|  |
| --- |
| CXX01 |
| Application- Eindopdracht documentatie |
| Advanced C Programming |

|  |
| --- |
| Rene Schouten (0928619)  Raber Ahmad (0921954) |

# Opdracht

In ons vorig opdracht hadden we de Graph met succes ontworpen en geschreven. In de eindopdracht wordt de Graph en de DoubleLinkedList gebruikt om een Applicatie te maken die de kortste route kan bepalen tussen twee vertexes. De kortste route wordt berekend met behulp van de Bellman-Ford algoritme. In de vorige opdracht werd de Graph api en de rest ervan uitgebreid behandeld. In dit document wordt er alleen gefocust op de applicatie zelf.

# Bellman-Ford algortime

Het hele algoritme staat in een aparte functie in het main.c bestand. De functie van de algoritme ziet er als het volgende uit:

void bellmanFordAlg(Graph\* graphToSearch, Vertex\* start, Vertex\* destination){

De functie heeft als parameter een Graph pointer, dit is een pointer naar een graph met vertexes en weights. Daarnaast heeft het nog 2 vertex pointers genaamd start en destination. Tussen deze twee vertexes wordt er dan vervolgens de kortste route bepaald door het algoritme.

De bellman-ford algoritme kan als het ware in 4 onderverdelen worden verdeeld om met succes de kortste route tussen twee vetexes te vinden. De vier onderdelen zijn het volgende:

* Initialiseren van de graph
* Herhaald doorrekenen van de edges
* Controleren op negatieve weight cycli
* Printen van de resultaten

In de code wordt er ook onderscheid gemaakt tussen deze vier onderdelen.

## Initialiseren van de graph

//assigning infinity to all other vertexes excluding start vertex

DllNode\* vertexIterator = graphToSearch->vertices->head;

while (vertexIterator)

{

Vertex\* vertex = vertexIterator->data;

vertex->data = malloc(sizeof(bellmanFordVertexData));

if(start->ptrToNode == vertexIterator){

((bellmanFordVertexData\*)vertex->data)->distance = 0;

((bellmanFordVertexData\*)vertex->data)->previous = NULL;

}

else{

((bellmanFordVertexData\*)vertex->data)->distance = INT\_MAX;

((bellmanFordVertexData\*)vertex->data)->previous = NULL;

}

vertexIterator = vertexIterator->next;

}

In het stukje code hierboven wordt het eerste onderdeel uitgevoerd. Door heel de graph door worden alle vertexes op infinitief gezet. In de code schrijven we dat op als INT\_MAX. Dat is het grootste waarde die een signed integer kan hebben in ons programma. Alleen de start vertex wordt in dit geval niet op infinitief gezet maar op 0 gezet.

## Doorrekenen van de edges

//relaxing the edges

vertexIterator = graphToSearch->vertices->head;

EdgeIteratorData edgeIteratorData;

while(vertexIterator){

Edge\* currentEdge = setEdgeIterator(&edgeIteratorData,graphToSearch->vertices->head->data);

while(currentEdge){

Vertex\* sourceVertex = ((Vertex\*)edgeIteratorData.currentVertex->data);

Vertex\* destinationVertex = currentEdge->destination;

bellmanFordVertexData\* sourceData = sourceVertex->data;

bellmanFordVertexData\* destinationData = destinationVertex->data;

int vertexDistance = sourceData->distance;

int destinationVertexDistance = destinationData->distance;

int tmpDistance = vertexDistance + currentEdge->weight;

if(vertexDistance != INT\_MAX)

{

if(tmpDistance < destinationVertexDistance){

destinationData->distance = tmpDistance;

destinationData->previous = sourceVertex;

printf("- %d %s %s\n",currentEdge->weight,sourceVertex->name, destinationVertex->name);

}

}

currentEdge = nextEdge(&edgeIteratorData);

}

vertexIterator = vertexIterator->next;

}

In dit stukje code worden alle edges doorgerekend.

## Controleren op negatieve cyclisch

//negative loop check

bool negativeCycleFound = false;

Edge\* currentEdge = setEdgeIterator(&edgeIteratorData,graphToSearch->vertices->head->data);

while(currentEdge){

Vertex\* sourceVertex = ((Vertex\*)edgeIteratorData.currentVertex->data);

Vertex\* destinationVertex = currentEdge->destination;

bellmanFordVertexData\* sourceData = sourceVertex->data;

bellmanFordVertexData\* destinationData = destinationVertex->data;

if(sourceData->distance != INT\_MAX && sourceData->distance + currentEdge->weight < destinationData->distance)

{

printf("error the graph contains a negative cycle\n");

negativeCycleFound = true;

break;

}

currentEdge = nextEdge(&edgeIteratorData);

}

In dit stukje code wordt er gecheckt op negatieve cycli. Wanneer de vertex data + de weight van de huidige edge kleiner is dan de destination data dan kan er te maken zijn met een negatieve cycli. Indien dat wel zo is dan wordt de boolean “nagativeCycleFound” op waar gezet en wordt er gestopt met het loopen.

## Printen van de resultaten

//print the results

if(negativeCycleFound == false)

{

Vertex\* resultPath = destination;

while(resultPath)

{

printf("<- %s ",resultPath->name);

resultPath = ((bellmanFordVertexData\*)resultPath->data)->previous;

}

}

In het stukje code wordt de resultaten van de algoritme geprint in de terminal.

# FileToGraph API

Met behulp van de FileToGraph api wordt een graph volgeladen. De api is als het volgende opgebouwd.

void printNodes(nodeInfo \*n);

Hier worden de nodes één voor één uitgeprint in de terminal.

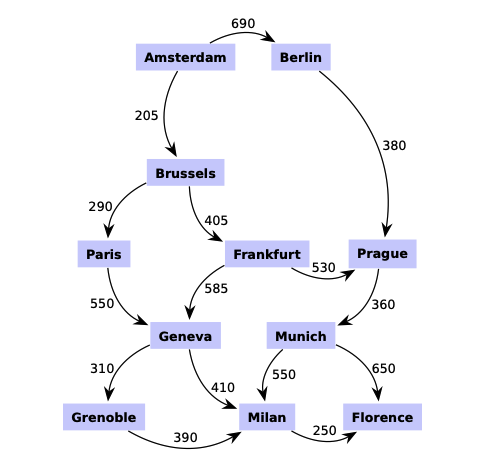
void printEdges(edgeInfo \*e);

De verbonden edges aan de vertexes worden allemaal uitgeprint.

Graph\* loadGraphFromFile(char\* filePath);

In dit stukje wordt een bestand gelezen en volledig als een graph pointer teruggegeven.

# Kortste route



Hiernaast zien we een klein overzicht van verschillende steden met daarbij de behorende weights. We kunnen nu ons programma en algoritme gebruiken om de kortste afstand te bepalen tussen verschillende steden te vinden. Als voorbeeld willen we de kortste afstand weten van Amsterdam naar Florence.

Als eerst maken we een graph aan en daarbij laden we een json bestand in die in een bepaald formaat geschreven is. De json bestand is een kopie van de rechter graph/foto, deze wordt in ons eigen programma geladen. Het json bestand kunnen we terugzien in de rechter afbeelding. De vertexes hebben allemaal een eigen unieke id-nummer met een bijbehorende label. Daarna worden alle verbindingen tussen de vertexes neergezet met bijbehorende weights.

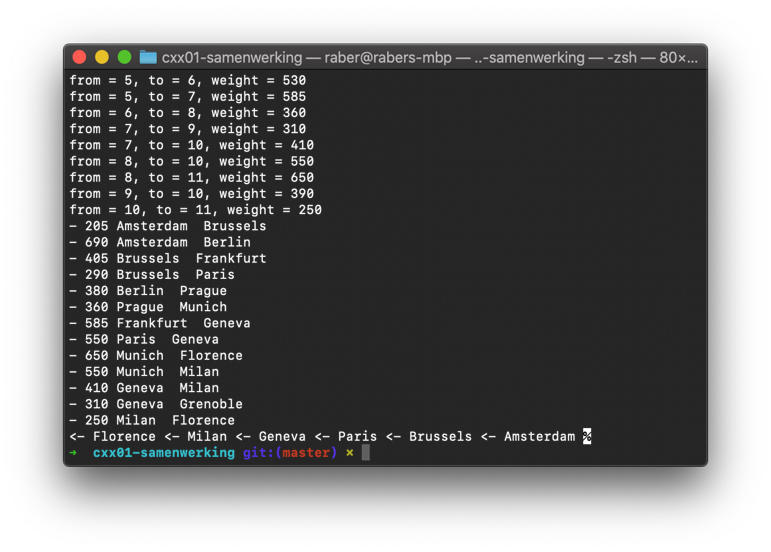
Graph\* graph = loadGraphFromFile("json/citiesShortestPath.json");

Hier wordt het bestand overgezet naar een graph pointer.

bellmanFordAlg(graph,searchVertexByName(graph,"Amsterdam"), searchVertexByName(graph,"Florence"));

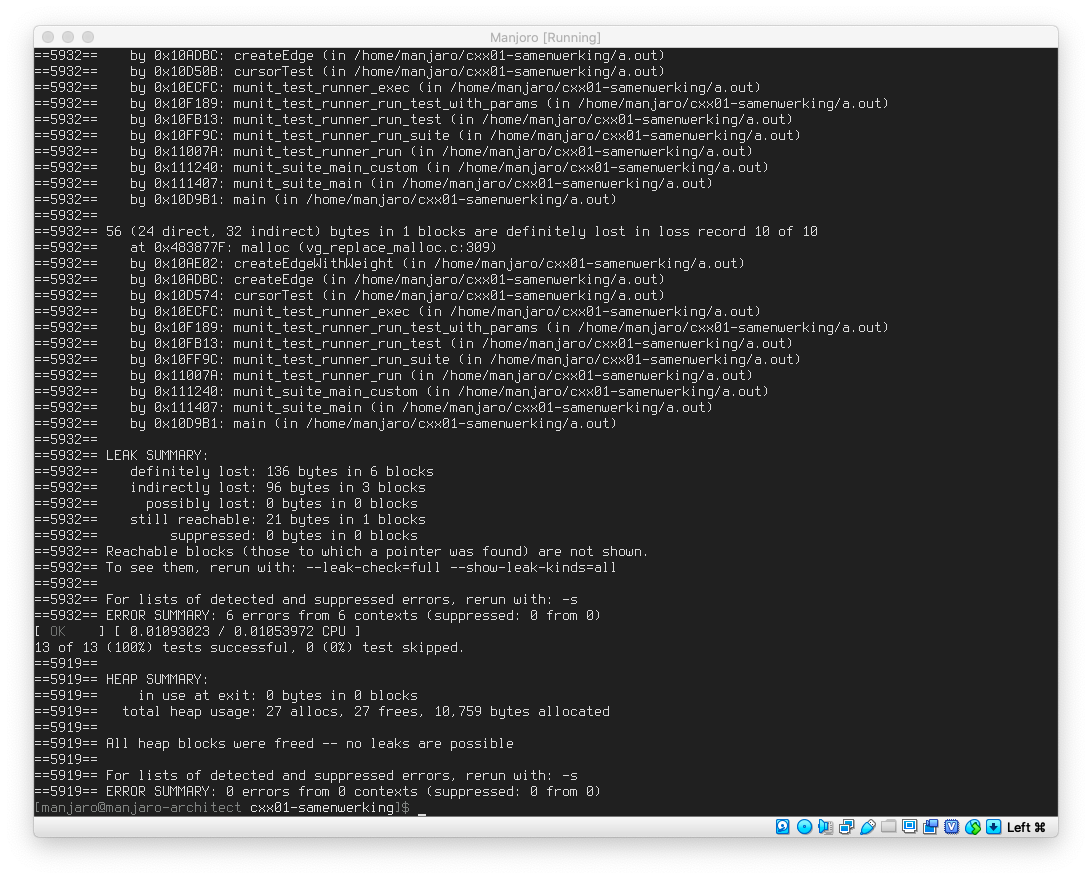
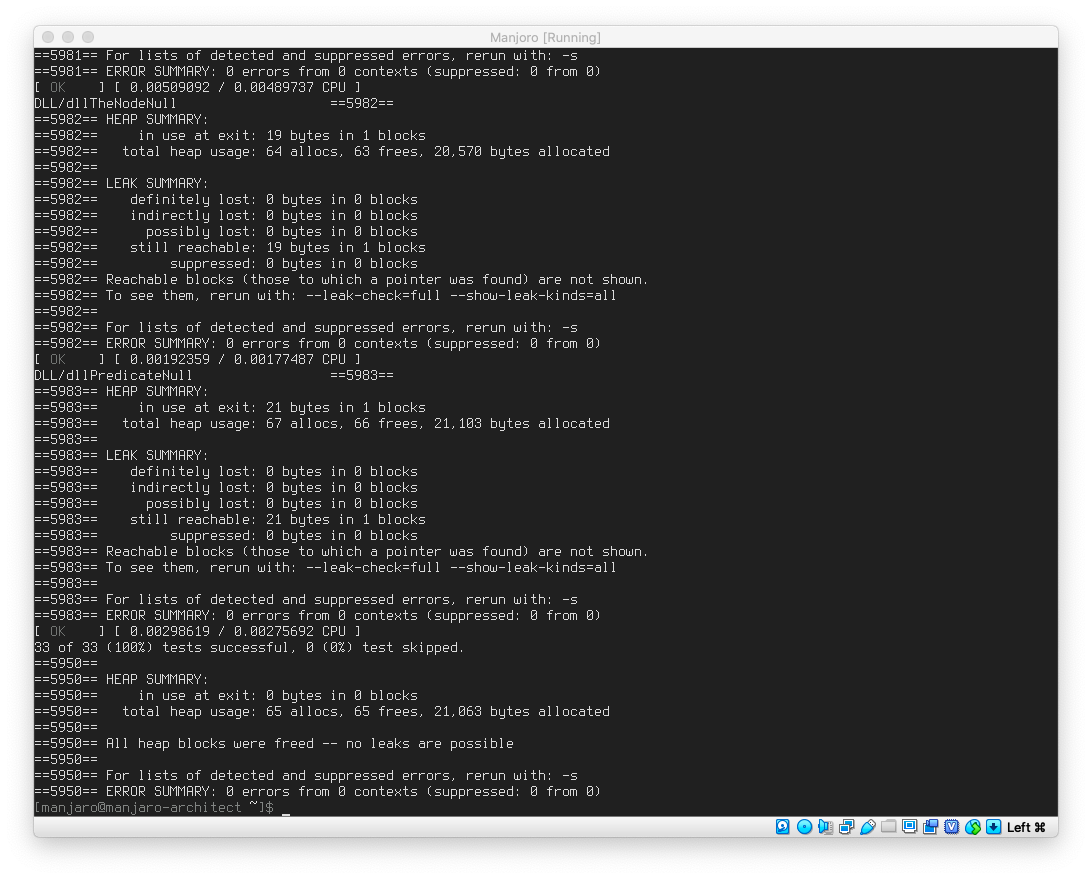
Vervolgens kunnen we met de bovenste stukje code de algoritme aanroepen en de juiste parameters meegeven.

Eerste parameter is de graph waar we uit willen zoeken. De tweede parameter is de startplek, in ons geval is dat Amsterdam. Als laatste wordt de bestemming ingevoerd dat is dus Florence.

Wanneer we dit programma uitvoeren krijgen we terug dat dit het kortste route is. Amsterdam -> Brussels -> Paris -> Milan -> Florence. Dit zien we ook in de afbeelding hier rechts.

# Valgrind en cppcheck

In het vorig document zagen we dat ons graph library niet volledig zonder memory-leaks was. De graph library was gebouwd boven op de doublelinkedlist. De doublelinkedlist alleenstaand was wel vrij van memory-leaks. Hieronder zijn de resultaten van zowel de graph als de dll opnieuw beschreven.



Hierboven zien we screenshot van de valgrind resultaten. De linker is het resultaat van de dllist+ graph en die van de rechter is het valgrind resultaat van alleen de dllist.

We kunnen zien dat de dll alleen geen memory-leaks alles wat wordt gealloceerd wordt ook weer verwijderd. We zien staan dat bij defintely lost, 0 bytes staan.

De graph samen met dll geeft wel aan dat er een memory leak is. We kunnen dat ook weer zien bij defintely lost in de linker afbeelding. Er staat namelijk dat er 136 bytes verloren zijn gegaan in 6 blokken.

## Valgrind applicatie

Hiernaast zien we de valgrind resultaten van de volledige applicatie die bovenop de graph is gebouwd en die weer op zijn weer boven de doublelinkedlist is gebouwd. We zien dat we 8 bytes zijn verloren in 1 block. Daarnaast zijn we 1616 bytes indirect verloren in 75 blocks. We denken dat dit komt omdat er in de onderliggende bouwstenen van ons applicatie al memory-leaks waren en dit dan het vervolg is voor de totale applicatie.

## Testen

Testen is dit keer niet zo uitgebreid gedaan met munittest zoals in de vorige opdrachten. In de afbeelding hiernaast hebben we een foutieve test voor het json naar graph conversie.

## Cppcheck en static analysis

Tijdens het schrijven van de code werd er in de QT IDE gewerkt. QT geeft tijdens het schrijven van de code al automatisch meldingen over de code die zojuist is geschreven zonder het nog uitgevoerd te hebben. De uitvoer van de cppcheck staat in het bestand met de naam cppcheck.txt.