Datastrukturer och algoritmer 5DV149

Obligatorisk uppgift 5



*Bildrättigheterna tillhöra UWA EDU och är under CC licens.*

Max Malmer(c17mmr)

Kontakt: [c17mmr@cs.umu.se](mailto:c17mmr@cs.umu.se) 2019-04-03, v.2.0

Innehållsförteckning

[1. Komplettering 3](#_Toc5192495)

[2. Introduktion 4](#_Toc5192496)

[2.1 Bakgrund 4](#_Toc5192497)

[2.2 Problembeskrivning 4](#_Toc5192498)

[3. Användarhandledning 5](#_Toc5192499)

[3.1 Programmet 5](#_Toc5192500)

[3.2 Indata och kartfilen 6](#_Toc5192501)

[4. Systembeskrivning 7](#_Toc5192502)

[4.1 Grafen 7](#_Toc5192503)

[4.2 Programmet 7](#_Toc5192504)

[4.3 Övriga datatyper 7](#_Toc5192505)

[5. Algoritmbeskrivningar 7](#_Toc5192506)

[5.1 Grafen 7](#_Toc5192507)

[5.2 Programmet 9](#_Toc5192508)

[6. Testkörningar 10](#_Toc5192509)

[7. Reflektioner 13](#_Toc5192510)

[8. Referenser 14](#_Toc5192511)

# Komplettering

Efter att ha fått tillbaka uppgiften med betyget O så gick genom alla kommentarer och insåg att jag behövde tänka om lite. I rapporten la jag in numrering på alla rubriken och ett avsnitt om felhantering i testnings avsnittet av rapporten. I koden så la jag till mycket mer felhantering för felaktiga kartfiler. Även så insåg jag att gått lite överstyr och gjort sådant att min graf hanterar allt som en allmän graf, inte en riktad. Så det enda jag behövde göra för att ändra det var att ta bort lite kod. Nu hanteras varje båge för sig i grafens struktur. Jag gjorde också om grafens inläsning lite för att kunna hantera grafer i gränsfall med antalet noder och bågar.

I övrigt så gick jag också genom och såg till att mitt program kunde hantera om en nod inte hade någon granne. Vet att detta inte ingick i specifikationen men jag tänkte att det var ett bra tillägg. Jag kontrollerade också sådant att grafen hanterar om man sätter in två likadana noder i programmet som att dom är kopplade som specifikationen säger. Något jag slutligen gjorde också var att jag kommenterade koden i is\_connected.c sådant att den blev lite lättare läsbar.

# Introduktion

## Bakgrund

”Programming is a skill best acquired by practice and example rather then from books’’ – Alan Turing

Detta är en rapport som beskriver ett program och en implementation av en graf för kursen Datastrukturer och algoritmer 5DV149. Syftet med uppgiften är att använda existerande datatypsimplementationer till att implementera några grafalgoritmer, lära sig att bryta ner ett komplext problem i mindre problem ,kunna använda filhantering av textfiler och dokumentera ett program. Allt detta implementeras i programmeringsspråket C men tillämpad dynamisk minnesallokering.

## Problembeskrivning

I denna uppgift beskrivs ett program som läser in en textfil som i sin tur beskriver en riktad graf. Programmet kan sedan undersöka i fall att det finns någon stig mellan noderna i grafen genom att traversera bågarna med hjälp utav en bredden-först-sökning. Till exempel så kan filen som läses in specificera ett antal bågar mellan några flygrutter. Eller så kan den beskriva om det går att handla vissa produkter från vissa särskilda land i ett köpes system. Datatypen graf har en enormt stor grad av olika implementationer och är viktig för oss som digitalt samhälle. Särskilt idag när machine learning blir ett mer och mer aktuellt ämne så blir grafen bara viktigare och viktigare eftersom det är den huvudsakliga implementationen. Denna graf kan inte åstadkomma något sådant dessvärre, men den kan ha andra praktiska användningar.

# Användarhandledning

## Programmet

Programmet är indelat i två filer, **graph.c**(där implementationen av själva grafen ligger) och **is\_connected.c**(där programmet som läser in textfilen och traverserar grafen ligger). För att kompilera och köra programmet så behövs också några filer från kodbasen på den ovannämnda kursen. Dessa är: **graph.h list.h list.c queue.h queue.c dlist.h dlist.c array\_1d.h array\_1d.c array\_2d.h array\_2d.c**. Det krävs också att kodbasen är minst av version 1.0.7 eller nyare. Tecknet $ används för början av en ny rad i valfri terminal.

Programmet kompileras med följande kommando:

**$gcc -Wall -std=c99 -g -o is\_connected is\_connected.c graph.c queue.c list.c dlist.c array\_1d.c array\_2d.c**

För att köra programmet använd följande kommando:

**$./is\_connected dinkartfil.map**

När programmet startat så kommer du se följande prompt:

**Enter origin and destination (quit to exit):**

För att avsluta programmet så skriv som instruerat quit, annars så kan ett exempel på en indata du kan skriva vara:

**Enter origin and destination (quit to exit): NODE1 NODE2**

Programmet söker sedan genom grafen efter dessa två noder och kontrollerar sedan om det finns en stig mellan dom. Om det finns en stig mellan noderna eller om noderna är samma nod så kommer följande utskrift dyka upp:

**There is a path from NODE1 to NODE2.**

Om det inte finns en stig mellan noderna så skrivs följande ut:

**There is no path from NODE1 to NODE2.**

## Indata och kartfilen

Kartfilen som skall läsas in ska vara i formatet .map. Själva filens utformning ska vara som följande:

#Förslagsvis en kommentar om filen och dess innehåll  
4(antalet bågar i grafen)  
NODE1 NODE2 #kommentar  
NODE2 NODE3 #kommentar   
NODE4 NODE5 #kommentar

Filen får innehålla blankrader som kommer att ignoreras utav programmet. Kommentarsrader börjar med symbolen # och är helt valfria. Nodnamnen ska bestå av max 40 stycken alfanumeriska tecken och filen kan vara i antingen Dos eller Unix format. Det ska inte heller förekomma dubblettbågar i filen. Ett exempel på en kartfil kan vara:

# SAS ar bast  
8  
UME BMA # Umea-Bromma  
BMA UME # Bromma-Umea  
BMA MMX # Bromma-Malmo  
MMX BMA # Malmo-Bromma  
BMA GOT # Bromma-Goteborg  
GOT BMA # Goteborg-Bromma  
LLA PJA # Lulea-Pajala  
PJA LLA # Pajala-Lulea

Om det skulle vara så att din fil saknar noder eller är i fel format kommer detta skrivas ut:

**dinfil.map has no nodes or is in the wrong format.**

Om det skulle vara så att din fils bågar är felformaterade så kommer följande felmeddelande att skrivas ut:

**dinfil.map has no edges or is in the wrong format.**

Om efter programmet avslutas så skrivs följande meddelande ut: **Normal exit.**

Om du använder noder som inte finns eller felformaterade strängar så kommer följande utskrift:

**Please use the format: node\_name node\_name**

# Systembeskrivning

## Grafen

Grafen som jag har implementerat den är en relativt vanlig implementation som kallas en ’’adjecency matrix’’. Det innebär alltså att grafen är representerad av en matris med sanningsvärden. När grafen initieras så är samtliga av värdena falska men när noder läggs in i grafen så blir koppling mellan nod till nod sann. När sedan en båge läggs in i grafen så läggs sanningsvälden med inställningen sant in på korresponderande platser i matrisen. Nodernas namn och index representeras i en separat array. Noderna ligger på samma korresponderade palts i matrisen som i sin egen array. Det finns också några övriga data som representerar hur många bågar, noder och hur många noder det som mest är i grafen. Jag valde att implementera alla funktioner som specificeras i grap.h. Hela grafen finns representerad i graph.c.

## Programmet

Programmet jag har byggt använder den i introduktionen specificerade körningen. Programmet har ansvaret för att av allokera minne och har också algoritmen som söker genom grafen i abstraktionen utav en riktad graf.

## Övriga datatyper

Det finns fler datatyper som inte jag har implementerat som detta program använder sig av. Dessa datatyper är: lista, riktad lista, kö, tvådimensionell array och endimensionell array. Dom övriga datatyperna återfinns i filerna: **graph.h list.h list.c queue.h queue.c dlist.h dlist.c array\_1d.h array\_1d.c array\_2d.h array\_2d.c**. Dessa datatyper är implementerade av kursansvariga och handledare från institutionen för datavetenskap på Umeå Universitet.

# Algoritmbeskrivningar

Nedan beskrivs alla algoritmer i implementationen som inte är triviala.

## Grafen

**graph\_empty():** Är algoritmen som skapar en tom graf. Den allokerar minne för alla variabler som internt används som t.ex. matrisen och arrayen som innehåller alla sanningsvärden. Alla sanningsvärden är allokerade som datatypen bool för läsbarhet. Indata är de maximala antalet noder som grafen kan ha och utdata är en tom allokerad graf.

**nodes\_are\_equal():**Är en algoritm som jämför två noder i grafen och kontrollerar om det är samma nod. Indata är två noder att jämföra, en graf och utdata är ett sanningsvärde, sant om dom är lika och falskt om dom inte är det.

**graph\_is\_empty():**Kontrollerar om grafen är tom eller inte. Indatat är en graf att kontrollera och utdatat är sant om grafen är tom, falskt om den inte är det.

**graph\_has\_edges():** Kontrollerar om grafen har några bågar eller inte. Indatat är en graf att kontrollera och utdatat är sant om grafen har bågar, falskt om den inte har det.

**graph\_insert\_node():** Sätter in en nod i grafen. Indatat är grafen samt namnet på noden att sätta in och utdatat är en modifierad graf.

**graph\_find\_node():** Söker genom grafen efter en nod som specificeras per namn i indatat. Den gör detta genom att kontrollera alla namn i arrayen med noder för att sen om någon har det namet. Indatat är namet på noden och grafen. Utdatat är noden som du sökte efter eller NULL om den inte finns i grafen.

**graph\_node\_is\_seen():** Är en algoritm som söker genom grafen efter en nod för att kontrollera om den har blivit sedd eller inte. Detta gör den genom att göra jämförelser på varje nod i arrayen med noder. Indatat är grafen och noden som ska kontrolleras. Utdatat är ett sanningsvärde, det är falskt om den inte är tidigare sedd och sant om den har varit tidigare sedd.

**graph\_node\_set\_seen():** Går genom arrayen med noder och sätter en nod till sedd eller inte sedd beroende på sanningsvärdet i indatat. Indata är grafen, noden och sanningsvärdet som ska sättas. Utdatat är den modifierade grafen.

**graph\_reset\_seen():** Är en algoritm som itererar hela arrayen med noder och sätter alla sanningsvärden som representerar om noden är sedd eller ej till falskt. Indatat är grafen och utdatat är den modifierade grafen.

**graph\_insert\_edge():** Itererar både arrayen med noder och arrayen med sanningvärden. När den hittar den korresponderande nodparet så sätter den dess kopplingar till sann i matrisen med sanningsvärden. Indatat är grafen och de två noderna som bågen är mellan. Utdatat är den modifierade grafen. Algoritmen är odefinierad om noderna inte finns i grafen.

**graph\_delete\_node():** Tar in en graf och en nod och tar bort noden som specificerats i indata. Indata för algoritmen är grafen och noden. Utdatat är den modifierade grafen. Algoritmen är odefinierad om noderna inte finns i grafen. Algoritmen är odefinierad om noden inte finns i grafen.

**graph\_delete\_edge():** Tar in en graf och två noder som en båge går mellan och tar bort bågen ut grafen. Indata är grafen och de två noderna bågen går mellan och utdata är den modifierade grafen. Algoritmen är odefinierad om noderna inte finns i grafen.

**graph\_choose\_node():** Tar in en graf och ger tillbaka en slumpmässig nod från grafen. Indata är en graf och utdata är en slumpmässig nod.

**graph\_neighbours():** Är en algoritm som tar och kontrollerar en nod för att sedan söka genom sanningsmatrisen för att utröna vilka noder som den noden har bågar till. Indatat är grafen och noden att kontrollera. Utdatat är en riktad lista med namnen på alla grannar i form av strängar. Listan måste av allokeras efter användning av denna algoritm.

**graph\_kill():** Är den funktion och algoritm som av allokerar allt minne som använts utav grafen. Den gör detta genom användning av **free()** funktionen. Indatat är grafen som skall av allokeras och utdatat är inget.

**graph\_print();** Kan användas till att skriva ut grafen. Indatat är grafen du vill skriva ut och utdata är inget. Men du får en fin utskrift i terminalen.

## Programmet

**find\_path();**Är en grafsökningsalgoritm som använder på-bredden-sökning för att traversera grafen som en riktad graf för att hitta eller inte till noden som den skall söka till. Den använder datatypen kö för att hålla koll på vart i grafen den är och datatypen riktad lista för att veta vilka grannar som en nod har.

**main();** Är den funktion som sköter allokering av grafen i programmet och inläsningen av filen. Den allokerar grafens storlek genom att läsa hur många rader som filen har för att inte riskera minnesfel. Denna algoritm sköter även alla utskrifter till terminalen. Indatat är filnamnet på filen som ska läsas in och udatat är heltalet 0 när körningen avslutas.

# Testkörningar

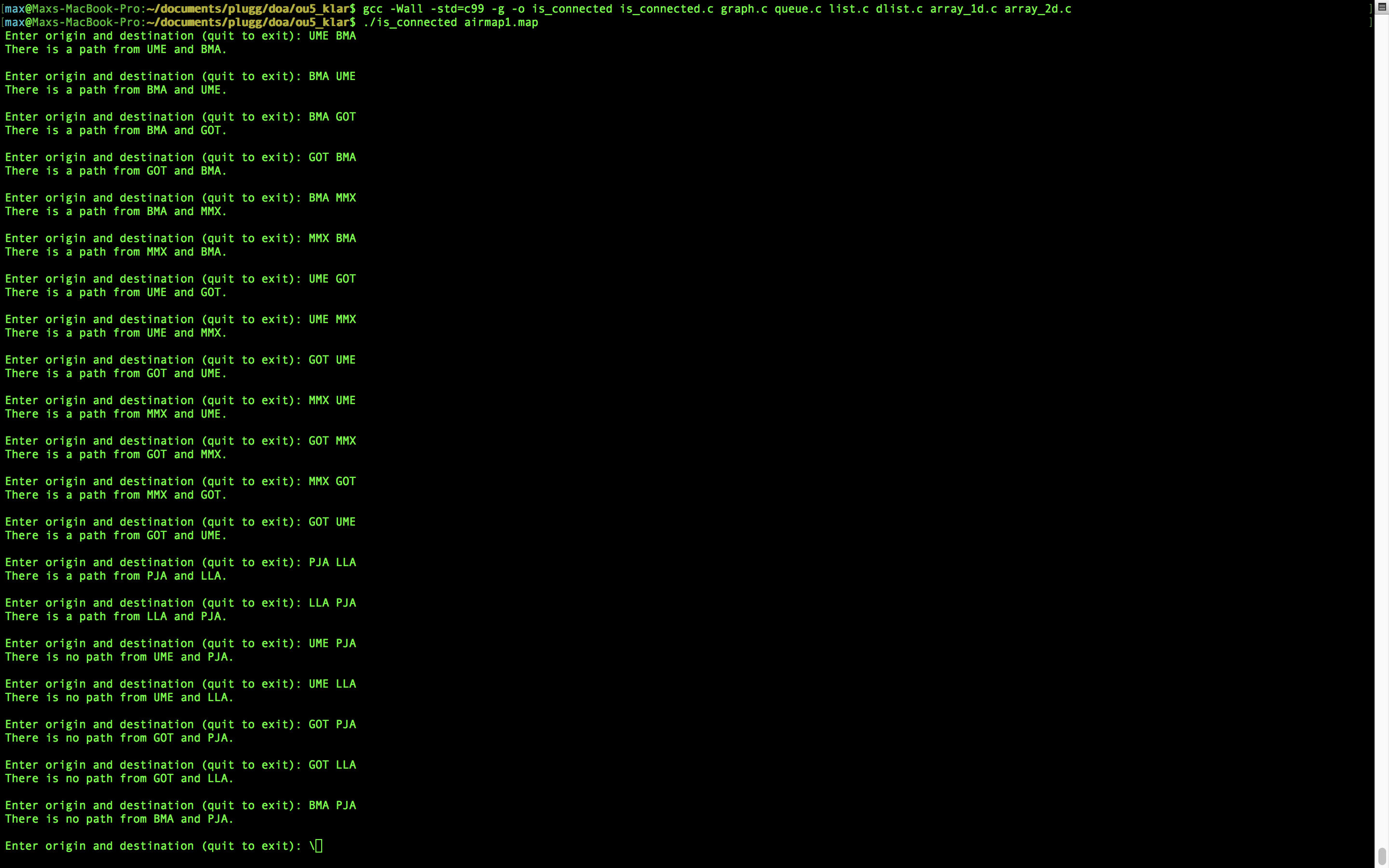
Jag har testat programmets och grafens funktionalitet med hjälp av 3 kartfiler. Dom ser ut på följande sätt:

# Some airline network  
8  
UME BMA # Umea-Bromma  
BMA UME # Bromma-Umea  
BMA MMX # Bromma-Malmo  
MMX BMA # Malmo-Bromma  
BMA GOT # Bromma-Goteborg  
GOT BMA # Goteborg-Bromma  
LLA PJA # Lulea-Pajala  
PJA LLA # Pajala-Lulea

1  
A B

2  
A B  
C D

Min ursprungliga plan var att utveckla ett .sh testskript för att testa alla bågar och stigar mellan alla specificerade noder. Tyvärr så blev det inte så att jag hade tid till att göra detta i slutändan. Men jag höll mig till min plan och testade min implementation med samma metod på alla tre kartfiler. Dessutom så testade jag att plocka sönder kartfilerna för att se om mina felkontroller fungerade som dom ska. För att kontrollera att min implementation inte hade några minnesläckor eller minnesfel så använde jag valgrind. De system jag provat att köra min implementation på är: MacOS 10 High Sierra, Ubuntu 18.4 och Windows 10. På nästkommande sida rapportern återfinns en exempelbild på en testkörning.

****

Jag testade också programmet för felaktiga inmatningar. Detta gjorde jag genom att skapa flera felaktiga kartfiler för att sedan starta programmet med dom. Exempel på sådana felaktiga kartfiler är t.ex.

0   
A B  
B A  
C D

1  
A B   
B A

2  
A BBBBBBB B

2

A BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

# Reflektioner

Överlag så har det här varit en mycket bra och intressant uppgift. Jag tycker verkligen om formatet att jobba mot en förutbestämd gränsyta med förutbestämda mål då detta mer än ofta överensstämmer med hur man arbetar med kod i verkligheten.

Dock så hade det varit en snabbare och mer minneseffektiv implementation om jag valt att använda en ’’edge list’’ eller en ’’adjacency list’’ istället. Problemet med en ’’adjacency martix’’ är att den allokerar mycket minne för grafen och håller det. Samt så tar alla grafsökningar som görs O(n^2) i i tid, minst. Den typen av implementation jag gjort här har egentligen bäst applikation om man ska använda heltal för att representera noderna. Martisen har också stora fördelar över listan om man vet i förväg på vilken index som noderna man söker ligger på.

Med det sagt så har jag tidigare implementerat graf som lista och jag var nyfiken på hur det skulle fungera att implementera den som martis istället och det visade sig vara en intressant utmaning. Dock så hade jag absolut kunnat göra en bättre implementation än den jag gjorde. Jag experimenterade och använda fscanf istället för sscanf vilket visade sig vara väldigt krångligt och tog många timmar av läsning för att förstå hur den korrekt skulle användas. Min implementation tar också och använder lite mer minne än vad den faktiskt behöver.

Det ska dock sägas att jag gjorde denna implementation helt själv då jag just nu gör ett år på kåren och inte hade någon att jobba med. Och utifrån dom förhållandena så är jag mycket nöjd med min arbetsinsats.

# Referenser

1. <https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs> 2019-03-25
2. <https://www.youtube.com/watch?v=zaBhtODEL0w> 2019-03-25
3. <https://www.tutorialspoint.com/c_standard_library/index.htm> 2019-03-25
4. [http://www8.cs.umu.se/kurser/5DV149/VT19-1/labbar/ou5/spec.html 2019-03-25](http://www8.cs.umu.se/kurser/5DV149/VT19-1/labbar/ou5/spec.html%202019-03-25)
5. <https://www.turing.org.uk/> 2019-03-25
6. Janlert, Lars-Erik., Wiberg, Torbjörn. 2000. *Datatyper och Algoritmer*. (2:a upplagan). Lund: Studentlitteratur AB.