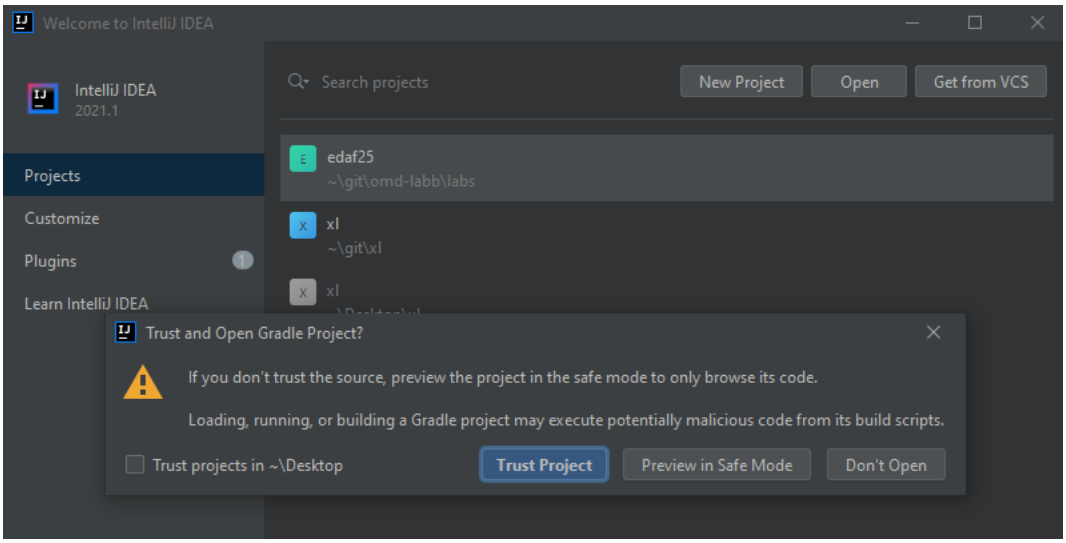
På skolans datorer finns Eclipse installerat. Om du vill jobba med laborationen hemma och inte redan laddat ner Eclipse så finns anvisningar för detta på hemsidan för den första programmeringskursen, EDAA10 Programmering i Java: https://cs.lth.se/edaa10/java-och-eclipse-paa-egen-dator/

Om Eclipse frågar vilket "workspace" du vill använda så ska du välja att skapa ett nytt på godtycklig plats (dock ej inuti projektkatalogen du tidigare packat upp, dvs `edaf25-labs` eller bara `labs`). För att öppna projektkatalogen inuti Eclipse måste man från menyraden välja `File`, sedan `Import...` varefter det skall öppnas en dialogruta med många alternativ. Markera under "Gradle"-mappen alternativet "Existing Gradle Project" och klicka sedan på `Next` två gånger. När du ser en ruta som frågar efter "Project root directory", välj `Browse...` och välj katalogen som heter labs och som du packade upp tidigare. Beroende på vilket verktyg du använde när du packade upp zip-filen kan den ligga inuti en katalog som heter edaf25-labs. Välj *inte* katalogen edaf25-labs! Genom att clicka på `Finish` ska nu projektet öppnas i Eclipse och du bör se ett nytt projekt som heter `edaf25` i "Package Explorer" till vänster i Eclipse-fönstret. Du hittar källkodsfiler respektive JUnit-tester under src/main/java respektive src/test/java (gå inte ner under src).

[1] Canvas-sidan hittas från kurshemsidan `http://cs.lth.se/edaf25`

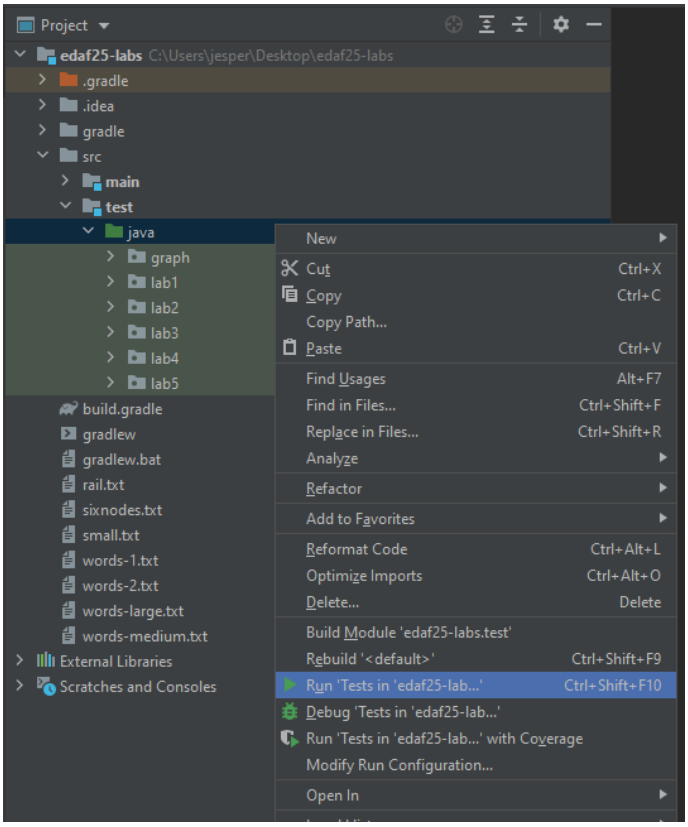
IntelliJ

Välj `File->Open...` i menyraden. Om det är första gången du startar IntelliJ så syns fönstret "Welcome to IntelliJ IDEA", clicka där på knappen `Open`.



I dialogrutan för att öppna projekt kan du navigera till den nedladdade projektkatalogen (katalogen labs, inte edaf25-labs om en sådan existerar) och markera den. IntelliJ bör automatiskt upptäcka att det är ett Gradle-projekt.

Första gången projektet öppnas i IntelliJ kan det ta lite tid för IntelliJ att bygga projektet, innan testen kan köras. Testen kan köras genom att högerklicka på en katalog under test-mappen och välja "Run tests in ...".



Visual Studio Code

Välj `File->Open Folder...` i menyraden inuti VS Code för att öppna den redan uppackade projektkatalogen `edaf25-labs`.

Om det är första gången du öppnar ett Java-projekt kommer VS Code fråga om du vill installera tillägget “Extension Pack for Java”. Det måste du installera för att kunna köra testen i labben. Följ stegen i “Getting Started” som visas för att slutföra installering av tillägget, sedan bör testen i projektkatalogen kunna köras via “Testing” fliken.

Laboration 1

Mål: Att få insikt i hur grafer kan representeras med klasser och objekt i Java. Laborationen ger också träning i att använda biblioteken `JUnit` och `Google Truth` för testning.

Läsanvisningar

Läs igenom avsnitt 10.1 till 10.3 i boken (Koffman, Wolfgang) och/eller föreläsningsbilderna. Vi kommer använda en lite annorlunda och delvis förenklad implementering av grafdatastrukturer än den som visas i boken men det är i huvudsak liknande design.

Förberedelser för den första laborationen

Läs igenom den inledande texten nedan under rubriken ”Datorarbete”. Lös uppgift D1 och D2. Läs igenom upggift D3 och D4.

Datorarbete

För att kunna implementera grafalgoritmer i Java behöver vi en datastruktur av Java-klasser som representerar grafen.

En graf är endast en samling noder och bågar. Noderna kan numreras och bågar representeras då enkelt som par av nod-nummer. Således finns en rimlig (men ineffektiv) design: att representera en graf med en lista av par av nod-nummer. Detta är ineffektivt eftersom det tar lång tid att ta reda på vilka grannar som någon nod u har: man måste söka igenom hela listan av bågar för att hitta alla bågar som utgår från nod u. Istället finns två mer effektiva sätt att spara en graf i en datastruktur: närhetslistor (engelska: *adjacency list*), och grannmatris (engelska: *adjacency matrix*). Dessa två sätt att representera grafer beskrivs i mer detalj i kursboken i avsnitt 10.3 och på föreläsningsbilderna.

Lösningen med närhetslistor använder mindre minne, i allmännhet, och det är lösningen som implementerats i `src/main/java/graph` i projektkatalogen. För att indexera rätt närhetslista används en `Map` från Javas standardbibliotek.

Vi använder i laborationerna gränssnittet `Graph` för att arbeta med grafer. Det representerar *riktade* grafer med ett fixt antal noder och bågar. Det är också ett generiskt gränssnit som kan ha noder av olika typer, inte bara heltal. Så här ser gränssnittet ut:

```
public interface Graph<T> {
  int vertexCount();
  Collection<T> vertexSet();
  Collection<T> neighbours(T v);
}
```

Metoden `vertexSet` ger mängden av noder i grafen, och `neighbours(v)` anger grannarna för någon nod `v`. Typparametern `T` anger typen på noderna i grafen. Vi kommer till att börja med bara använda heltal för att representera noderna, men i senare laborationer kommer vi att använda strängar som noder också. I denna laboration kommer vi att jobba med klassen `SimpleGraph` som implementerar detta gränssnitt med heltalsnoder, alltså `Graph<Integer>`. `SimpleGraph` tillåter ej duplicerade bågar eller bågar som går från en nod till sig själv. Läs igenom och bekanta dig med koden för `SimpleGraph`.

En ny graf av typen `SimpleGraph` skapas till exempel på följande sätt:

```
Graph<Integer> g = new SimpleGraph(3, new int[][] { {0,1}, {2,1} });
```

Det första argumentet till konstruktorn anger antalet noder, sedan följer en vektor som anger bågarna. Bågarna anges som par av heltal där det första talet anger startnoden och det andra anger slutnoden för bågen. Bågarna är i det här fallet från nod 0 till 1 och från nod 2 till 1.

Notera att `SimpleGraph` är icke muterbar – efter att en graf har byggts kan man inte lägga till bågar eller ändra antalet noder. Det är god praxis att använda icke muterbara objekt så mycket som möjligt. Om ett objekt är muterbart så är det möjligt att skriva metoder som förstör information i objektet vilket sedan kan ställa till problem för senare användningar av samma objekt.

Uppgifterna i denna labb ger övning i att implementera några mycket enkla grafalgoritmer som fungerar på alla objekt av typen `Graph` (därmed fungerar de för både `SimpleGraph` och `DistanceGraph` som vi kommer titta på i senare labbar). Ni kommer också att testa algoritmerna genom att skapa några grafer av typen `SimpleGraph` och kontrollera att era algoritmer ger rätt resultat för dessa.

D1. Läs koden i filen `src/test/java/graph/TestGraph.java`. Denna klass är en testklass som testar att metoderna `Graph.neighbours()` och `Graph.toString()` fungerar som de ska. Testklassen använder biblioteket JUnit som testramverk.[2]

Bekanta dig med din utvecklingsmiljö genom att köra testklassen `TestGraph`. Efter du har kört testen och sett att de fungerar, prova att ändra något i koden i `SimpleGraph` så att ett av testen misslyckas och kör sedan testklassen. Ångra din ändring och kör testen igen och se att testen i `TestGraph` lyckas.

D2. Slutför metoderna `vertexCount` och `edgeCount` i filen `src/main/java/lab1/Lab1.java` inuti projektkatalogen. Dokumentationskommentaren vid metodrubrikerna anger vad respektive metod skall göra.

Notera att metoden har deklarerats som `static` eftersom den inte behöver använda några attribut i den omgivande klassen.

Testa din lösning genom att köra testen i katalogen `src/test/java/lab1`. Testen i `TestVertexCount` och `TestEdgeCount` ska bli gröna. Korrigera eventuellt din kod och testa på nytt tills testen går igenom.

D3. Slutför metoden `edgeBetween` i klassen `Lab1` och se till att testen i klassen `TestEdgeCount` går igenom.

D4. Lägg till ett par nya test i klassen `TestEdgeBetween` och korrigera eventuellt din kod tills testen går igenom. Kan du komma på några intressanta specialfall som borde testas?

D5. Slutför metoden `buildGraph` i klassen `Lab1` och se till att testet i `TestBuildGraph` går igenom.

[2] Här används också ett bibliotek som heter Google Truth som tillhandahåller hjälpfunktionen `assertThat`.

Laboration 2

Mål: Att ge träning i att lösa olika problem med hjälp av djupet-först-genomgång av grafer.

Läsanvisningar

Läs igenom följande avsnitt i boken (Koffman, Wolfgang): 10.1 – 10.4 och föreläsningsbilderna. Under laborationen kommer vi att använda samma gränssnitt och klasser för grafer som i föregående laboration.

Förberedelser

Läs igenom den inledande texten nedan under rubriken ”Datorarbete”. Lös uppgifterna D1 och D2. Läs igenom uppgift D3. Läs texten i uppgift D4. Läs igenom uppgift D5.

Datorarbete

Under denna laboration skall vi behandla några grafproblem som alla har det gemensamt att de kan lösas genom att man traverserar grafen djupet först.

Djupetförstgenomgång utgående från en nod `v` kan implementeras som en rekursiv procedur.[3] I pseudokod kan det se ut så här (`dfs` är metodens namn och `v` är en nod i grafen):

```
dfs(v)
    markera v som besökt
    för varje granne u till v:
        om u ej är besökt:
            dfs(u)
```

Denna metod kommer anropas rekursivt på alla noder som går att nå, genom att följa en serie bågar, via den första noden som metoden anropas på. Det är dock inte säkert att alla noder i grafen kan nås på detta sätt eftersom grafen inte nödvändigtvis är sammanhängande. Dessutom kan det för en riktad graf vara möjligt att nå hela grafen från vissa noder men inte från andra. För enkelhets skull begränsar vi oss till oriktade grafer i denna laboration (dvs. en riktad graf där det för varje båge finns en annan båge i motsatt riktning). Om en oriktad graf är sammanhängande så kan man nå hela grafen från en godtycklig nod i grafen. Om den ej är sammanhängande kallas vare sammanhängande del i den för en komponent av grafen.

Några observationer om djupetförstgenomgång:

- Det är viktigt att markera behandlade noder som besökta eftersom det kan finnas cykler i grafen. Vi kan därför under rekursionen komma tillbaka till en nod som vi redan besökt. Om vi då inte kontrollerar detta utan bara gör ett rekursivt anrop så hamnar vi i en evig loop (som slutar med `StackOverflowError` i Java).
- Algoritmen ovan gör inget meningsfullt. Den visar bara hur en genomgång djupetförst går till i princip. Ni kommer under denna laboration att komplettera koden på olika sätt för att lösa flera olika grafproblem.

I projektkatalogen för laborationerna finns det en fil som heter `src/main/java/lab2/Lab2.java`. I denna finns en metod `dfs` som implementerar pseudokoden ovan, med följande metodrubrik:

```
static <T> void dfs(Graph<T> g, T u, Set<T> visited)
```

[3] Kan du någon annan princip för att implementera djupetförstgenomgång?

Studera koden innan du börjar lösa uppgifterna nedan. Lägg märke till att metoden har en typparameter `<T>`, så att metoden kan användas på grafer med vilken nodtyp som helst.

D1. Rita två oriktade grafer med 5 eller fler noder på papper: en sammanhängande och en icke sammanhängande. Översätt sedan de två graferna du ritat till kod i metoderna `buildConnected` och `buildDisconnected` i klassen `src/main/java/lab2/GraphFactory.java`. Kom ihåg att det måste finnas bågar i båda riktningar för varje oriktad båge!

Om du har gjort rätt ska testen i `TestGraphFactory` passera.

D2. Slutför metoden `isConnected` i klassen `Lab2`. Den ska returnera `true` om och endast om grafen som skickats in är sammanhängande (enligt definitionen ovan).

För att avgöra om en graf är sammanhängande räcker det att välja ut en godtycklig nod (`v`) och undersöka om det finns vägar från denna till alla andra noder i grafen.[4] Du kan lösa denna uppgift med hjälp av `dfs`-metoden som finns i samma klass (`Lab2`).

När du är klar ska testen i klassen `TestConnected` gå igenom.

D3. Slutför metoden `nbrOfComponents` i klassen `Lab2`. Den ska returnera antalet komponenter i grafen.[5]

När du är klar ska testen i klassen `TestNbrOfComponents` gå igenom.

D4. Slutför metoden `pathExists` i klassen `Lab2`. Den har följande metodrubrik:

```
public static void pathExists(Graph<Integer> g, int u, int v)
```

Den ska returnera `true` om och endast om nod v är nåbar utgående från nod u i graf g. Viktigt: här får du ej använda den färdigskrivna `dfs`-metoden – du ska istället skriva din egen metod som implementerar djupet-först sökning så att sökningen avbryts så fort som en väg från u till v har hittats! Tänk på att du kan utnyttja retur-värdet av din dfs-metod för att ta reda på om det är dags att avbryta sökningen.

När du är klar ska testen i klassen `TestPathExists` gå igenom.

D5. *Frivillig uppgift.* Antag att man inte bara är intresserad av om det finns en väg mellan två noder utan också hur en sådan väg ser ut. Vi kan t.ex. returnera en lista av de noder som finns på vägen mellan noderna. Om vi hittat en väg kommer alltså u att vara det första elementet i den returnerade listan och v det sista. Metoden kan ha följande rubrik:

```
public static List<Integer> findPath(Graph<Integer> g, int u, int v)
```

Denna metodrubrik finns redan i klassen `Lab2`. Implementera metoden. Tips: Använd en hjälpmetod som utför själva sökningen. Denna kan nu ha en extra parameter som är den lista man fyller på med noder efter hand som vägen konstrueras. Det kan vara praktiskt att låta hjälpmetoden returnera `true/false` för att kunna bryta så snart en väg hittats. När en nod besöks läggs den in i listan. Om man besöker alla grannar utan att hitta en väg tar man bort den igen.

[4] Försök att övertyga dig själv om att detta stämmer!
[5] Tips: Även här kan du använda dig av metoden `dfs`.

Laboration 3

Mål: Att förstå breddenförstsökning: hur det går till och dess tillämpningar i en graf. En implementation av ett ordspel ("Word Ladder") skall utföras.

Läsanvisningar

Läs följande avsnitt i läroboken: 10.4 (de delar som handlar om breddenförstgenomgång). Läs också föreläsningsbilder från de föreläsningar, som behandlar motsvarande avsnitt. Dessa finns på kursens hemsida.

Förberedelser

Läs igenom den inledande texten under rubriken ”Datorarbete” nedan och lös uppgifterna D1 – D2.

Datorarbete

Word ladders är ett spel där det gäller att hitta en kedja ord som binder samman ett startord med ett slutord, där varje intilliggande par av ord uppfyller ett visst kriterium. Till exempel, om kriteriet är att intilliggande ord skiljer sig på en bokstav och startordet är “varm” och slutordet är “kall” så blir följande ordkedja en giltig lösning:

varm → vars → vals → vall → kall

Problemet att hitta ordkjedjor kan modelleras och lösas med hjälp av grafer. För att modellera ordkjedjor med en graf så låter vi våra ord motsvaras av noder i grafen och lägger in bågar mellan varje par av ord som uppfyller kriteriet att de kan vara grannar i en ordkedja. Lösningen på pusslet är någon väg i grafen som går från det ena startordet till det andra.

Till bågarna knyts i detta fall inte någon data. Detta tillsammans med att noderna bara representerar strängar gör det möjligt att använda en `Map<String, Set<String»` för att lagra bågarna i grafen. Nycklarna i denna utgörs av orden i vår ordlista. Till varje ord associeras den mängd av ord (strängar) som är grannar till nyckelordet.

På tidigare föreläsning har visats hur man med hjälp av breddenförstsökning kan söka kortaste vägen mellan noder i en graf där alla bågar har samma vikt. Nedanför visas pseudokod för en variant av breddenförstsökning som beräknar kortaste avstånd från en nod `u` till en nod `v`:

```
distance = 0
markera u besökt
curLevel = tom mängd
lägg till u i curLevel
så länge curLevel inte är tom:
    nextLevel = tom mängd
    för varje nod w i curLevel:
        om w == v:
            return distance
        för varje granne n till w:
            om n inte är besökt:
              markera n besökt
              lägg till n i nextLevel
    distance = distance + 1
    curLevel = nextLevel
return -1
```

I denna variant väljer vi inte att (som boken) lägga alla noder i en och samma kö efterhand som vi besöker dem. Istället använder vi två mängder: en som består av de noder som finns på ett visst kortaste avstånd, `distance`, från utgångsnoden (`u`) och en för dem som befinner sig på avståndet distance+1 från denna. Den senare (`nextLevel`) byggs upp av ännu ej besökta grannar till den förra (`curLevel`) i loopen. I slutet av loopen låter vi sedan `nextLevel` bli den aktuella nivån och ökar `distance` med ett så att värdet motsvarar det avstånd noderna i `curLevel` ligger från utgångsnoden. Om vi använt en kö istället för två mängder så hade kön tidvis innehållit element på två olika nivåer (den aktuella och den som ligger på avstånd ett mer än den aktuella från utgångsnoden). Då hade vi behövt något sätt att hålla reda på de individuella köelementens avstånd. Genom att dela upp elementen i två olika samlingar slipper vi denna komplikation. Vi kunde ha valt två köer, men två mängder går lika bra eftersom det är likgiltigt i vilken ordning noderna på en viss nivå besöks.

Som framgår av koden ovan så behöver vi för varje nod som behandlas i grafen få tillgång till grannarna. Detta är den enda grafoperation som utförs. Om vi väljer att arbeta direkt med map-representationen av grafen så kan vi enkelt få tillgång till grannar genom att anropa `get()` på `Map`-objektet. Detta paketeras i en klass som implementerar `Graph`-gränssnittet, så att grannarna kan hämtas genom `Graph.neighbours()`.

D1. I projektkatalogen finns det en klass `WordGraph` (inuti `src/main/java/lab3`). Klassen implementerar `Graph<String>` och representerar en ordkjedje-graf enligt ovan med ett `Map` attribut.

Du ska börja med att implementera inläsning av en ordlista från en fil. Orden ska alltså läggas in i grafen. Du behöver inte skapa bågarna ännu, det ska göras i senare deluppgift!

Inläsningen av ordlista från fil skall ske i `WordGraph` konstruktorn. Sökvägen till filen som innehåller orden anges som första argument till konstruktorn.

Tips! Följande kod visar hur man kan läsa in ord från en fil.

```
Reader in = Files.newBufferedReader(wordfile);
Scanner scan = new Scanner(in)) {
while (scan.hasNext()) {
    String word = scan.nextLine();
    ...
}
```

Slutför också metoderna `vertexCount` och `vertexSet` så att testen i klassen `TestWordLoading` går igenom.

D2. För att hantera kriteriet för grannskap mellan ord finns det ett gränssnitt som heter `WordCriteria`. Du ska implementera detta gränssnitt genom att slutföra klassen `OneLetterDiff` och implementera metoden `adjacent`. Kodkommentaren beskriver hur metoden ska fungera.

När du är klar med denna deluppgift ska testen i klassen `TestOneLetterDiff` gå igenom.

D3. Skriv kod som skapar bågarna för ordgrafen i konstruktorn till `WordGraph` så att testen i klassen `TestWordEdges` går igenom.

D4. Slutför metoden `distance` i klassen `Lab3`. Denna metod ska beräkna antalet ord i den kortaste ordkedjan som innehåller två stycken startord. Om ingen sådan kedja finns ska -1 returneras.

När du är klar med denna deluppgift ska testen i klassen `TestWordLadders` gå igenom. Testen använder sig av två små ordlistor (`words-1.txt` och `words-2.txt`) och en större med nästan 6000 ord (`words-3.txt`). Det borde inte ta mer än ett par sekunder att köra dessa test – om det tar mycket längre tid så har du troligtvis något fel eller gör något väldigt ineffektivt.

Vid redovisningen ska ni beskriva hur implementeringen av bredden-först-genomgång av en graf skiljer sig respektive liknar implementeringen av djupet-först-genomgång av en graf.

Laboration 4

Mål: Att behärska Dijkstras algoritm för att hitta kortaste väg mellan noder i en graf.

Läsanvisningar

Läs avsnitt 10.6 i läroboken (avsnittet om Dijkstras algoritm). Läs också föreläsningsbilder från föreläsning 6 som behandlar motsvarande avsnitt.

Förberedelser

Läs igenom den inledande texten under rubrikerna ”Datorarbete”. Gör uppgift D1. Läs igenom de övriga uppgifterna och studera de klasser som finns färdigskrivna i laborationskoden.

Datorarbete

Dijkstras algoritm kan användas för att hitta kortaste vägen mellan två noder i en riktad graf. Till skillnad från bredden-först genomgång kan Dijkstra hitta kortaste vägen i en graf med viktade bågar, det vill säga en graf där avstånden på bågar kan vara olika. Dijkstras algoritm fungerar för alla grafer där bågarnas avstånd är positiva – den fungerar ej på grafer där det finns bågar med negativa avstånd. För att implementera Dijkstras algoritm används vanligtvis en prioritetskö där ännu ej besökta noder sorteras i ordning efter deras avstånd från startnoden. Prioritetskön fylls på efter hand och från början innehåller den bara startnoden.

Nedan finns pseudokod för en variant av Dijkstra som hittar kortaste vägen från en nod `u` till en annan nod `v`. Prioritetskön innehåller element med två attribut: en nod i grafen och dess avstånd från ursprungsnoden på den kortaste vägen. I koden används {nod, avstånd} för denna typ av element. Dessa element hålls sorterade i prioritetskön efter ökande avstånd.

```
1  markera alla noder som ej besökta
2  pq = en tom prioritetskö
3  lägg till {u, 0} i pq
4  markera u som besökt
5  spara avståndet till u = 0
6  så länge pq inte är tom:
7      {x, dist} = ta ut minsta elementet ur pq
8      om x == v:
9          return dist
10     annars:
11         för varje båge e från x:
12             w = e.destination
13             newdist = dist + e.distance
14             wdist = det sparade avståndet från u till w
15             om w ej är besökt eller newdist < wdist:
16                 markera w som besökt
17                 spara avståndet till w = newdist
18                 // Notera: w kan redan finnas i pq med ett längre avstånd.
19                 lägg till {w, newdist} w i pq
20 return -1 // Om v ej hittats.
```

Kommentarer till koden ovan:

- Avstånden på bågarna kan ha olika enhet beroende på vilket problem algoritmen appliceras på. Avstånden skulle t.ex. kunna vara heltal eller flyttal som representerar tid eller körsträcka.

- Koden visar inte hur avstånden till olika noder sparas. Till detta kan man använda en tabell, t.ex. en `HashMap`. I denna tabell lägger man in avståndet till en nod första gången den besöks i algoritmen och sedan uppdateras avståndet ifall en kortare väg hittas. Man behöver då inte ha någon separat datastruktur för att lagra vilka noder som besökts - det är samma som de noder som finns som nyckel i avståndstabellen.

I denna laboration ska du implementera två varianter av Dijkstras algoritm: den som visas ovanför och sedan en som inte bara beräknar kortaste avståndet utan också ger kortaste vägen mellan två noder.

För att implementera koden i laborationen behövs grafer där bågarna kan ha avstånd, vilket inte finns i vårat tidigare grafgränssnitt. Därför inför vi en ny klass `DistanceGraph` som har heltalsavstånd på bågarna.

D1. Studera klassen `DistanceGraph` som finns i `src/main/java/lab4`.

För att spara element i prioritetskön i Dijkstra-implementeringen ska du använda en liten hjälpklass som heter `PQElement` och finns i `src/main/java/lab4/Lab4.java`. Bekanta dig med den klassen.

D2. Läs noga igenom algoritmen ovan och kommentarerna om den. Slutför sedan metoden `distance` i klassen `Lab4` i `src/main/java/lab4/Lab4.java`.

När du är klar med denna deluppgift ska testen i klassen `TestDistance` gå igenom.

D3. Du ska nu implementera en variant av Dijkstras algoritm som beräknar kortaste vägen mellan två noder som en lista. Detta ska implementeras i metoden `shortestPath` och när du är klar ska testen i klassen `TestShortestPath` gå igenom.

Om en graf har tre noder: 0, 1, 2, och en båge från nod 0 till ett samt en båge från 1 till 2, då ska den kortaste vägen vara listan $[0, 1, 2]$.

För att skapa listan med den kortaste vägen kan man spara föregående nod när man hittar en ny kortaste väg till en nod `w`. Använd till exempel en `HashMap` för att spara de föregående noderna.

D4. Lägg märke till att en nod kan förekomma flera gånger i prioritetskön i algoritmen som visas ovanför. Detta är inget problem förutom att det kan innebära att loopen på rad 11 körs i onödan. Kan du förklara varför det ibland är onödigt att köra loopen på rad 11 igen för en viss nod?

Går det att ändra din algoritm på något sätt så att den undviker onödiga exekveringar av loopen på rad 11? Implementera en lösning för detta och visa under redovisningen hur det fungerar.

Laboration 5

Mål: Att behärska och kunna implementera en algoritm för beräkning av maximalt flöde i ett nätverk.

Läsanvisningar

Denna algoritm finns inte i läroboken men har behandlats på föreläsning. Studera relevanta bilder från denna föreläsning på kursens hemsida.

Förberedelser

Läs igenom texten under rubriken ”Datorarbete”. Gör uppgift D1. Läs igenom och påbörja uppgift D2.

Datorarbete

Det finns en speciell klass av grafer som kallas flödesnätverk, där varje båge har en icke-negativ kapacitet och ett aktuellt flöde. Sådana grafer kan användas för att representera, t.ex. järnvägsnätverk, nätverk av vatten eller oljeledningar, eller vägnätverk. Ett viktigt problem som dyker upp i flödesnätverk är att beräkna det största möjliga flödet mellan två noder i grafen. Detta problem är känt som “Maximum Flow” problemet och vi ska implementera en algoritm för att lösa det under denna laboration.

Generellt kan vi formulera problemet så här: vi har en riktad graf med icke-negativ kapacitet knuten till varje båge. I grafen finns det två speciella noder: källan som inte har några inkommande bågar och sänkan som saknar utgående bågar. Varje båge måste tilldelas ett icke-negativt flöde som är mindre än eller lika med bågens kapacitet. För alla noder utom källan och sänkan måste det totala flödet in i noden vara lika med det totala flödet ut ur noden. Problemet är att givet dessa kriterier hitta flödesvärden som maximerar det totala flödet in i sänkan (eller, motsvarande, det totala flödet ut ur källan).

På föreläsning har Ford-Fulkersons algoritm för beräkning av maximalt flöde presenterats. Algoritmen ger efter ett ändligt antal steg det optimala flödet förutsatt att de kapaciteter som knyts till bågarna är heltal. I algoritmen brukar man förutsätta att noderna är numrerade från 0 och uppåt och att nätverket representeras av en heltalsmatris. I matrisen anger elementet på rad `i` och kolumn `j` kapaciteten för den båge som förbinder noderna `i` och `j`. Om det saknas bågar mellan vissa par av noder så är motsvarande element i matrisen 0.

D1. Studera klassen `FlowGraph` som finns i `src/main/java/lab5`.

Denna klass representerar ett flödesnätverk och ser lite annorlunda ut från tidagre grafklasser. Den huvudsakliga skillnaden är att grafen representeras med en matris av kapaciteter med attributet `capacity`. Om det finns en båge från en nod u till en annan nod v så är `capacity[u][v]` större än noll.

D2. Huvuduppgiften i denna laboration är att implementera Ford-Fulkerson algoritmen. Mer specifikt ska du implementera Edmonds-Karp varianten där bredden först-sökning används för att hitta en väg i grafen med positivt flöde.

Slutför metoden `maxFlow` i klassen `Lab5`. När du är klar ska testen i klassen `TestMaxFlow` gå igenom. (Testen garanterar inte att din kod är helt korrekt: även om testen går igenom kan det finnas fel i din kod som eventuellt upptäcks senare!)

Som lite hjälp till implementationen har vi några råd till hur du bör strukturera din kod.

- Metoden skall skapa en matris för att representera residualgrafen. Denna matris lagras utanför `FlowGraph`, inuti metoden `maxFlow`. Det är en matris som anger hur mycket flödeskapacitet det finns tillgängligt i grafen med något aktuellt flöde redan genom grafen.
- Initialt har residualmatrisen samma värden som grafens kapacitetsmatris.
- I algoritmen skall man upprepat försöka hitta en väg från källan till sänkan som består av bågar med positiva värden i residualmatrisen. Detta skall fortgå tills man inte längre hittar någon sådan väg. Det är lämpligt att låta en hjälpmetod försöka hitta en sådan väg:
  - Skapa en hjälpmetod som utför bredden först-sökningen i Edmonds-Karp. Metodhuvudet för hjälpmetoden bör se ut ungefär såhär: `boolean bfs(int numVertex, int start, int end, int[][] residual, int[] pred)`
    Metoden ska inte modifiera `residual`, men den modifierar `pred` så att `pred[u]` innehåller föregångaren till u i vägen från `start` till `end`. Detta kommer fungera likt hur kortaste vägen beräknades i laboration 4.
  - Hjälpmetoden returnerar `true` om det finns en väg med kapacitet större än noll i residualmatrisen.
- Efter att hjälpmetoden anropats, om det fanns en väg med flöde större än noll, så måste man hitta "flaskhalsen" på den vägen. Det vill säga, den båge som har minst värde. Det är detta värde som bestämmer hur mycket man kan öka flödet på vägen.
- När flaskhalsen hittats skall residualmatrisen uppdateras för hela den funna vägen från källan till sänkan. Om t.ex. flaskhalsens värde är `c` så innebär det att flödet kan ökas med `c` på alla bågar på vägen från källan till sänkan och därmed minskar de med `c` på motsvarande bågar i residualmatrisen. Dessutom läggs ett omvänt flöde till i residualmatrisen. Endast residualmatrisen behöver uppdateras. `c` returneras till den anropande huvudmetoden som därmed kan hålla reda på flödets storlek.

D3. Nu ska du implementera inläsning av ett flödesnätverk från fil. Filerna har följande format:

- På första raden i filen finns ett heltal som anger antalet noder i grafen (`n`).
- Därefter följer en rad med ett heltal som anger antalet bågar i grafen (`m`)
- De `m` följande raderna har utformningen `u v c` där `u` och `v` är nodnummer (mellan 0 och n-1) och `c` är kapaciteten för bågen från `u` till `v`. Kapaciteterna är positiva heltal eller -1. Om kapaciteten är -1 betyder det att det inte finns någon gräns på hur stort flöde som får skickas genom bågen.

Implementera detta i metoden `loadFlowGraph` i klassen `Lab5`. När filinläsningen fungerar och din algoritm är korrekt ska alla testen i klassen `TestFromFile` gå igenom. Här finns två enkla grafer och en större graf.

D4. *Frivillig uppgift.* Som extra utmaning och övning på att tillämpa Ford-Fulkerson kan ni försöka lösa uppgiften Waif från Kattis öppna problemdatabas:
`https://open.kattis.com/problems/waif`

Hur skulle detta problem kunna modelleras med ett flödesnätverk?

När ni redovisar ska ni kunna förklara Ford-Fulkersons algoritm och hur ni har implementerat den.