UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport obligatorisk uppgift

Datastrukturer och algoritmer (C) 5DV149 7.5 hp

Obligatorisk uppgift nr

5

Namn	Matilda Nilsson, Amine Balta
E-post	id14mnn@cs.umu.se, id14aba@cs.umu.se
CS	id14mnn, id14aba
Datum	2019-07-11
Kursansvarig	Niclas Börlin (niclas.Borlin@cs.umu.se)
Version	2.0

Innehållsförteckning

1. Introduktion

Denna uppgifts huvudsakliga syfte är att med hjälp av olika problemlösningsstrategier och programmeringsspråket C komma fram till en lösning för att hitta en koppling mellan två flygplatser. Konkret ska programmet kunna läsa in en beskrivning av en riktad graf och därefter ska programmet svara på frågan om två noder i grafen är länkade till varandra, om två flygförbindelser finns. För att få fram om två noder är länkade ska programmet göra en bredden-först-traversering av grafen. Det som först måste ske är att data från en textfil med informationen om flygförbindelserna läses in, sedan läses det över i en graf, därefter ska de två flygplatsinmatningarna läsas in, där gör bredden-först-traviseringen för att då se om de är kopplade i grafen.

2. Användarhandledning

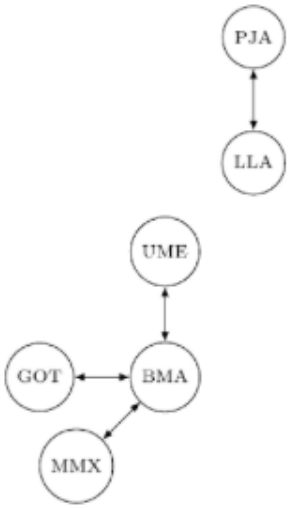
För att programmet ska kunna exekveras bör en en mapp med samtliga filer finna tillgängliga:

- `dlist.h, dlist.c`
- `grapth.c, graph.h`
- `queue.h, queue.c`
- `array_2d.c, array_2d.h`
- `list.c, list.h`
- `util.h`
- `is_connected.c,` vilket är main-fil.

Utöver dessa filer finns även en kartfil. Programmet börjar med att den läses in. Den kan se ut som följande:

```
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
```

UME och BMA är två olika noder, så programmet gör är att den läser in denna filen och bygger upp alla noder som tillsammans blir denna kartan: (Där ser man vilka noder som är länkade till varandra).



Figur 1. Graf motsvarande airmap.map

För att kompilera programmet krävs följande namn och flaggor:

```
gcc -g -std=c99 -Wall -o is_connected *.c
```

Då kompileras alla .c filer i huvudmappen. När programmet ska exekveras skriver man ./isConnected samt eventuell filnamn på den textfil man tester. I detta fall blir det:

```
./is_connected 1-airmap1.map
```

Om allt fungerar som det ska är nästa steg att användaren ombeds att skriva in vilka destinationer man vill testa. Därefter får användaren svar om förbindelse finns eller inte, alternativt om noderna finns med. Exempel på testkörning visas i figur 2.

```
Amines-MacBook-Pro:OU5_Sommar aminebalta$ ./is_connected 1-airmap1.map
Enter origin and destination (quit to exit): UME GOT
There is a path from UME to GOT.

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

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

Enter origin and destination (quit to exit): quit
Normal exit.
Amines-MacBook-Pro:OU5_Sommar aminebalta$
```

Figur 2. Testkörning av programmet is_connected.c

3. Systembeskrivning

Huvudsyftet med hela programmet är att det ska läsa in en beskrivning av en riktad graf och sedan besvara frågor på om det finns en förbindelse mellan noderna. De datatyper som används i programmet är array_2d, queue och list. Programmet börjar med att skapa en datatyp av `char` som representerar en `node` och den innehåller också en `bool` som håller koll på om den är besökta eller inte. Därefter skapas två olika arrayer, en endimensionell som innehåller alla noder, samt en tvådimensionell som håller koll på alla förbindelser. En kö skapas också som används när programmet gör en bredden-först-sökning, sökningen körs när användaren frågar om det finns en förbindelse mellan två noder. Om använder skriver fel, exempelvis flygplatser som inte finns kommer det upp felmeddelanden. Om användare vill avsluta programmet skriver denne in ”quit”. Skriver användaren två flygplatser som finns i grafen kollas det som sagt efter en koppling, finns det en återkopplar programmet med att det finns en väg, om inte får man meddelande om det också. I båda fallen får användaren fortsätta skriva tills ”quit” anropas.

3.1. Algoritmbeskrivning

3.1.1. Läser in kartfil

1. Läser in textfilen som ett argument.
2. Öppnar filen, skriver ut ett felmeddelande om filen inte kan öppnas eller om ingen fil finns.
3. Börjar läsa från filen.
    3.1. Så länge det är en tom rad eller en hashtag så fortsätter den, men när den läser en siffra sparar den ner det som antalet noder.
    3.2. Om antalet noder är 0, så skrivs felmeddelande ut och programmet stängt av.

3.1.2. Skapa graph

1. Skapar en tom graf genom att skapa en instans av datatypen graph.
2. Gör plats för antal noder gånger två, eftersom att en koppling mellan två flygplatser måste skrivas två gånger. exempelvis BMA -> UME och UME -> BMA.
    2.1 Om den nyskapade grafen inte är tom skrivs ett felmeddelande ut och programmet stängs av.
3. Sätter in unika noder i grafen genom att söka igenom kartfilen radvis.
4. Allokerar minne för de två olika noder som läses in och sparar därefter ner dem i varsin variabel, sökningen bortser från tomma rader samt kommentarer (#).
5. Frigör minne som inte används.
6. När filen är läst stängs den ner.

3.1.3. Bredden-först-sökning

1. Läser användarens input, alltså startnod och slutnod. Först när slutnoden är inskriven och användare tryckt enter börjar programmet söka om det finns en koppling mellan noderna.

1.1. Om någon av noderna inte existerar i kartfilen, skrivs ett felmeddelande ut och användaren ombeds skriva in en nod som existerar.
1.2. Om noderna är lika ska ett felmeddelande skrivas ut och användaren får ett nyt försök.
1.3. Om noderna har en koppling får användaren information om det.
1.4. Om endast en nod skrivs in får användaren ett felmeddelande och får därefter ett nytt försök.

2. Programmet fortsätter tills användaren skriver ”quit” och den får också information om att programmet stänges av på ett korrekt sätt.

3.2 . isConnected

```
int main(int argc, char *argv[])
```

Funktionen för main är att den kör hela programmet. De huvudfunktionerna för main är att läsa in en textfil samt skapa en graf av informationen tagen från textfilen, därefter hantera och läsa in input från användaren och slutligen göra en bredden-först-sökning utifrån de användarinput som programmet fått in. Den har också ansvar för att bortse från kommenterade samt blanka rader, och även hantera minne.

Funktioner som anropas i main():
- blank_line()
- comment_line()
- graph_empty()
- graph_is_empty()
- graph_find_node()
- graph_insert_node()
- graph_insert_edge()
- graph_find_node()
- find_path()
- graph_kill()

```
bool find_path(graph *g,node *src,node *dest)
```

Skapas en ny tom kö genom att anropa där noder kan sätts in när de har besökts, alla besökta noder sätts som besökta och sätts in i kö. Så länge det finns grannar till varje nod som är obesökta så fortsätter den att travestera igenom. Sedan när sökningen är klar förstörs listan. När en en destination har hittats så förstörs listan av alla grannar, därefter återställer man grafen och dequeue traveseringskön. Sedan returneras `true` om destinationen hittades, annars `false`.

Funktioner som anropas i find_path():
- queue_empty()
- graph_node_set_seen()
- queue_enqueue()
- queue_is_empty()
- queue_front()
- queue_dequeue()
- graph_neighbours()
- dlist_first()
- dlist_is_end()
- dlist_inspect()
- nodes_are_equal()
- graph_node_is_seen()
- graph_node_set_seen()
- dlist_next()

- dlist_kill()
- graph_reset_seen()
- queue_kill()

`int first_character(const char *c)`
Funktionen används för att returnera position av det första värdet som inte är mellanslag eller om endast mellanslag hittas returneras -1.

Funktioner som anropas i first_character():
- isspace()

`int last_character(const char *c)`
Funktionen används för att returnera sista platsen som är `char`. Eller så returneras -1 om endast mellanslag hittas.

Funktioner som anropas i last_character():
- isspace()
- strlen()

`bool blank_line(const char *c)`
Om randen endast innehåller mellanslag returneras `true`, annars `false`.

Funktioner som anropas i blank_line():
- first_character()

`bool comment_line(char *c)`
Returnerar `true` om det är en kommenterad rad, det vill säga att första tecknet som inte är ett mellanslag är '#'.

Funktioner som anropas i comment_line():
- first_character()

3.3 Graph

Nedan beskrivs de relevanta delarna och funktionerna i hur grafen fungerar. Grafen används för att definiera och deklarera metoderna som som krävs för att kunna modifiera samt skapa en graf från en inläst fil. Syftet är att samla all funktionalitet som kärvs för att underhålla hur grafen ska fungera.

`struct node`
Det skapades en struct för noderna för att kunna återanvända data samt hålla reda på den på ett bra sätt. Det kan även förenkla minneshanteringen eftersom att den struct som skapades kunde återanvändas.

`bool nodes_are_equal(const node *n1,const node *n2)`
För att kolla om två noder är samma nod, det vill säga lika. Noderna jämför och returnerar `true` om de är lika.

```
graph *graph_empty(int max_nodes)
```

Skapar en ny tom graf som har en tvådimensionell matris, för att skapa den anropas funktionen `array_2d_create`.

`bool graph_is_empty(const graph *g)`
Kollar om en graf är tom, alltså inte innehåller några noder, det gör den genom att kolla på första indexet i matrisen, $(0, 0)$, och sedan iterera för varje index.

`graph *graph_insert_node(graph *g, const char *s)`
Sätter in nya noder i grafen, strängen kan max innehålla 40 symboler. Undviker sedan dubbletter genom att kontrollera om noden redan existerar någon annanstans. Loopar på samma sätt som tidigare med hjälp av `array_2d` tills den hittar första lediga platsen där det sedan sätts in en ny unik nod.

`node *graph_find_node(const graph *g, const char *s)`
Hittar noder som finns i grafen genom att skriva in namnet på noden man vill hitta, kollar igenom de befintliga noderna och jämför det med input-noden, jämförelsen sker med hjälp av `strcmp`. Även här används matrisen `array_2d` för att loopa igenom matrisen efter noderna, i detta fall användes `array_2d_has_value` som ska köras i loopen tills det inte finns några värden kvar i raden. Samt `array_2d_inspect_value` för att kolla noderna där man är i processen.

`graph *graph_reset_seen(graph *g)`
Man vill ändra status på alla noder tillbaka till osedda efter varje sökning, denna gjordes på liknande sätt som innan med tvådimensionella matrisen genom två loopar som itererar varje rad samt varje kolumn i varje rad. Varje nod inspekteras och sedan sätts den till ett osett värde, `visited = false`.

`graph *graph_insert_edge(graph *g, node *n1, node *n2)`
För att sätta in grannar i grafen vill man kolla alla rader i matrisen med hjälp av `array_2d_has_value` och sedan inspektera noden den befinner sig på med hjälp av `array_2d_inspect_value`, där jämförs ursprungsnoden med input-noden. Sedan kontrolleras dubblett, varefter det kontrolleras om index är tomt och då sätt nya noden in i matrisen med hjälp av funktionen `array_2d_set_value`.

`dlist *graph_neighbours(const graph *g,const node *n)`
Vill returnera en lista av grannar till noden, för att hitta den specifika listan måste den aktuella noden anropas. Grannarna i dennes kolumn måste då läggas in i en lista. Då ska först en tom lista skapas, sedan skapas en position som blir första positionen, sedan måste den aktuella noden hittas. Vilket den gör med hjälp av en loop som itererar genom raderna, det görs som innan med `array_2d_has_value`. För varje nytt index som hämtas med `array_2d_inspect_value` jämförs den inmatade noden med den noden som är aktuell, jämförelsen sker med hjälp av `nodes_are_equal`. Sedan loopas kolumnerna och varje granne som finns hämtas och läggs i den tomma listan, när en nod läggs till i listan ändras positionen som sattes i början.

`void graph_kill(graph *g)`
När listan ska raderas, när minnet ska frigöras måste programmet gå igenom varje rad och kolumn, och för varje index som man går igenom måste det kontrolleras om det finns ett värde, vilket görs med hjälp av `array_2d_has_value` och om det finns hämtas värdet med hjälp av

`array_2d_inspect_value` och om noden och kodnamnet finns ska den frigöras men hjälp av `free()`.

4. Testkörningar

I figur 3 visas testkörningar av programmet is_connected.c som gjordes för att säkerställa att programmet beter sig rätt, dessa tester är manuellt gjorda och kan därför inte upptäcka tyngre fel. Men de olika scenarion som testats i följande ordning är om förbindelser mellan två noder kan hittas, om det inte finns en förbindelse och hur det visar sig, om en destination inte finns, om en förbindelse inte finns samt när användaren vill avsluta programmet.

```
> ./isConnected airmap1.map
Enter origin and destination (quit to exit): BMA BMA
Airports is equal, try again.

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

Enter origin and destination (quit to exit): BMA ARN
Airport did not exist in map, try again.

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

Enter origin and destination (quit to exit): quit
Normal exit.
Program ended with exit code: 0
```

Figur 3. Testkörning av programmet is_connected.c

5. Arbetsfördelning

Vi delade upp laborationen i två delar, isConnected och graph.c där var och en fick ansvar för varsin del. En stor del av uppgiften par-programmerades. Vi hade under hela implementationsfasen en öppen dialog där vi informerade den andra om vad som hade gjorts. Git användes för att vi skulle kunna arbeta var för sig men också för att på ett enkelt och smidigt sätt få ihop de arbetet som var och en gjorde till ett gemensamt program.

6. Reflektioner

Största utmaningen med denna laborationen var att komma igång med implementationen av programmet. Första utmaningen var att ta sig igenom laborationsspecifikationen då det var en hel del information att ta in och förstå.

Då vi båda läst kursen tidigare tycker vi att uppgiften har förbättrats och blivit mycket tydligare, men trots detta är det en omfattande uppgift som kräver en rätt så hög kunskap för programmering C anser vi. Med tanke på att vi genomförde laborationen under sommaren så har vi haft en begränsad tillgång till handledning vilket har medfört att vi under vissa perioder har haft svårt att gå vidare när vi stött på problem.

Trots detta känner vi verkligen att vi har lärt oss otrolig mycket under laborationens gång och den roligaste delen var när programmet började visa livstecken.

7. Förändringar

1. En innehållsförteckning har lagts till i början på rapporten.
2. Beskrivning av hur grafen fyller sitt syfte har lagts till precis i början under under 3.3.
3. En punktlista för varje funktion som anropar vilka har lagts till under 3.2.
4. Version och syfte för både `graph.c` och `is_connected.c` har lagts till i första filkommentaren i vardera kodfil.