FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR
Vakgroep Informatietechnologie (INTEC)

Software Ontwikkeling 1

Academiejaar 2012–2013

SOI@intec.UGent.be

Practicum 4

Fietsroute-graaf d.m.v. ijle matrix

1. Routeplanners en grafen

Via websites zoals *fietsnet.be* kun je leuke fietsroutes maken. Deze fietsroutes zijn gebaseerd op de fietspunten die voorzien werden door Toerisme Vlaanderen. Je fietst je route dan door van het ene fietspunt naar het andere te fietsen. De tijd van de te grote fietskaarten is voorbij. Nu heb je voldoende aan je lijstje fietspunten die je wilt doen.

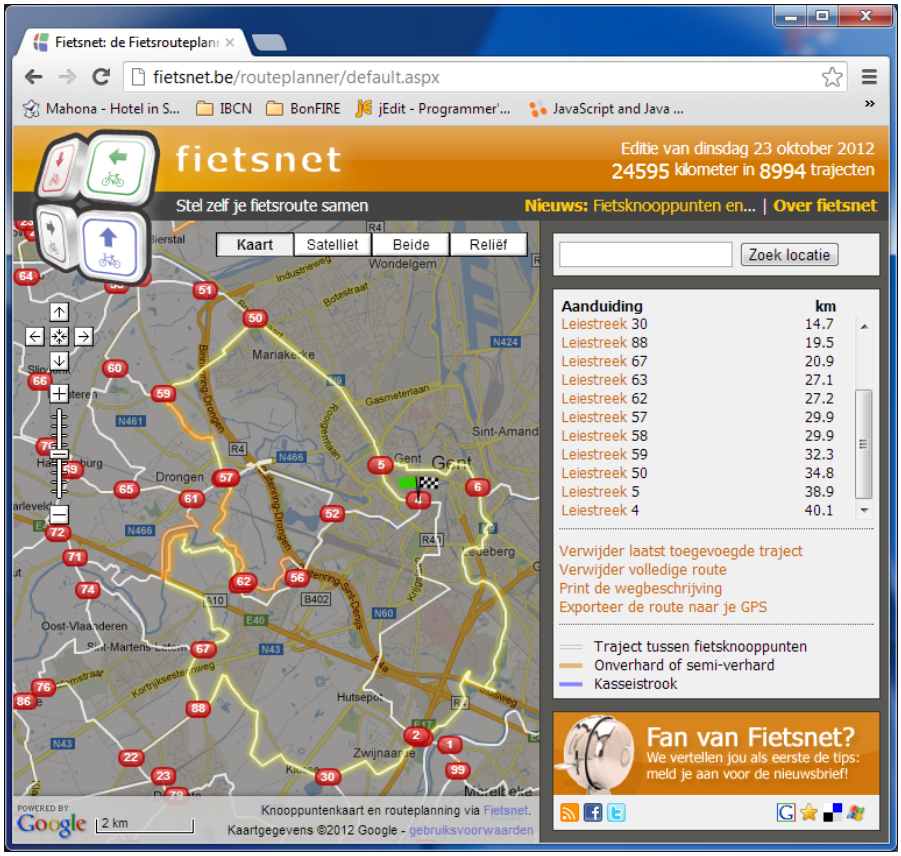
Aangezien we in dit practicum enkel geïnteresseerd zijn in welke fietspunten genomen worden, kan je heel eenvoudig de fietsroute als graaf voorstellen. De fietspunten zijn dan ook de knooppunten van de graaf en de routes tussen de punten zullen dan verbindingen in je graaf zijn. In de cursus wordt uitgelegd hoe grafen kunnen voorgesteld worden (zie Appendix C.5, p.206 van de cursus).

Wij zullen in dit practicum gebruik maken van een uitbreiding op de matrix-voorstelling. Omdat we ook willen bijhouden langs welke route een bepaald knooppunt wordt benaderd, zullen we werken met een **gerichte graaf**. In de cursus wordt gesproken dat de waarde die bijgehouden wordt in de matrix 0 of 1 zal zijn. In deze opgave wijken we hier wat van af: in onze voorstelling houden we bij hoe vaak een bepaalde route wordt genomen en het is dit aantal dat we in de matrix zullen opslaan. Aangezien Vlaanderen meer dan 4000 knooppunten heeft ingepland, is het niet aangewezen om alle 4000 knooppunten in onze matrix-voorstelling op te nemen. We zullen daarom een **ijle matrix** gebruiken om onze graaf voor te stellen.

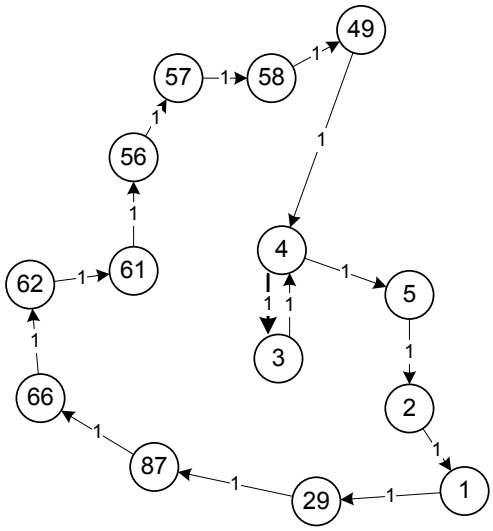
IJle matrices (Engels: *sparse matrices*) zijn matrices waar de meerderheid van de elementen leeg of nul zijn. Het is dan vaak minder efficiënt een array-datastructuur te gebruiken om een ijle matrix voor te stellen aangezien dan veel geheugen verloren gaat voor het bijhouden van irrelevante elementen. Er bestaan efficiëntere structuren voor dergelijke matrices. Hierbij moet dan een afweging gemaakt worden tussen snelle toegang tot de individuele elementen van de matrix, en geheugengebruik.

Om onze ijle matrix te implementeren zullen we de hieronder beschreven datastructuur gebruiken. Eenvoudig gezegd, wordt een graaf (eigenlijk de ijle matrix) voorgesteld als een speciale collectie van graafelementen (`GraphElement`). Elke `GraphElement` heeft twee wijzers: een wijzer naar het volgende element in dezelfde matrixkolom (`nextRow`) en een wijzer de volgende matrix-kolom, in het bijzonder naar het eerste element in de volgende kolom (`nextColumn`). Het `GraphElement` bevat ook de rij en kolom die het inneemt in de matrix (respectievelijk `from` en `to`) en de waarde (`count`), die aanduidt hoe vaak een route genomen wordt.

Een voorbeeldje, de volgende route van *fietsnet.be*



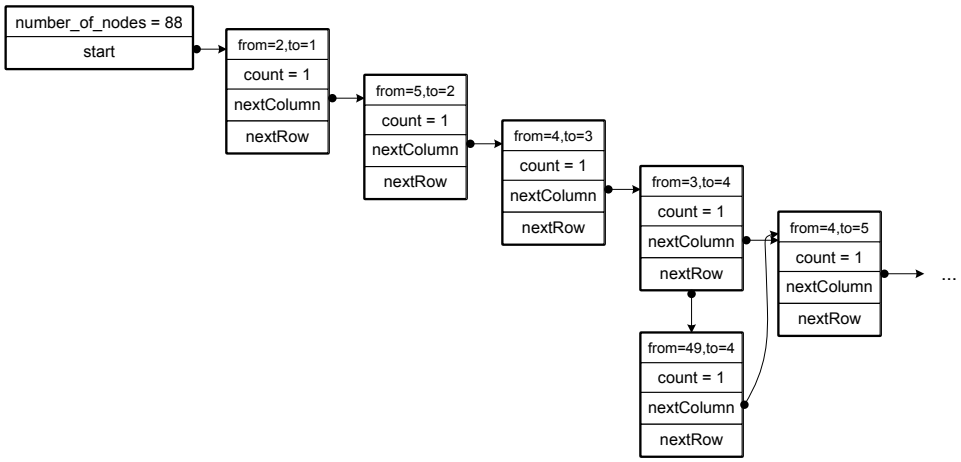
wordt schematisch voorgesteld als de onderstaande graaf. Let op de namen van de fietspunten. De fietspunten bij *fietsnet.be* worden genummerd vanaf 1, maar in onze graaf zullen de **knooppunten starten vanaf het cijfer 0**. Merk ook op dat de knooppunten van fietsnet.be streekgebonden zijn. Om verwarring te vermijden zullen we ons steeds beperken tot één enkele regio, bijv. de Leiestreek.



Deze graaf wordt als de volgende ijle matrix voorgesteld. De rijnummers zijn de startknooppunten en de kolomnummers zijn de eindknooppunten van de verbindingen. Het aantal rijen en kolommen is gelijk aan het grootste fietspunt, hier is dit 88. Zo zie je dat bijv. de fietsroute van fietspunt 67 naar fietspunt 63 resulteert in een graafverbinding van knoop 66 naar 62. Hiervoor zal dus een matrixelement op rij 66 en kolom 62 met waarde 1. Zoals je kan zien aan de grootte van de matrix en het beperkt aantal matrixelementen, is het duidelijk waarom er geopperd wordt voor een ijle matrix.

	0	1	2	3	4	5	…	29	…	49	…	56	57	58	…	61	62	…	66	…	87
0																					
1								1													
2		*1*																			
3					*1*																
4				*1*		*1*															
5			*1*																		
⋮																					
29																					1
⋮																					
49					1																
⋮																					
56													1								
57														1							
58										1											
⋮																					
61												1									
62																1					
⋮																					
66																	***1***				
⋮																					
87																			1		

Schematisch kan de datastructuur als volgt worden voorgesteld:



2. Opgave

Aangezien C++ een objectgeoriënteerde taal is, wordt in deze opgave gevraagd om een aantal klassen te implementeren, die gebruikt worden voor onze graaf. Twee headerfiles (`Graph.h` en `Iterators.h`) met daarin de te implementeren methodes, een testfile (`main.cpp`) en enkele voorbeeldroutes van Fietsnet.be (`fietsnet{1,2,3}.gpx`), zijn reeds gegeven bij de opgave. Implementeer de gevraagde klassen in een nieuw aan te maken bestanden `Graph.cpp` en `Iterators.cpp`.

`GraphElement`

De klasse `GraphElement` stelt een element voor in de matrix, en bevat de coördinaat `from` en `to` (respectievelijk het start- en eindknooppunt van de verbinding) een positief getal `count` (het aantal keer dat de verbinding wordt genomen), een wijzer (`nextRow`) naar het **volgende element** in **dezelfde kolom** en een wijzer naar het **eerste element** van de **volgende kolom** (`nextColumn`). De definitie van `GraphElement`, met constructor, wordt hieronder weergegeven (je hoeft hier niets meer te implementeren).

```
class GraphElement {

      unsigned int from;
      unsigned int to;
      unsigned int count;
      GraphElement* nextColumn;
      GraphElement* nextRow;

  public:
      GraphElement(unsigned int from, unsigned int to, unsigned int count=0){
            ...
      };
      // de nodige getters en setters
  };
```

Graph

Deze klasse stelt de bovenstaande datastructuur voor grafen (ijle matrix) voor en zal gebruikt worden om zowel de fietsroutes als de frequentiegraaf in voor te stellen. De klasse bevat slechts twee attributen:

- `number_of_nodes` duidt het aantal knopen in de graaf voor. Hier wordt het gebruikt om het aantal fietsknooppunten weer te geven.
- `start` wijst naar het eerste element uit de eerste kolom van de ijle matrixvoorstelling van de graaf.

De klasse-definitie wordt hieronder gegeven:

```
class Graph {
    unsigned int number_of_nodes;
    GraphElement* start;

 public:
    Graph(unsigned int n=0);
    Graph(const Graph& g);
    ~Graph();

     // de nodige getters en setters zijn al geïmplementeerd

    int put_path(unsigned int from, unsigned int to, unsigned int count=1);
    int number_of_transitions(unsigned int node) const;

    const Graph Graph::operator+(const Graph& other) const;
};
```

Implementeer de volgende methodes van de klasse `Graph`:

- `Graph(unsigned int number_of_nodes=0)`

  Basis constructor: initialiseer op een correcte manier de `Graph` klasse. `number_of_nodes` staat hierbij voor het grootste fietspunt dat gebruikt wordt in de graaf. Dit komt ook overeen met het aantal rijen en kolommen van de ijle matrix.

- `Graph(const Graph& g)`

  De Copy constructor: vergeet geen **deep copy** te maken.

- `~Graph()`

  Destructor: geef al het aangemaakte geheugen vrij. Doe geen onnodige deletes, hier zal streng op toegezien worden.

- `int put_path(unsigned int from, unsigned int to, unsigned int count=1)`

  Voegt een nieuwe verbinding toe aan de graaf. `from` is hierbij de startknoop en `to` de eindknoop. Dit zijn al de indexen zoals ze gebruikt worden in de graaf, dus deze

starten vanaf **0**. Aangezien deze verbindingen gericht zijn, is de volgorde van de knooppunten `from` en `to` wel degelijk belangrijk. `count` geeft het aantal keer weer dat deze verbinding genomen is.

Belangrijk, als deze verbinding al bestaat, dient de nieuwe `count` bij de `count` van het bestaande GraphElement object opgeteld te worden. Zo kunnen we bijhouden hoe vaak een bepaalde verbinding genomen wordt.

Vergeet ook niet dat de `GraphElementen` geordend zijn, zowel volgens rijindex (`from`) als volgens kolomindex (`to`). Zo zal `nextRow` steeds verwijzen naar een element met grotere rij-index en zal nextColumn steeds verwijzen naar het element met een grotere kolomindex en met de kleinste rij-index van die kolom. Dit houdt ook in dat de `start` pointer steeds moet wijzen naar het element met de kleinste rij- en kolomindex.

Deze methode geeft `-1` terug als er een fout optreedt, en `0` als het toevoegen gelukt is. Verbindingen met een `count` gelijk aan `0` mogen **niet** toegevoegd worden, aangezien we een ijle matrix voorstelling gebruiken voor onze grafen.

- `int number_of_transitions(unsigned int node) const`

  Geeft het aantal keren terug dat een bepaald knooppunt `node` werd genomen, of -1 bij een fout. Om het eenvoudig te maken, mag je toekomen in een knooppunt en vertrekken uit dat knooppunt als twee individuele gebeurtenissen tellen. In de voorbeeldgraaf hierboven betekent dit dat het knooppunt 4 viermaal wordt aangedaan en de andere knooppunten elk tweemaal.

  Maak voor deze methode enkel gebruik van je `Iterator` klassen. Deze worden hierna uitgelegd.

  Ga voor jezelf na of je weet waarom bij deze methode `const` staat en bij de vorige methoden niet.

- `const Graph operator+(const Graph& other) const`

  Deze methode zal de '+' operatie *overloaden*. We zullen dus grafen kunnen optellen. Eenvoudig gezegd, grafen optellen betekent eigenlijk dat we de frequenties van de verbindingen zullen optellen.

  Maak opnieuw enkel gebruik van een `Iterator`. Let wel, kies de meest geschikte `Iterator`, zodat je de minste `GraphElementen` moet doorlopen.

`MatrixRowIterator en MatrixColumnIterator`

Deze klassen implementeren een `Iterator`. Een `Iterator` is een object dat toelaat om op eenvoudige wijze alle elementen van een collectie te doorlopen, zonder de

implementatie van deze collectie te moeten kennen. De `MatrixRowIterator` zal over de elementen uit een rij van de matrixvoorstelling lopen en de `MatrixColumnIterator` over de elementen uit dezelfde kolom.

Speciaal aan deze Iteratoren is dat, ook al houden we in onze ijle matrix voorstelling enkel de elementen bij die verschillend zijn van 0, de `Iterator` toch doet alsof het hier gaat over een volledig ingevulde matrix. M.a.w. ook al bestaat er in het voorbeeldje geen verbinding van knooppunt 4 naar 56, zal de MatrixRowIterator van rij 4, eens de `Iterator` aan kolom 56 gekomen is, de waarde 0 teruggeven.

Om een voorbeeld te zien van hoe een `Iterator` gebruikt wordt, kun je kijken naar de methode `printGraph` in het `main.cpp` bestand.

Aangezien de functionaliteit van beide `Iteratoren` gelijkaardig is, zullen we enkel de `MatrixRowIterator` volledig bespreken. Let wel, de implementatie van de `MatrixColumnIterator` zal aanzienlijke verschillen hebben.

De klasse-definitie wordt hieronder gegeven:

```
class MatrixRowIterator {
    GraphElement* cursor;
    unsigned int row;                   // bij ColumnIterator is dit column
    unsigned int currentColumn;         // bij ColumnIterator is dit currentRow
    const Graph* g;

public:
    RowIterator(const Graph* g, unsigned int row=0);
    ~RowIterator();

    int next();
    bool hasNext() const;
};
```

Implementeer de volgende methodes van de klasse `MatrixRowIterator` (en gelijkaardig voor de klasse `MatrixColumnIterator`):

- `RowIterator(const Graph* g, unsigned int row=0)`

  Constructor: initialiseert de `Iterator` op `GraphElement* cursor`. Deze `cursor` verwijst steeds naar het volgende effectieve (i.e. non 0) element uit de rij `row` van de matrix die de graaf voorstelt. Als er op geen enkele kolom voor deze rij elementen zijn, zal deze `cursor` uiteraard verwijzen naar `NULL`. `currentColumn` moet op 0 gezet worden en `g` is een verwijzing naar de ijle matrix. Je hebt dit nodig om aan bepaalde attributen te kunnen.

- `~RowIterator()`

  De destructor: als er geheugen werd aangemaakt, dient dit natuurlijk verwijderd te worden.

- `int next()`

  Geeft het aantal keren dat de verbinding tussen de rij van deze `MatrixRowIterator` en `currentColumn` werd genomen. Als deze verbinding niet in onze ijle matrix is opgenomen, wordt 0 teruggegeven. Als de verbinding bestaat, wordt uiteraard de correcte waarde teruggegeven en wordt de cursor verplaats naar het volgende effectieve element (i.e. niet 0 element) in de rij van de ijle matrix. Als er voor deze rij verder geen elementen meer in de ijle matrix zijn opgenomen, wordt de cursor op `NULL` gezet. Vanaf dan wordt opnieuw de waarde 0 teruggegeven tot we buiten de grenzen van onze matrix gaan. Bij een fout wordt er `-1` teruggeven.

  In het voorbeeldje hierboven zou dit dan betekenen dat de `MatrixRowIterator` voor rij 4 achtereenvolgens 0, 0, 0, 1, 0, 1, 0, 0, … wordt teruggegeven. Er wordt op het einde steeds 0 terug gegeven tot `currentColumn` gelijk is aan `nr_of_nodes` van de graaf. Bij een volgende oproep zal dan -1 teruggegeven worden (we gaan buiten de grenzen van de rij van deze ijle matrix).

- `bool hasNext() const`

  Deze methode geeft `true` terug als er nog verdere elementen in de matrixrij zijn (dit mogen ook 0 elementen zijn), anders resulteert deze methode in `false`.

3. Extra methoden in main.cpp

- `void testGraph(const Graph* g, unsigned int row=0)`

  Een uitgebreide test om je implementatie te checken. Voor de evaluatie zullen we iets gelijkaardigs doen.

- `void printGraph(const Graph& g)`

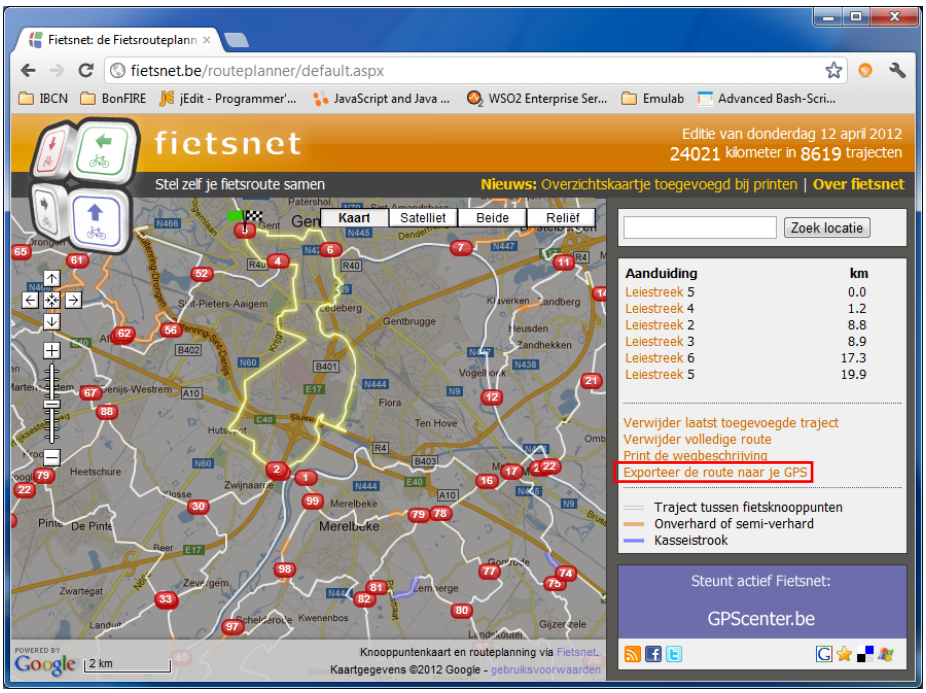
  De methode toont je de volledige matrix voorstelling van je graaf. Ze maakt gebruik van de `MatrixRowIterator`.

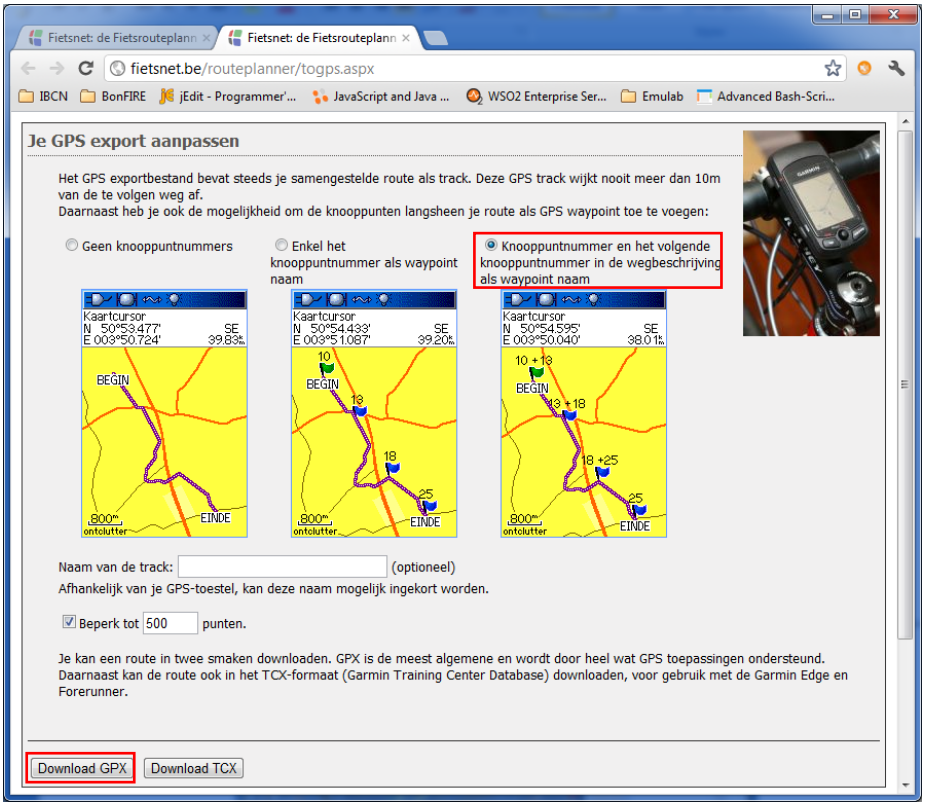
- `void readGraphFromGpx(const char* file, Graph& g)`

  Een hulpmethode die een fietsroute in het GPX formaat omzet tot een graaf. Via deze methode kun je de fietsroutes die je via *fietsnet.be* gemaakt hebt omvormen tot een graaf.

Let op de volgende zaken als je fietsroutes exporteert:

- Selecteer de optie “Knooppuntnummer en het volgende knooppuntnummer in de wegbeschrijving als waypoint naam”
- Gebruik de knop “Download GPX” om je fietsroute in het correcte formaat te verkrijgen.





4. Tips

Belangrijk: Spring steeds zo zuinig mogelijk om met geheugen.

Belangrijk: Zorg ervoor dat je programma geen geheugenlekken vertoont!

Belangrijk: Zorg ervoor dat je compileert. Syntax fouten zullen zwaar afgestraft worden, je hebt immers een IDE die je helpt correcte code te schrijven.

Indien je geheugen alloceert met `new` of `new[]`, vergeet dit dan niet terug vrij te geven (met `delete` of `delete[]` respectievelijk).

Om te controleren of er geen geheugenlekken in je oplossing zitten, maak je best gebruik van de ingebouwde geheugenchecker van MS Visual C++ Express Edition. Zie hiertoe ook het document FAQ_VisualCppEE.pdf op Minerva.

Schrijf duidelijke code, geen cryptische constructies. Vooraleer je aan de implementatie begint, denk eerst goed na over hoe de datastructuur werkt.

5. Indienen

- Er kan samen met een collega tijdens het practicum aan de oplossing begonnen worden, maar het is belangrijk dat nadien de opgave individueel afgewerkt en ingediend wordt.
- Zip je broncode bestanden `Graph.cpp` en `Iterators.cpp` in een bestand naam_voornaam_4.zip. Gebruik standaard ZIP-formaat, geen RAR of 7zip.
- Zet in je bronbestanden bovenaan je naam en de naam van het bestand in commentaar.
- Respecteer de signatuur van de reeds gegeven methoden en de types van de gegeven attributen. Het veranderen van de volgorde van de parameters is **niet** toegestaan. Doe je dit toch, dan zal dat zich reflecteren in je score.
- Stuur je oplossing door middel van de dropbox op Minerva door naar Bruno Volckaert, ten laatste op **zondag 18 november 2012**, **23:59**.

Veel succes!