

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
DListNode* 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 "negativeCycleFound" op waar gezet en wordt er gestopt met het lopen.

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 geladen. 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);
```

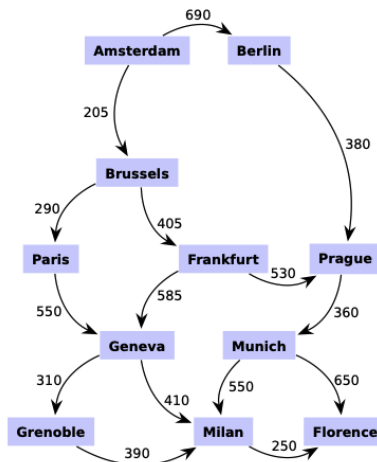
De verbonden edges aan de vertexes worden allemaal uitgeprint.

```
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.



```
{
  "nodes": [
    {"id": 1, "label": "Amsterdam"},
    {"id": 2, "label": "Berlin"},
    {"id": 3, "label": "Brussels"},
    {"id": 4, "label": "Paris"},
    {"id": 5, "label": "Frankfurt"},
    {"id": 6, "label": "Prague"},
    {"id": 7, "label": "Geneva"},
    {"id": 8, "label": "Munich"},
    {"id": 9, "label": "Grenoble"},
    {"id": 10, "label": "Milan"},
    {"id": 11, "label": "Florence"}
  ],
  "edges": [
    {"from": 1, "to": 2, "weight": 690},
    {"from": 1, "to": 3, "weight": 205},
    {"from": 2, "to": 6, "weight": 380},
    {"from": 3, "to": 4, "weight": 290},
    {"from": 3, "to": 5, "weight": 405},
    {"from": 4, "to": 7, "weight": 550},
    {"from": 5, "to": 6, "weight": 530},
    {"from": 5, "to": 7, "weight": 585},
    {"from": 6, "to": 8, "weight": 360},
    {"from": 7, "to": 9, "weight": 310},
    {"from": 7, "to": 10, "weight": 410},
    {"from": 8, "to": 11, "weight": 650},
    {"from": 8, "to": 10, "weight": 550},
    {"from": 9, "to": 10, "weight": 390},
    {"from": 10, "to": 11, "weight": 250}
  ],
  "directed": true,
  "weighted": true
}
```

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.

```
cx01-samenwerking — raber@rabers-mbp — ~/samenwerking — zsh — 80x...
from = 5, to = 6, weight = 530
from = 5, to = 7, weight = 585
from = 6, to = 8, weight = 360
from = 7, to = 9, weight = 310
from = 7, to = 10, weight = 410
from = 8, to = 10, weight = 550
from = 8, to = 11, weight = 650
from = 9, to = 10, weight = 390
from = 10, to = 11, weight = 250
- 205 Amsterdam Brussels
- 690 Amsterdam Berlin
- 405 Brussels Frankfurt
- 290 Brussels Paris
- 380 Berlin Prague
- 360 Prague Munich
- 585 Frankfurt Geneva
- 550 Paris Geneva
- 650 Munich Florence
- 550 Munich Milan
- 410 Geneva Milan
- 310 Geneva Grenoble
- 250 Milan Florence
<- Florence <- Milan <- Geneva <- Paris <- Brussels <- Amsterdam
+ cx01-samenwerking git:(master) *
```

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.

```
Manjaro [Running]
==5932== by 0x109080: createEdge (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x109598: cursorTest (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x109770: munit_test_runner_exec (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097193: munit_test_runner_run_test_with_params (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097613: munit_test_runner_run_test (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097792: munit_test_runner_run_suite (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x110876: munit_test_runner_run (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x111248: munit_suite_main_custom (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x111487: munit_suite_main (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x109761: main (in /home/manjaro/cxx01-samenwerking/a.out)
==5932==
==5932== 56 (24 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 18 of 18
==5932== at 0x483877f: malloc (vg_replace_malloc.c:399)
==5932== by 0x109761: createEdge (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x109080: createEdge (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097792: munit_test_runner_run_test (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097613: munit_test_runner_run_test_with_params (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x1097792: munit_test_runner_run_suite (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x110876: munit_test_runner_run (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x111248: munit_suite_main_custom (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x111487: munit_suite_main (in /home/manjaro/cxx01-samenwerking/a.out)
==5932== by 0x109761: main (in /home/manjaro/cxx01-samenwerking/a.out)
==5932==
==5932== LEAK SUMMARY:
==5932==    definitely lost: 136 bytes in 6 blocks
==5932==    indirectly lost: 96 bytes in 3 blocks
==5932==    possibly lost: 0 bytes in 0 blocks
==5932==    still reachable: 21 bytes in 1 blocks
==5932==    suppressed: 0 bytes in 0 blocks
==5932== Reachable blocks (those to which a pointer was found) are not shown.
==5932== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==5932==
==5932== For lists of detected and suppressed errors, rerun with: -s
==5932== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
[ 0.0 ] [ 0.0189323 / 0.0185392 CPU ]
10 of 10 (100%) tests successful, 0 (0%) test skipped.
==5919==
==5919== HEAP SUMMARY:
==5919==    in use at exit: 0 bytes in 0 blocks
==5919==    total heap usage: 27 allocs, 27 frees, 18,759 bytes allocated
==5919==
==5919== All heap blocks were freed -- no leaks are possible
==5919==
==5919== For lists of detected and suppressed errors, rerun with: -s
==5919== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
manjaro@manjaro-arch:~$ cd01-samenwerking/
```

```
Manjaro [Running]
==5981== For lists of detected and suppressed errors, rerun with: -s
==5981== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ 0.0 ] [ 0.0050992 / 0.0049737 CPU ]
==5982==
==5982== HEAP SUMMARY:
==5982==    in use at exit: 19 bytes in 1 blocks
==5982==    total heap usage: 64 allocs, 63 frees, 20,570 bytes allocated
==5982==
==5982== LEAK SUMMARY:
==5982==    definitely lost: 0 bytes in 0 blocks
==5982==    indirectly lost: 0 bytes in 0 blocks
==5982==    possibly lost: 0 bytes in 0 blocks
==5982==    still reachable: 19 bytes in 1 blocks
==5982==    suppressed: 0 bytes in 0 blocks
==5982== Reachable blocks (those to which a pointer was found) are not shown.
==5982== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==5982==
==5982== For lists of detected and suppressed errors, rerun with: -s
==5982== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ 0.0 ] [ 0.00192359 / 0.00177487 CPU ]
==5983==
==5983== HEAP SUMMARY:
==5983==    in use at exit: 21 bytes in 1 blocks
==5983==    total heap usage: 67 allocs, 66 frees, 21,103 bytes allocated
==5983==
==5983== LEAK SUMMARY:
==5983==    definitely lost: 0 bytes in 0 blocks
==5983==    indirectly lost: 0 bytes in 0 blocks
==5983==    possibly lost: 0 bytes in 0 blocks
==5983==    still reachable: 21 bytes in 1 blocks
==5983==    suppressed: 0 bytes in 0 blocks
==5983== Reachable blocks (those to which a pointer was found) are not shown.
==5983== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==5983==
==5983== For lists of detected and suppressed errors, rerun with: -s
==5983== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ 0.0 ] [ 0.00298619 / 0.00275692 CPU ]
33 of 33 (100%) tests successful, 0 (0%) test skipped.
==5958==
==5958== HEAP SUMMARY:
==5958==    in use at exit: 0 bytes in 0 blocks
==5958==    total heap usage: 65 allocs, 65 frees, 21,863 bytes allocated
==5958==
==5958== All heap blocks were freed -- no leaks are possible
==5958==
==5958== For lists of detected and suppressed errors, rerun with: -s
==5958== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
manjaro@manjaro-arch:~$ cd01-samenwerking/
```

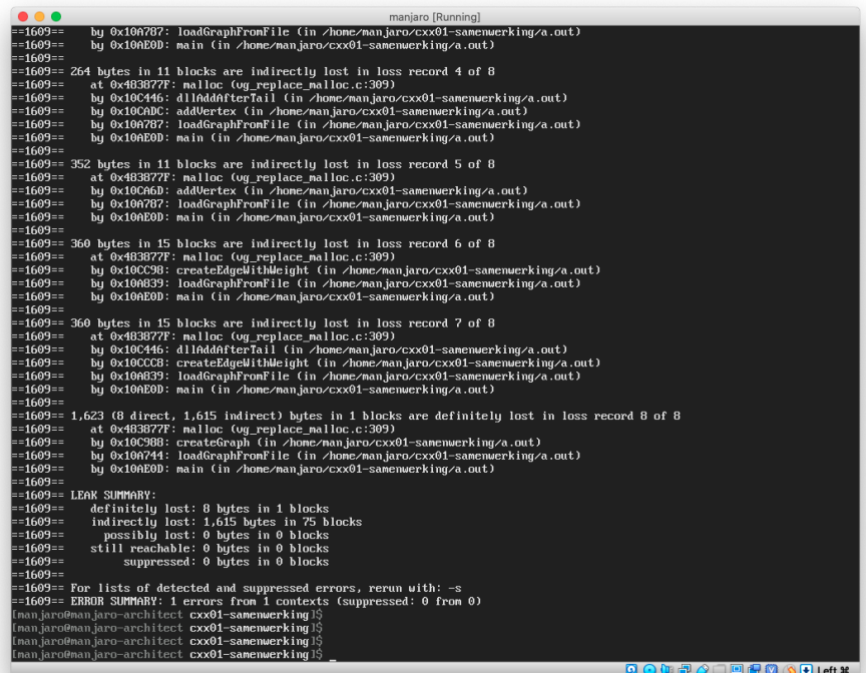
Hierboven zien we screenshot van de valgrind resultaten. De linker is het resultaat van de dllist+ graph en die van de rechter is het valgrind resultaat van alleen de dllist.

We kunnen zien dat de dll alleen geen memory-leaks alles wat wordt gealloceerd wordt ook weer verwijderd. We zien staan dat bij definitely 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 definitely lost in de linker afbeelding. Er staat namelijk dat er 136 bytes verloren zijn gegaan in 6 blokken.

Valgrind applicatie

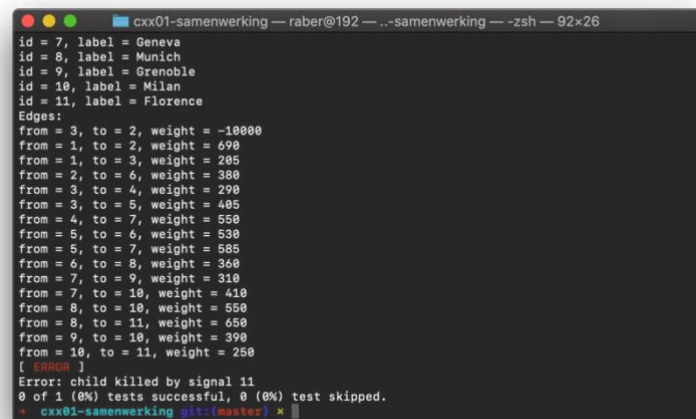
Hiernaast zien we de valgrind resultaten van de volledige applicatie die bovenop de graph is gebouwd en die weer op zijn weer boven de doublelinkedlist is gebouwd. We zien dat we 8 bytes zijn verloren in 1 block. Daarnaast zijn we 1616 bytes indirect verloren in 75 blocks. We denken dat dit komt omdat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



```
manjaro [Running]
==1609== by 0x10A787: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== 264 bytes in 11 blocks are indirectly lost in loss record 4 of 8
==1609== at 0x483877F: malloc (vg_replace_malloc.c:309)
==1609== by 0x10C446: dllAddAfterTail (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10CADC: addVertex (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10A787: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== 352 bytes in 11 blocks are indirectly lost in loss record 5 of 8
==1609== at 0x483877F: malloc (vg_replace_malloc.c:309)
==1609== by 0x10C46D: addVertex (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10A787: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== 360 bytes in 15 blocks are indirectly lost in loss record 6 of 8
==1609== at 0x483877F: malloc (vg_replace_malloc.c:309)
==1609== by 0x10CC98: createEdgeWithWeight (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10A839: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== 360 bytes in 15 blocks are indirectly lost in loss record 7 of 8
==1609== at 0x483877F: malloc (vg_replace_malloc.c:309)
==1609== by 0x10C446: dllAddAfterTail (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10CCB8: createEdgeWithWeight (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10A839: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== 1,623 (8 direct, 1,615 indirect) bytes in 1 blocks are definitely lost in loss record 8 of 8
==1609== at 0x483877F: malloc (vg_replace_malloc.c:309)
==1609== by 0x10C988: createGraph (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10A744: loadGraphFromFile (in /home/manjaro/cxx01-samenwerking/a.out)
==1609== by 0x10AE0D: main (in /home/manjaro/cxx01-samenwerking/a.out)
==1609==
==1609== LEAK SUMMARY:
==1609==    definitely lost: 0 bytes in 1 blocks
==1609==    indirectly lost: 1,615 bytes in 75 blocks
==1609==    possibly lost: 0 bytes in 0 blocks
==1609==    still reachable: 0 bytes in 0 blocks
==1609==    suppressed: 0 bytes in 0 blocks
==1609==
==1609== For lists of detected and suppressed errors, rerun with: -s
==1609== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
manjaro@manjaro-architect: cxx01-samenwerking$
manjaro@manjaro-architect: cxx01-samenwerking$
manjaro@manjaro-architect: cxx01-samenwerking$
manjaro@manjaro-architect: cxx01-samenwerking$
```

Testen

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



```
id = 7, label = Geneva
id = 8, label = Munich
id = 9, label = Grenoble
id = 10, label = Milan
id = 11, label = Florence
Edges:
from = 3, to = 2, weight = -10000
from = 1, to = 2, weight = 490
from = 1, to = 3, weight = 285
from = 2, to = 6, weight = 380
from = 3, to = 4, weight = 290
from = 3, to = 5, weight = 485
from = 4, to = 7, weight = 550
from = 5, to = 6, weight = 530
from = 5, to = 7, weight = 585
from = 6, to = 8, weight = 360
from = 7, to = 9, weight = 310
from = 7, to = 10, weight = 410
from = 8, to = 10, weight = 550
from = 8, to = 11, weight = 650
from = 9, to = 10, weight = 390
from = 10, to = 11, weight = 250
[ ERROR ]
Error: child killed by signal 11
0 of 1 (0%) tests successful, 0 (0%) test skipped.
manjaro@manjaro-architect: cxx01-samenwerking$
```

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.