UMEÅ UNIVERSITY
Department of Computing Science
Assignment report

March 22, 2020

5DV149 - Assignment 5

Data Structures and Algorithms (C) Spring 2019, 7.5 Credits

Finns en väg?

Name Viktoria Nordkvist, Tobias Bergström

user@cs.umu.se `id19vntm@cs.umu.se & id19tbm@cs.umu.se`

Teacher
Niclas Börlin

Graders
Niclas Börlin, Ola Ringdahl, Fredrik Peteri, Lennart Steinvall

Contents

1 Introduktion

För denna uppgift skulle ett program skrivas som skulle kunna ta in en riktad graf och svara på om det finns en väg mellan två noder som användaren skriver in. Till detta program skulle det skrivas en implementation av en riktad graf som programmet skulle läsa in och programmet skulle även ta in en fil som innehöll grafbeskrivningen. För att svara på frågan om det finns en väg mellan noderna skulle bredden-först-traversering användas. För denna uppgift skulle all kod skrivas i programspråket C.

2 Användarhandledning

För att köra programmet så krävs det att du kompilerar det i din terminal. Vid körning kommer det att skrivas ut instruktioner på skärmen som talar om att du ska skriva in två noder eller så kan du avsluta programmet genom att skriva quit. Om du skriver in två noder kommer programmet undersöka om det går att ta sig mellan dem. Programmet kommer att tala om för dig om någon eller båda noder du skrivit in inte finns och då kommer du få försöka igen. Programmet kommer även att skriva ut ett felmeddelande om du skriver för få noder eller för många. Programmet kommer inte att skriva ut vägen från startnoden till slutnoden utan det kommer endast skriva ut om det är möjligt att ta sig mellan dem eller ej. Om du skriver in samma nod två gånger kommer programmet tolka det som en möjlig väg att ta sig och kommer skriva ut att det finns en väg mellan dem då de är samma.

För att använda detta program behöver det kompileras med de inkluderade filerna som behövs för implementationen och avläsningsprogrammet. Du kompilerar på följande sätt:

gcc -std=c99 -Wall -o isconnected is_connected.c graph.c array_1d.c dlist.c queue.c list.c

För att köra programmet behövs filnamnet på avläsningsprogrammet skrivas tillsammans med namet på filen som innehåller grafbeskrivningen som avläsningsprogammet ska ta in och använda för att svara på om det finns en väg mellan två noder. Programmet kommer att tala om ifall filen med grafbeskrivningen har en felaktig struktur. Du kör programmet på följande sätt:

./isconnected airmap1.map

2.1 Exempel på testkörningar

I detta exempel är noderna flygplatser och huvudprogrammet talar då om för dig om det är möjligt att flyga mellan de två flygplatser, noder, som du skriver in. Här kan du se alla utskrifter som programmet gör för att hjälpa dig.

```
itchy:~/edu/DOA/ou5$ gcc -std=c99 -g -o isconnected is_connected.c graph.c array_1d.c dlist.c queue.c list.c
itchy:~/edu/DOA/ou5$ ./isconnected airmap1.map
Enter origin and destination (quit to exit): UME BMA
There is a path from UME to BMA.

Enter origin and destination (quit to exit): UME GOT
There is a path from UME to GOT.

Enter origin and destination (quit to exit): LLA UME
There is no path from LLA to UME.

Enter origin and destination (quit to exit): PJA PJA
There is a path from PJA to PJA.

Enter origin and destination (quit to exit): dfkdrg UME
Origin does not exist

Enter origin and destination (quit to exit): GOT aaknlnskjlnjsg
Destination does not exist

Enter origin and destination (quit to exit): dja afdadff
None of the nodes exists

Enter origin and destination (quit to exit): quit
Normal exit.
```

Figure 1: Testkörning av huvudprogrammet.

3 Systembeskrivning

Implementationen av grafen är gjord med hjälp av ett fält som en riktad lista. Vi valde denna implementation för att vi tyckte det var enklare att jobba med och förstå när vi skulle planera vår implementation och lägga upp en plan för de olika funktionerna hur de skulle fungera och hänga ihop.

När beslutet om implementationen var taget så konstruerades två strukter, en graf och en nod. I grafen valde vi att ha vårt fält då grafen skulle vara konstruerad som ett fält. Det finns även en variabel size som håller storleken på fältet. Den andra strukten var noden som innehåller en bool som vi använde för att kunna hålla koll på om en nod var sedd eller ej vilket vi behöver vid traverseringen. Den innehåller även namnet på noden samt en lista med varje nods grannar.

3.1 Informell funktionsspecifikation: Graf

Detta är den informella funktionsspecifikationen på datatypen graf:

Empty() - Konstruerar en tom graf som inte innehåller några noder eller bågar. Inga inparamterar och returnerar den tomma grafen.

Insert-Node(v, g) - Lägger in noden v i grafen g. Inparametrar är en nod och grafen som ska ändras. Returnerar den ändrade grafen.

Insert-Edge(e, g) - Lägger till en båge e i grafen g. Inparametrar är en båge och den graf som ska ändras. Returnerar den ändrade grafen.

Isempty(g) - Kollar om grafen g är tom eller ej. Inparametrar är den graf som ska granskas. Returnerar *true* om den är tom eller *false* om den inte är det.

Has-no-edges(g) - Kollar om grafen g saknar bågar. Inparametrar är grafen som ska granskas. Returnerar *true* om den saknar bågar eller *false* om den inte gör det.

Choose-node(g) - Denna funktion returnerar godtycklig nod ur grafen. Inparametrar är grafen som ska granskas. Returnerar en godtycklig nod ur grafen g.

Neighbours(v, g) - Denna returnerar alla grannar till noden v som ligger i grafen g. Inparametrar är noden vars grannar ska inspekteras samt grafen g som noden befinner sig i. Returnerar nodens grannar.

Delete-node(v, g) - Tar bort noden v ifrån grafen g. Detta förutsätter dock att den inte finns en båge mellan en annan nod i grafen. Inparamterar är noden som ska tas bort och grafen som ska modifieras. Returnerar den ändrade grafen.

Delete-edge(e, g) - Tar bort en bågen e mellan två noder i grafen g. Inparametrar är bågen som ska tas bort samt grafen som ska modifieras. Returnderar

den ändrade grafen. [1]

3.2 Informell funktionsspecifikation: Fält

Datatypen fält har använts för att spara ner element av den skapade konstruerade datatypen *node*. Detta representerar alla de olika noderna i listan.

Create(lo, hi) - Skapar en tom array med *lo* som undre gräns i arrayen och *hi* som övre gräns. Inparametrar är två heltal. Returnerar en tom array.

Set-value (i, v, a) - Sätter värdet eller modifierar på plats i i arrayen a till värdet v. Inparametrar är ett heltal som index, ett värde att spara på platsen och arrayen som ska ändras. Returnerar den modifierade arrayen.

Low(a) - Anger arrayens undre gräns. Inparameter är arrayen som ska inspekteras. Returnerar den undre gränsen.

High(a) - Anger arrayens övre gräns. Inparameter är arrayen som ska inspekteras. Returnerar den övre gränsen.

Has-value(i, a) - Kollar om det finns ett värde på platsen i i arrayen a. Inparametrar är den arrayen som ska inspekteras. Returnerar *true* om det finns ett värde på platsen. Annars false.

Inspect-value(i, a) - Inspekterar värdet på platsen i i arrayen a. Inparametrar är index som ska inspekteras samt arrayen. Returnerar värdet på den platsen. [2]

3.3 Informell funktionsspecifikation: Riktad Lista

Datatypen riktad lista har i denna implementation använts för att spara grannarna relaterade till en nod i grafen. Den riktade listan är den del av den skapade datatypen *node* som heter *neighbours*. Detta representerar relationer noden har till alla andra noder i grafen.

Empty() - Skapar en tom lista som inte innehåller några element. Returnerar den tomma listan.

Insert(v, p, l) - Sätter in värdet v direkt innan positionen p i listan l. Inparametrar är ett värdet, en position och en lista att modifiera. Returnerar positionen för det nyinsatta värdet.

Isempty(l) - Kollar om listan är tom eller ej. Inparametrar är listan som ska inspekteras. Returnerar *true* om och endast om listan är helt tom.

[1] Janlert, Lars-Erik och Wiberg, Torbjörn. Datatyper och algoritmer. Sida 339. Studentlitteratur, Andra upplagan.

[2] Janlert, Lars-Erik och Wiberg, Torbjörn. Datatyper och algoritmer. Sida 92. Studentlitteratur, Andra upplagan.

Inspect(p, l) - Hämtar värdet som är lagrat på positionen p i listan l. Inparametrar är positionen och listan att inspektera. Returnerar värdet på platsen.

Isend(p, l) - Kollar om positionen p är den sista i listan l. Inparametrar är en position och lista som ska inspekteras. Returnerar *true* om p är sista platsen i listan.

First(l) - Hämtar första positionen i listan l. Inparametrar är listan som ska inspekteras. Returnerar första platsen i listan.

Next(p, l) - Sätter positionen p till nästkommande plats i listan l. Inparametrar är positionen som ska modifieras samt listan. Returnerar den modifierade positionen.

Remove(p, l) - Tar bort värdet på positonen p i listan l. Inparametrar är positionen där värdet som ska tas bort ligger samt listan. Returnerar positionen som kommer efter den borttagna positionen. [3]

3.4 Informell funktionsspecifikation: Kö

Datatypen kö har använts vid bredden först sökningen som görs vid i find_path funktionen. Se kapitel 3.6 för mer infromation om detta.

Empty() - Skapar en tom kö som inte innehåller några element. Returnerar en tom kö.

Isempty(q) - Inspekterar om kön är tom. Inparametrar är kön som ska inspekteras. Returnerar *true* om kön är tom. Annars false.

Front(q) - Inspekterar det fösta värdet i kön q. Inparametrar är kön som ska inspekteras. Returnerar det första värdet i kön.

Enqueue(v, q) - Köar värdet v sist i kön q. Inparametrar är värdet som ska köas och kön. Returnerar den modifierade kön.

Dequeue(v, q) - Tar bort första värdet i kön q. Inparametrar är ett värde och kön som ska modifieras. Returnerar den modifierade kön. [4]

[3] Janlert, Lars-Erik och Wiberg, Torbjörn. Datatyper och algoritmer. Sida 66. Studentlitteratur, Andra upplagan.

[4] Janlert, Lars-Erik och Wiberg, Torbjörn. Datatyper och algoritmer. Sida 159. Studentlitteratur, Andra upplagan.

3.5 Implementerad funktionsspecifikation: Graf

En stor av del av grafens informella funktionsspecifikation behölls och i denna sektion beskrivs det som skilljer sig mellan den informella specifikationen och implementationen gjord för denna uppgift.

Nodes-are-equal(n1, n2) - Detta är en extra funktion som är del av implementationen. Denna tar in två noder och kollar om de är samma. Inparametrar är de två noder som ska jämföras. Returnerar *true* om de är samma och *false* om de inte är samma.

Empty(m) - I implementationen så skiljer sig Empty() med att vi tar in antalet maxnoder som inparameter. Detta då vi använt oss av en endimensionell array för att konstruera grafen, då array inte är en dynamisk datatyp måste storleken sättas vid allokering. Annars fungerar den i stort sätt samma.

Insert-node(g, str) - Denna implementationen tar in en string (namn) istället för en nod. Detta då vi skapar en nod i insert node och då vi ska läsa in filer med nodnamnen så är detta den infon vi kan få utifrån om noden.

Find-node(g, str) - Denna bygger på samma princip som graph-insert-node, istället för att vi tar in en nod så tar vi in en string (namn). Detta då strings är ett effektivt sätt att jämföra noder med tanke på hur insert är implementerad.

Node-is-seen(g, v) - Kollar om noden v har status seen = *true* eller *false*. Inparametrar är grafen och noden som ska granskas. Returnerar *true* om den är "seen" och *false* om den inte är det.

Node-set-seen(g, v, b) - Sätter noden v till seen = boolen b. Inparametrar är grafen och noden som ska modifieras samt ett booleskt värde att sätta noden till. Returnerar den ändrade grafen.

Reset-seen(g) - Sätter alla noders seen status till *false* i grafen g. Inparametrar är den grafen man ska gå igenom. Returnerar den ändrade grafen.

Choose-node(g) - Godtycklig nod i denna implementationen är första noden som ligger i grafen.

Kill(g) - Returnerar allt allokerat minne i grafen g. Inparametrar är grafen som ska avallokeras. Returnerar ingenting.

Delete-node(e, g) - Denna är inte med i implementationen då den inte används och då inte fyller någon funktion.

Delete-edge(e, g) - Denna är inte med i implementationen då den inte används och då inte fyller någon funktion.

Print(g) - Denna är inte med i implementationen då den inte används och då inte fyller någon funktion.

3.6 Bredden-först-traversering

Vid vår bredden-först-traversering använde vi oss av en kö för att traversera grafen. Kön skapar vi i vår funktion find_path som undersöker om det finns en väg mellan de två noder som användaren anger. Först lägger vi till startnoden och sedan dess grannar. Sedan sätter vi den till sedd och går till dess första granne, lägger till dess grannar som ej är sedda sist i kön. Vi valde att göra våran grannlista med endast namn vilket betyder att vi måste hämta motsvarande nod från fältet. Detta underlättar för oss när vi ska lägga till grannar i kön, för vi vill endast lägga till grannar som inte redan är sedda. Eftersom vi hämtar noderna från fältet måste vi inte bry oss om att sätta alla till exempel MMX till sedda då vi hämtar samma MMX nod varje gång och då bara behöver sätta MMX till sedd en gång. Efter att vi lagt till grannarna som ej är sedda i kön och satt noden vi traverserar till sedd tar vi bort den från kön. Detta upprepas tills kön är tom eller att slutnoden har hittats. Varje gång första namnet i kön läses av så jämförs det med slutnodens namn och på så sätt gör vi vår bredden-först-traversering för att ta reda på om det går att ta sig mellan de två noderna som användaren anger.

4 Algoritmbeskrivning

Vårt huvudprogram is_connected består av sju funktioner och en main funktion. Av de sju funktionerna var fyra av dem givna och de är *first_non_white_space, last_non_white_space, line_is_blank, line_is_comment,* . De andra tre har vi skrivit och de är *line_is_digit*, *find_path* och *read_file* .

Funktionen read_file fungerar på sådant sätt att den tar in en pekare till en graf och namnet på filen med grafbeskrivningen. Denna funktion använder sig av de fyra givna funktionerna för att urskilja vilka rader den ska läsa av och inte. Den använder sig även av den egenskrivna line_is_digit för att kunna läsa av siffran som står i filen med grafbeskrivningen. Siffan som står i grafbeskrivningen är antalet bågar som finns i grafen och den siffran behöver vi komma åt för att skapa grafen. Vi använder den för att veta hur stor vi ska göra vår graf för om vi kan komma åt antalet bågar i grafen så vet vi då att maxantalet noder är då antalet bågar multiplicerat med två, vilket är den storlek vi har valt att göra vår graf. Funktionen read_file fungerar på följande sätt:

- Öppnar filen
- Kollar om filen är tom, om så skrivs ett felmeddelande ut
- Loopar och läser en rad i taget tills den kommit till slutet av filen
  - Undersöker om raden är en blankrad eller kommentar då går vi vidare till nästa rad
  - Kollar första icke-blankrad om det finns en siffra, om så tar vi den och använder den för att skapa vår graf och sätta storleken på den. Finns det ingen skrivs ett felmeddelande ut för då är filen felaktig
  - Om det bara står en nod på raden då skrivs ett felmeddelande ut för då är filen felaktig
  - Om den första avlästa noden inte redan finns i fältet ska den läggas till i fältet
  - Om den andra avlästa noden inte redan finns i fältet ska den läggast till i fältet
  - Hämtar den första inlästa nodens lista med grannar
  - Går igenom listan med grannar
    - Om första noden redan har en båge till den andra noden finns det dubletter i den inlästa filen och då skrivs ett felmeddelande ut för då är filen felaktig
    - Annars läggs den andra noden till i den första nodens grannlista
- Stänger filen
- Returnerar grafen

Funktionen find_path är vår funktion som returnerar om det går att ta sig mellan de två noderna som användaren anger. Den tar in en pekare till grafen och de två noderna som användaren angett. Funktionen fungerar på följande sätt:

- Hämtar startnodens lista med grannar
- Kollar om startnoden och slutnoden är samma, om de är det så returneras sant
- Om de inte är samma skapas en kö för att påbörja en bredden-först-traversering
  - Lägg till startnoden i kön
  - Går igenom dess lista med grannar och lägger till dem i kön
  - Sätter startnoden till sedd och tar bort den från kön
  - Går sedan igenom kön
    - Hämtar första noden i kön och dess lista med grannar
    - Kollar om denna nod är slutnoden, om så returnerar vi sant och dödar grafen
    - Medan vi inte är i slutet av den nuvarande nodens lista
      - Hämtar noden med motsvarande namn som de som finns i grannlistan
      - Om någon granne till noden redan är sedd läggs den ej till i kön, är den ej sedd läggs den till
      - Den nuvarande noden sätts sedan till sedd och tas bort från kön
  - Sätter alla noder i grafen till osedda och dödar grafen

Funktionen line_is_digit är vår funktion som vi använder för att hämta antalet bågar som står i filen med grafbeskrivningen. Den fungerar på sådant sätt att den tar in en char och med hjälp av en funktion atoi som försöker göra om den intagna char variabeln till en int, vilket endast kommer fungera om det är en siffra. Om den lyckas så returnerar atoi int värdet och om den inte lyckas, till exempel om den char som togs in av funktionen var ett namn så returnerar atoi noll. Funktionen returnerar falskt om atoi returnerar noll och funktionen returnerar sant om atoi returnerar ett värde som inte är noll.

5 Testkörningar

Vi har gjort olika testkörningar för att testa så att våran graf implementation fungerar som den ska. Detta har gjorts med åtta olika filer som skickas in till programmet som läses av. Dessa tester testar att våran implementation klarar av det formatet som är specificerat i specifikationen. Specifikationen finns som

bilaga 1 under kaptiel 8 Bilagor. Nedan redovisas de olika testen som har använts samt vad de testar.

1-airmap1.map - Detta testet används för att testa en standard graf som inte är sammanhängande. De inputs som är testade är:

- UME GOT
- UME PJA
- UME GOT
- quit

Inputs på senare test ser liknande ut men för att inte fylla dokumentet med tester kommer dessa inte skrivas ut i rapporten vissa inputs är långa och tar upp en del plats. Alla dessa inputs gav de resultaten de skulle göra

2-repeated-questions.map - Detta testet använder samma graf som den tidigare men testar alla olika kombinationer av noder två gånger. Alla dessa inputs gav de resultaten de skulle göra.

3-directed-graph.map - Detta testet använder en annan graf än de två tidigare och testar så att implementationen hanterar riktade grafen så den ska göra. Alla dessa inputs gav de resultaten de skulle göra.

4-standard-test.map - Detta testet är väldigt likt det första, dock innehåller detta inga kommentarer eller blankrader. Detta är alltså till för att se så koden fungerar även utan kommentarer eller blankrader. Alla dessa inputs gav de resultaten de skulle göra.

5-big-map.map - Detta testet är kollar så att implementationen klarar av stora grafen med stor mängd information. Tester kollar så att implementationen klarar av att hitta en path även om den behöver traversera långt i grafen. Alla dessa inputs gav de resultaten de skulle göra.

6-single-edge.map - Detta testet använder sig endast av en båge och kollar så att den fungerar som den ska. Alla dessa inputs gav de resultaten de skulle göra.

7-single-node.map - Detta testet använder sig endast av en node och kollar så att den fungerar som den ska. Alla dessa inputs gav de resultaten de skulle göra.

x-bad-map-format.map - Detta testet använder ett felaktigt format och används för att kolla så att våran filavläsning nekar denna typen av format. Alla dessa inputs gav de resultaten de skulle göra.

6 Arbetsfördeling

När det kommer till koden har vi i stort sätt skrivit allting tillsammans. Vi har hjälpt varandra koda och disskuterat de olika algoritmerna och hur bäst att implementera dem, etc. Både implementationen av grafen och huvudprogrammet är skrivit tillsammans. Detta har hjälpt oss båda då varken av oss är erfarna programmerare och man kan snabbt och enkelt hitta varandras misstag och tankevurpor. Då vi komminucerat mycket såg vi också till att vi båda hade samma plan i åtanke under hela processen och att vi var i sync när det kom till vad som ska göras och vad som är gjort.

Alla de olika testerna har också gjorts tillsammans, både test för minnesläckor (Valgrind) och test för implementationen (se Kapitel 5 Testkörningar.). Då koden är skriven tillsammans så valde vi att såklart köra dessa tillsammans.

När det kommer till rapporten så har vi valt att ta var sin rubrik att skriva för att effektivisera skrivandet av rapporten. Då båda tagit del av alla delar av programmeringen så underlättade det vid skrivandet av denna rapport då vi båda har full koll på alla delar som ska täckas i rapporten.

Arbetsfördelningen vi har valt offrar lite effektivitet för bättre kommunikation och information, genom hela processen såg vi till att båda visste exakt vad som ska göras, vad som har gjorts och hur allting hänger ihop. Som sagt kanske det inte var det mest effektiva men vi har helt eliminerat risken att någon kodar någonting som bygger på ett missföstående mellan gruppmedlemmarna eller liknande som leder till att man måste skriva om kod. Detta tycker båda har varit bekvämt och fungerar bra.

7 Reflektioner

Vi tycker att denna uppgiften har varit väldigt svår men lärorik. Detta har varit ett stort steg i jämförelse med tidigare programmeringsuppgifter då det krävt både mer kod och mer förstående, vi har dock på så sätt också lärt oss mycket mer. Vi stötte aldrig på riktigt stora problem även om det var svårt, de mesta gick att lösa efter lite felsökning.

Det som var svårast med uppgiften var att komma igång med implementationen, vi spenderade mycket tid på att disskutera hur grafen skulle implementeras och hur vi ville att den skulle fungera. Detta gjorde att mycket tid gick till att bygga upp en mental modell tillsammans för att få en förståelse för hur systemet ska se ut. Efter den delen var klar så rullade hela uppgiften på ganska väl och vi stötte som sagt inte på några större problem, vi stötte mest på småproblem men ofta.

Det roligaste med uppgiften tyckte vi var filhanteringen, även om det var lite krångligt så kändes det användbart och var kul att göra något som var väldigt nytt då ingen av oss hållt på med filhantering i denna formen innan. Det var

också fantastiskt skönt att få det att fungera tillslut.

I sin helhet har uppgiften varit väldigt lärorik då den tar upp en del nya delar som vi inte hanterat innan när det kommer till programmering. Även om detta gjorde det lite mer krångligt så är det just det som gjort det mer lärorikt.

8 Bilagor

- Filen kan innehålla blankrader och kommentarsrader som ska ignoreras.
  - En kommentarsrad inleds med tecknet #.
- Första raden (förutom blank- och kommentarsrader) innehåller ett heltal med antalet kanter i grafen.
- Övriga rader innehåller en båge var:
  - Raderna innehåller två nodnamn var separerade av mellanslag.
  - Raderna kan inledas med blanktecken som ska ignoreras.
  - Raderna kan avslutas med mellanslag och extra text som också ska ignoreras.
  - Nodnamnen består av alfanumeriska tecken och är vardera max 40 tecken långa.
    - Alfanumeriska tecken inkluderar tecknen 0-9, a-z, A-Z.
  - Avslutande rad i filen kan sakna radslut.
- Textfilen kan vara i unix eller dos-format.
- Det finns inga dubblettbågar i filen.

Figure 2: Krav på filhanteringen