UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport obligatorisk uppgift

Datastrukturer och algoritmer (C) 7.5 p
5DV149

Obligatorisk uppgift nr

4

Namn	Jonas Sjödin
E-post	id16jsn @cs.umu.se
Datum	2017-03-17
Handledare	Daniel Harr, Jakob Lindqvist, Jakob Vesterlind, Sebastian Sandberg

1. Introduktion

1.1 Bakgrund

I denna obligatoriska uppgift i kursen *datastrukturer och algoritmer* har uppgiften varit att se ifall olika noder i en graf är sammankopplade efter inläsning av en fil. För att lösa detta problem ska programmet ta in input från användaren angående vilka noder den vill kolla ifall de är sammankopplade. Sedan traverseras grafen med bredden först för att hitta ifall noderna är sammankopplade.

1.2 Förutsättningar

- Inga minnesfel eller minnesläckor
- Endast läsa igenom kartfilen en gång
- Huvudprogrammet ska finnas i filen isConnected.c
- Programmet ska kunnas kompilera med flaggorna `-std=c99`, `-Wall`, `-Werror`

2. Användarhandledning

2.1 Krav på indata/kartfil

För att programmet överhuvudtaget ska kunna köras måste användaren tillhandahålla filen map1.map som ska ligga i projektmappen. Filen ska se ut på följande sett:

```
<antal kanter>
<startnod> <slutnod>
<startnod> <slutnod>
<startnod> <slutnod>
<startnod> <slutnod>
…
```

T.ex.

```
4
Nod1 Nod2
Nod2 Nod3
Nod3 Nod4
Nod4 Nod5
```

2.2 Krav för körning av programmet

För att kunna köra programmet ska först filen kompileras med någon kompilator, t.ex. GCC. För att kompilera i GCC skriver man:

```
c99 -Wall -Werror -g isConnected.c dlist.c graph.c queue.c list_2cell.c -o isConnected
```

Sedan kan användaren köra programmet med kommandot ./isConnected map1.map

2.3 Utdata

Användaren möts då av en textrad som säger:

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

Nu ska användaren skriva in två noder, t.ex:

"Nod1 Nod2"

Om noderna är sammankopplade får användaren svaret:

"Nod1 and Nod2 are connected"

Om noderna inte är sammankopplade får användaren svaret:

"Nod1 and Nod2 are not connected"

För att avsluta programmet måste användaren skriva: "quit"

2.4 Exempel

```
Enter origin and destination (quit to exit):Nod1 Nod2

Nod1 and Nod2 are connected

Enter origin and destination (quit to exit):Nod1 Nod8

Nod1 and Nod8 are not connected

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

3. Systembeskrivning

3.1 Systembeskrivning graph.c och graph.h

Min graf-fil, graph.c och graph.h har jag själv skapat. Jag har använt Johan Eliassons fil graph_nav_directed.c[1] och graph_nav_directed.h som inspiration och riktlinje när jag skapade min graf-fil. Min graf-fil använder sig av dlist.c och dlist.h och queue.c och queue.h som använder sig av list_2cell.c och list_2cell.h. Alla dessa filer finns att hämta från samma ställe som graph_nav_directed.c. Då jag inte behövde hela gränsytan för att lösa uppgiften implementerad jag endast de delar som jag ansåg behövdes. En delfrån gränsytan, `choose-node` hade jag kunnat implementera då den funktionen används på flera ställen i graph.c men jag valde att inte göra det då jag inte ansåg att det behövdes. T.ex. i traverseringsfunktionen skulle den kunnat ha använts men då hade man behövt köra två forloopar för att hitta värdet istället för en. Därför valde jag att inte implementera den. Jag har skapat tre abstrakta datatyper som programmet huvudsakligen använder sig av: `node`, `edge` och `vertex`. node innehåller en vertex, en lista på grannar och bool, visited som talar om ifall noden har besökts under traversering.

Gränsytan är hämtad från boken Datatyper och Algoritmer

(Janlert, Wiberg, 2000, s.339).

3.1.1 Gränsyta för riktad lista

Om funktionen är fetmarkerad betyder det att den är använd i min implementation.

`Empty() -> Graph(node, edge)`

`Insert-node(v:node, g:graph(node,edge)) -> Graph(node, edge)`

`Insert-egde(e:edge, g:graph(node,edge)) -> Graph(node, edge)`

`Isempty(Graph(node, edge)) -> Bool`

`Has-no-edge(g:Graph(node, edge)) -> Bool`

`Choose-node(g:Graph(node, edge)) -> node`

`Neighbours( v: node, g:Graph(node, edge)) -> Set(node)`

`Delete-node(v:node, g:Graph(node,edge)) -> Graph(node, edge)`

[1] (http://www8.cs.umu.se/kurser/5DV149/material_vt17/datatypes/index.html)

3.1.2 Funktioner

I header-filen deklarerar jag tre funktioner som skapas i graph.c. De är:

```
1. typedef bool compareFunc(vertex v1, vertex v2);
2. typedef void freeEdgeFunc(data d);
3. typedef void freeVertexFunc(vertex v);
```

1. Funktionen `compareFunc` tar in två vertexar och kollar om de är lika och returnerar true om de är det, annars false.
2. Funktionen `freeEdgeFunc` hör till minneshanteringen av kanter
3. Funktionen `freeVertexFunc` hör till minneshanteringen av vertexar

`bool graph_isEmpty(graph *myGraph);`
Denna funktion kollar om en graf är tom. Om den är det returnerar den "true" annars "false"

`graph *graph_empty(compareFunc *cf);`
Denna funktion skapar en tom graf.

`void graph_setVertexMemHandler(graph *myGraph, freeVertexFunc *f);`
Denna funktion sätter en minneshanterare på vertexar i en graf.

`void graph_setEdgeMemHandler(graph *myGraph, freeEdgeFunc *f);`
Denna funktion sätter en minneshanterare på kanter i en graf.

`void graph_freeEdge(graph *myGraph, edge e);`
Denna funktion friar minnet för en kant.

`bool graph_insertNode(graph *myGraph, vertex vert);`
Denna funktion kollar ifall grafen innehåller en likadan vertex som den tar som argument. Om den gör det returnerar den true, om den inte gör det så sätter funktionen in en nod i en angiven graf med en angiven vertex samt sätter en minneshanterare på noden och returnerar false.

`void graph_insertEdge(graph *myGraph, edge e);`
Denna funktion skapar en kant mellan två noder genom att ta en edge e som argument.

`bool graph_nodeExistInGraph(graph *myGraph, vertex v1);`
Denna funktion kollar om en nod finns i en graf. Den returnerar true om den gör det annars false.

`bool graph_traverseBTF(graph *g, vertex v1, vertex v2);`
Denna funktion traverserar grafen med bredden först för att se om det finns en väg att gå från v1 till v2.

`void graph_setVisitedFalse(graph *myGraph);`
Denna funktion sätter alla noders visited bool till false.

void graph_deleteNode(graph *myGraph, node *n);

Denna funktion tar bort en nod och alla dess komponenter samt friar allt det minne som den allokerat.

`void graph_free(graph *myGraph);`
Denna funktion tar bort hela grafen och friar dess minne.

Ytterligare kommentarer till hur de olika funktionerna ska användas finns i graph.c och graph.h filen.

3.2 Systembeskrivning isConnected.c

Denna fil har jag också skapat själv. Den använder sig av graf-filen för att skapa en graf, fylla den med noder och kanter och sedan köra själva programmet där användaren kollar om noderna finns i programmet.

`int main(int argc, char **argv) {`
Denna funktion kör hela programmet (som main-funktioner brukar göra). Först skapar den en graf. Sedan sätter den minneshanterare på grafen. Sedan läser den in filen och stoppar in alla noder och kanter som finns i filen i grafen. Efter detta stängs filen och användarprogrammet körs där användaren ger input på vilka noder som ska kollas om de är sammankopplade. När användaren är färdig frias minnet för grafen och programmet stängs.

`bool compareString(void *string1,void *string2){`
Denna funktion jämför två strängar och returnerar true om de är lika. Annars false.

`FILE *fetchFile(char *filePath){`
Denna funktion hämtar filen som användaren skriver att den ska hämta.

`void insertNodesAndEdges(graph *myGraph, FILE *mapFile){`
Denna funktion läser in från filen och sätter in dess noder och kanter i grafen. Om noderna redan finns i grafen friar den istället minnet.

bool nodesExists(graph *myGraph, vertex node1, vertex node2){
Denna funktion kollar ifall noderna finns i grafen med hjälp av min funktion graph_nodeExist. Om en eller två noder inte finns i grafen printar den det och returnerar false. Om startnoden och slutnoden är samma printar den det och returnerar false och om de båda noderna finns i grafen så returnerar den true och runProgram printar ifall de är sammankopplade eller inte.

void runProgram(graph *myGraph){
Denna funktion kör själva programmet som användaren interagerar med. Den tar input från användaren och kör kollen nodesExists för att se om de finns i grafen. Om de gör det printar den ifall de är sammankopplade eller inte. För att avsluta skriver användaren “quit”.

3.2 Systembeskrivning dlist.c, dlist.h och queue.c, queue.h

För att implementera grafen och bredden-först traverseringen använde jag mig av två färdiga filer abstrakta datayper riktad lista och kö där köimplementationen använders sig av list_2cell. Då jag inte skapat dessa funktioner men använder de så kommer här en kort beskrivning av dem.

```
void queue_dequeue(queue *q);
```
Denna funktion tar bort första värdet ur en kö.

```
void queue_enqueue(queue *q);
```
Denna funktion lägger till ett värde först i kön.

```
data queue_front(queue *q);
```
Denna funktion returnerar första värdet i kön.

```
queue queue_empty();
```
Denna funktion skapar och returnear en tom kö.

```
void queue_free(queue *q);
```
Denna funktion friar det allokerade minnet för kön.

```
void dlist_setMemHandler(dlist *l, memFreeFunc *f);
```
Denna funktion sätter en minneshanterare på en riktad lista.

```
dlist *dlist_empty(void);
```
Denna funktion skapar en tom riktad lista.

```
dlist_position dlist_first(dlist *l);
```
Denna funktion returnerar första poisitionen i listan.

```
dlist_position dlist_next(dlist *l,dlist_position p);
```
Denna funktion returnerar nästa position i listan.

```
bool dlist_isEmpty(dlist *l);
```
Denna funktion kollar om listan är tom. Om den är tom returnerar den true, annars false.

```
dlist_position dlist_insert(dlist *l, dlist_position p,data d);
```
Denna funktion sätter in ett värde i en lista på en viss position och returnerar positionen.

```
dlist_position dlist_remove(dlist *l, dlist_position p);
```
Denna funktion tar bort ett värde ur en lista på en viss position och returnar positionen.

```
void dlist_free(dlist *l);
```
Denna funktion friar minnet för en lista.

```
data dlist_inspect(dlist *l, dlist_position p);
```
Denna funktion inspekterar en position och returnerar datan på den positionen.

```
bool dlist_isEnd(dlist *l, dlist_position p);
```
Denna funktion kollar om positionen p är den sista positionen i listan.

4. Algoritmbeskrivning

Jag har skissat upp algoritmer på papper för att kunna skapa mina olika funktioner så snabbt och enkelt som möjligt och underlätta för mig när jag sedan debuggade och försökte fixa minnesläckorna. Jag använde mig dock av en färdig c++kod/pythonkod/Javakod/algoritm i breddenförst-traverseringen.[2]

[2] http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/

5. Komplexitetsanalys

För denna komplexitetsanalys tar jag och fokuserar på 6 funktioner som jag har skapat som tar olika lång tid beroende på hur många element som lästs in från filen. Nedan står deras bästafallskomplexitet, medelfallskomplexitet och värstafallskomplexitet. Komplexiteten är beräknad sett till hur många gånger loopar körs som är beroende på antalet inlästa noder och kanter och inte på konstanter då de tar lika lång tid varje gång och får därför O(1). En loop som beror på antalet noder får komplexiteten O(n) och en loop som beror av antalet noder inuti en loop som beror av antalet noder får då komplexiteten $O(n^2)$.

Funktion	Bästafall	Medelfall	Värstafall
graph_traverseBTF	O(n)	$O(n^2)$	$O(n^2)$
graph_insertNode	O(1)	O(n)	O(n)
graph_insertEdge	O(1)	O(n)	O(n)
graph_free	O(1)	$O(n^2)$	$O(n^2)$
insertNodesAndEdges	O(1)	$O(n^2)$	$O(n^2)$
runProgram	O(n)	$O(n^2)$	$O(n^2)$

6. Testkörning

För att testköra programmet har jag ofta debuggat programmet i min utvecklingsmiljö som är Clion[3] med debuggern GDB[4]. Där har jag tillexempel kollat minnesadresser för att se om noderna jag lägger in i grannlistorna är en faktisk nod och inte baran namnet på en nod osv. Jag har även printat info om noderna i terminalen för att t.ex. se vilka grannar de har eller vilken som är den aktiva noden. När jag sedan fått programmet att fungera började jag köra valgrind[5] för att kolla minnesläckor och minnesfel. Där upptäckte jag en hel del fel. Jag tog då och först fixade minnesläckorna genom att kommentera bort olika delar för att fokusera på små delar, man kan säga att jag använde problemlösningsstrategin devide and conquer. När jag sedan löste de små delarna la jag ihop de och då hade jag inga minnesläckor utan bara ett gäng minnesfel. Dessa tog jag och behandlade så att jag än en gång delade upp delarna för att hitta var felen uppstod och sedan fixade jag till det så så att hela programmet kunde köras utan några kompileringsfel, minnesläckor eller minnesfel.

7. Reflektioner

Jag är nöjd över mitt arbete då jag hann lösa uppgiften i tid men jag kunde ha börjat tidigare så att jag inte behövde stressa så mycket nu på slutet. En sak som jag hade kunnat ändra är som sagt att implementera choose-node funktionen från grafens gränsyta. Den hade kanske gjort så att jag fått några färre rader kod. J Jag har även lärt mig hur man ska bättre lägga upp ett arbete och skissa för att man inte ska behöva göra saker fler gånger samt att man ska läsa instruktionerna. T.ex. tre timmar före inlämningen insåga jag att man endast fick läsa från filen en gång och jag var tvungen att slå ihop två funktioner i isConnected, därav den lite fula freefunktionen av vertexarna i funktionen insertNodesAndEdges. Jag tycker att denna uppgift har visat det roliga med att koda närmare hårdvarunivå med minnesallokeringen.ag är dock som sagt väldigt nöjd med mitt arbete och min grafimplementation och jag har lärt mig otroligt mycket mer om språket c i denna uppgift än vad jag har gjort i någon annan del av kursen.

[3] https://www.jetbrains.com/clion/
[4] https://www.gnu.org/software/gdb/
[5] http://valgrind.org/