Report **29 juli 2025**

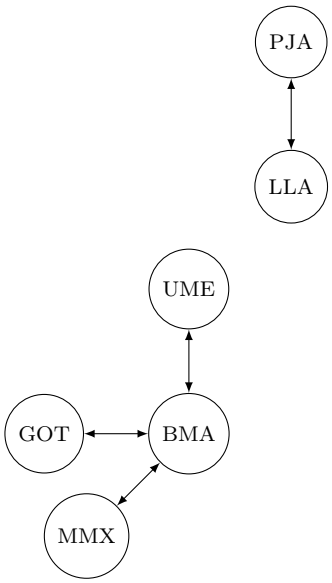
5DV149 Datastrukturer och algoritmer

Assignment 4 — Grafer och grafalgoritmer

version 1.0

Name	Jakob Nyström, Marc Meunier, Ellen Horneij
Username	tfy22jnm, tfy22mmr, tfy22ehj

Innehåll



Figur 1: En visuell interpretation av innehållet i filen `airmap1.map`

0 Versionshistorik

v1.0 2025-05-25 Första Inlämningen submission.

v2.0 2025-06-02 Andra inlämningen, fixat litet fel i graph reset seen.

v3.0 2025-07-29 Tredje inlämningen, fixat så att en felaktig kartfil ger felmedelande och programmet avslutas.

1 Introduktion

Målet med uppgiften var att implementera två olika varianter av den abstrakta datatypen graf. En graf består av noder som är sammanlänkade av kanter. En graf kan användas för många olika applikationer men en av de vanligare är att noderna beskriver platser och att kanterna beskriver hur dessa platser är sammanlänkade. Kanterna kan ha olika vikt för att beskriva olika egenskaper som t.ex. avståndet mellan platserna. I den applikationen av graph som används i denna uppgift har vi inga vikter. Applikationen är att visa hur flygrutter sammanlänkar flygplatser som i figur 1.

2 Användarguide

2.1 Kompilering

För att köra koden i `is_connected.c` behöver den kompileras med grafimplementationen som används och alla andra abstrakta datatyper som används. I våra implementationer används antingen `graph.c` eller `graph2.c`, `dlist.c`, `list.c`, `queue.c` och `array_2d.c`. Nedan följer en generel version av kommandot som används för att kompilera koden. Koden kompilerades med gcc.

```
gcc -o is_connected -std=c99 -Wall -I .../path/to/include
is_connected.c graph.c .../path/to/other/files.c
```

2.2 Filformat

Filformatet som användes i uppgiften följde en specifik form. Filen får innehålla både tomma rader och kommentarer som börjar med # som ska ignoreras. Först ska antalet kanter i grafen speciferas med ett heltal. Sedan ska resterande rader innehålla en kant var med startnod följt av ett mellanslag och destinationsnod. Nodnamnen innehåller endast alfanumeriska tecken och överstiger inte 40 tecken. Nedan visas filen som genererar grafen i figur 1.

```
# 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
```

2.3 Testkörning

När programmet körs så kommer den automatiskt att ladda in den förvalda filen och konstruera grafen. Programmet kommer att skriva ut följande meddelande.

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

Efter detta meddelande ska användaren skriva in två nodnamn som består av tre bokstäver med ett mellanslag mellan. Programmet returnerar då om det finns en väg mellan de två angivna noderna och skriver ut exempelvis följande meddelande.

```
There is a path from MMX to UME.
```

När programmet returnerat svaret så ställs frågan om på nytt. Programmet körs tills användaren skriver `quit`. Då avslutas programmet.

3 System description

3.1 Datastrukturer

Uppgiften består av tre olika kodfiler. Ett huvudprogram som har i uppgift att läsa in filen, bygga upp grafen och hantera inmatningar från användaren samt två olika implementationer av den abstrakta datatypen graf. Båda grafimplementationerna följer den givna gränsytan för datatypen.

3.1.1 Gränstytan till graf

Tabell 1: Gränsytan för en implementation av den abstrakta datatypen graf.

`graph_empty(v)`	Returnerar en tom graf med storleken `v`.
`graph_is_empty(g)`	Returnerar `true` om grafen `g` är tom, annars `false`.
`graph_has_edges(g)`	Returnerar `true` om grafen `g` har kanter, annars `false`.
`graph_insert_node(g,s)`	Insätter en nod med värdet `s` i grafen `g`.
`graph_has_node(g,s)`	Returnerar en nod med värdet `s` från grafen `g`, `NULL` om den inte finns.
`graph_is_seen(g,n)`	Returnerar seen-statusen för noden `n` i grafen `g`.
`graph_set_seen(g,n,b)`	Uppdaterar seen-statusen för noden `n` i grafen `g` med värdet `b`.
`graph_reset_seen(g)`	Uppdaterar seen-statusen till `FALSE` för alla noder i grafen `g`.
`graph_insert_edge(g,n1,n2)`	Insätter en kant mellan noderna `n1` och `n2` i grafen `g`.
`graph_delete_node(g,n)`	Tar bort noden `n` ur grafen `g`.
`graph_delete_edge(g,n1,n2)`	Tar bort kanten mellan `n1` och `n2` ur grafen `g`.
`graph_neighbours(g,n)`	Returnerar en lista med alla grannar till `n` i grafen `g`.
`graph_kill(g)`	Returnerar allt dynamiskt minne som använts av grafen `g`.
`graph_print(g)`	Skriver ut den givna grafen `g`.

3.2 Graf

Den första grafimplementationen använder sig av ett tvådimensionellt fält för att hålla informationen om noderna. Det är implementerat i en `struct graph` som innehåller själva matrisen och tre räknare som räknar antalet noder, kanter och det största tillåtna antalet noder och kanter. Matrisen innehåller noderna på första kolumnen. Varje nod består av en `struct node` som har ett värde, en position och en status som visar om noden har setts. Positionen används för att indexera matrisen. Så när noderna läggs

till i matrisen får de första lediga position. På raderna läggs pekare till noderna som är granne till den i första kolumnen. De läggs på kolumnen som har samma index som sin position Eftersom positionen för alla noder och grannar är känd så kan man indexera direkt utan att söka igenom matrisen när man t.ex. vill ta bort en granne.

3.2.1 Graph #2

Den andra implementationen av datatypen graf har en liknande grundstruktur mot den första implementationen med en `struct graph` som innehåller strukturen för att hålla noder och dess räknare. Men i denna implementationen så hålls noderna i en riktad lista istället för det tvådimensionella fältet som används i den första implementationen. I den nodlistan ligger `stuct node` som också innehåller varsin riktad lista där grannarna till noderna läggs in på ett sätt som liknar den första implementationen.

3.2.2 Kö

En implementation av den abstrakta datatypen kö användes för att implementera en bredden först algoritm. Datatypen är en först in, först ut lista där det första objektet i kön är det som tas ut först. Gränsytan för den implementationen av kö som användes presenteras i tabell 2.

Tabell 2: Gränsytan för en implementation av den abstrakta datatypen kö.

`queue_empty()`	Returnerar en tom kö.
`queue_is_empty(q)`	Returnerar `true` om kön `q` är tom, annars `false`.
`queue_enqueue(q,v)`	Lägger till värdet `v` längst bak i kön `q`.
`queue_dequeue(q)`	Tar bort och returnerar det första elementet i kön `q`.
`queue_first(q)`	Returnerar första värdet i kön `q`.
`queue_kill(q)`	Frigör allt dynamiskt minne som använts av kön `q`.
`queue_print(q)`	Skriver ut innehållet i kön `q`.

3.3 Riktad lista

En implementation av den abstrakta datatypen riktad lista användes i den andra implementationen av datatypen graf och för att skapa listan av grannar till noderna. Datatypen är dynamiskt allokerad och traverseras med hjälp av positioner från första till sista. Värden kan stoppas in var som helst i listan förutsatt positionen är känd sedan innan.

Tabell 3: Gränsytan för en implementation av riktad listan.

`dlist_empty()`	Returnerar en tom lista.
`dlist_is_empty(l)`	Returnerar `true` om listan `l` är tom, annars `false`.
`dlist_first(l)`	Returnerar positionen framför det första elementet i listan `l`.
`dlist_insert(l,v,p)`	Lägger till värdet `v` på positionen `p` i `l`.
`dlist_remove(l,p)`	Tar bort värdet på positionen `p` från listan `l`.
`dlist_next(l,p)`	Returnerar nästa position efter `p` i listan `l`.
`dlist_is_end(l,p)`	Kontrollerar om positionen `p` är den sista i listan `l`.
`dlist_inspect(l,p)`	Inspekterar värdet på position `p` i listan `l`
`dlist_pos_is_valid(l,p)`	Kontrollerar om positionen `p` är giltig i listan `l`
`dlist_pos_is_equal(l,p1,p1)`	Kontrollerar om positionen `p1` är samma som `p2` i listan `l`
`dlist_kill(l)`	Frigör allt dynamiskt minne som används av listan `l`.
`dlist_print(l)`	Skriver ut innehållet i listan `l`.

3.4 Tvådimensionellt fält

Tabell 4: Gränsytan för en implementation av tvådimensionellt fält.

`array_2d_create(lo1,hi1,lo2,hi2,kill_func)`	Skapar och returnerar en tvådimensionell array med index från lo till high med en `kill_func` för att avallokera innehållet.
`array_2d_kill(a)`	Frigör allt minne som använts av fältet `a`.
`array_2d_set_value(a,v,i,j)`	Sätter värdet `v` vid positionen `(i, j)` i fältet `a`.
`array_2d_inspect_value(a, i, j)`	Returnerar värdet vid positionen `(i, j)` i arrayen `a`.
`array_2d_low(a,d)`	Returnerar lågt index för dimensionen `d` i fältet `a`.
`array_2d_high(a,d)`	Returnerar högt index för dimensionen `d` i fältet `a`.
`array_2d_has_value(a,i,j)`	Kontrollerar om det finns ett värde på positionen `(i, j)` i fältet `a`.
`array_2d_print(arr)`	Skriver ut arrayen `arr` i tabellformat.

3.5 Algoritmer

I koden för `is_connected.c` används bredden först algoritmen för att söka igenom grafen, en algoritm för att läsa av och konstruera grafen samt en huvudfunktion.

3.5.1 Inläsning av fildatat och konstruktionen av grafen

Algoritmen för att läsa in datat gör i stora drag en inläsning av varje rad, insättning av de båda noderna och sedan insättning mellan noderna och sig själva.

```
algorithm g <- graph_from_file()

// Read the file
File = open_file("filename")

If File = NULL do
    // File did not read correctly exit program
    exit
data <- read_next_line(file)

// Skip comments and empty lines
while line_is_blank(data) or line_is_comment(data) do
    data <- read_next_line(file)

// read the number of edges to insert
nr_of_edges <- data
g <- graph_empty(nr_of_edges)

// Do the insertion for all edges
for i = 0, i < nr_of_edges, i <- i + 1 do
    data <- read_next_line(file)

    // Reads the current line on the expected format
    n1, n2 <- parse_data(data)

    // Insert into the graph
    g <- graph_insert(g, n1)
    g <- graph_insert(g, n2)

    // Read the nodes from the graph
    m1 <- graph_find_node(g, n1)
    m2 <- graph_find_node(g, n2)

    // Exit with error message if the nodes do not exist
    if m1 or m2 is NULL do
        exit

    // Insert edges
    g <- graph_insert_edge(g, m1, m1)
    g <- graph_insert_edge(g, m2, m2)
    g <- graph_insert_edge(g, m1, m2)

// Close the file and return the finished graph
close_file(file)
return g
```

3.5.2 `find_path`

Algoritmen `find_path` använder sig av bredden först traversering för att traversera grafen för att se om källnoden och destinationsnoden har en väg mellan varandra.

```
Algorithm b = find_path(src, dest: node, g: graph)

// Mark the starting node as seen
(src, g) <- Set-seen(src, g)
// Put it in an empty queue
q <- Enqueue(n, Queue-empty())

while not Isempty(q) do

    // Pick first node from queue
    src <- Front(q)
    q <- Dequeue(q)

    // If nodes_are_equal(src, dest)
        return true

    // Get its neighbours
    neighbour-set <- Neighbours(src, g)
    for each neighbour b in neighbour-set do
        if not Is-seen(b,g) then
            // Mark unseen node as seen and put it in the queue
            (b, g) <- Set-seen(b, g)
            q <- Enqueue(b,q)
```

4 Reflektioner

4.1 Arbetsfördelning

Eftersom vi var tre personer i gruppen och det fanns tre koder att skriva så delade vi upp så att alla i huvudsak skrev en kod var. Sedan fanns det ett visst överlapp och vi hjälpte varandra med de olika koderna, speciellt felsökning där de gick snabbast att förklara vad man gjort för en annan gruppmedlem. Felsökningen och testningen av koden gav oss andra bra förståelse i hur de andra gruppmedlämmarna tänkt när de skrivit koden och såg också till att vi alla fick en förståelse för varandras kod.

4.2 Reflektioner

Det var en intressant uppgift som var mindre utmanande än förväntat. Vi drog många lärdomar från OU3 och de misstagen vi begick där vilket gjorde att kodningen gick

lättare. Det gick även snabbare att felsöka koden. Grafimplementationerna var snarlika OU3 men `is_connected` var någonting helt nytt. Det var kul att få lära sig filhantering i c och implementationen av en bredden-först algoritm. Överlag en kul uppgift.