HOGESCHOOL ROTTERDAM / CMI

Algorithms

INFDEV02-6A

Number of study points:	4 ects
Course owners:	G. Costantini

Module description

Module name:	Algorithms
Module code:	INFDEV02-6A
Study points and hours of effort for full-time students:	This module gives 4 ects, in correspondence with 112 hours: • 3 x 6 hours frontal lecture • 3 x 8 hours self-study for the theory • the rest is self-study for the practicum
Examination:	Written examination and practical assignment (with practical assessment)
Course structure:	Lectures
Prerequisite knowledge:	Object oriented programming
Learning tools:	• Book: *Algorithms* (4rd edition); authors R. Sedgewick, K. Wayne • Lesson slides (pdf): found on N@tschool • Assignments, to be done at home (pdf): found on N@tschool
Connected to competences:	• Realisation • Analysis
Learning objectives:	At the end of the course, the student can: • do performance analysis [PERF] • implement and analyse – basic data structures [$\texttt{DS}^{\texttt{I}}$, $\texttt{DS}^{\texttt{A}}$] – sorting algorithms [$\texttt{SORT}^{\texttt{I}}$, $\texttt{SORT}^{\texttt{A}}$] – recursive data structures [$\texttt{REC}^{\texttt{I}}$, $\texttt{REC}^{\texttt{A}}$] – algorithms on graphs [$\texttt{GRAPH}^{\texttt{I}}$, $\texttt{GRAPH}^{\texttt{A}}$]
Course owners:	G. Costantini
Date:	14 november 2016

1 General description

Designing and manipulating efficient data structures is at the foundation of computer programming. These data structures solve complex problems through well-known, highly difficult techniques that would simply take too long to rediscover from scratch. When faced with certain classes of issues, lack of knowledge of algorithms and data structures might significantly impact a programmer's ability to tackle a given problem.
In this course we are going to explain some of the most popular data structures used in a variety of scenario's encountered in practice when dealing with structuring and traversal of data. We will also cover the principal pathfinding algorithms, which are extensively used for example in GPS devices, routers, etc.

1.1 Relationship with other teaching units

This course builds upon the development courses of the first year.

Knowledge acquired through the algorithms course is also useful for some of the projects. A word of warning though: projects and development courses are largely independent, so some things that a student learns during the development courses are not used in the projects, some things that a student learns during the development courses are indeed used in the projects, but some things done in the projects are learned within the context of the project and not within the development courses.

2 Course program

In the following table you can see the program of the course, divided per lesson. Each lesson is also associated to the corresponding book paragraphs. The last lesson of the course is reserved for a summary in preparation for the exam.

Lesson	Topic	Book paragraphs
1	Arrays; Complexity of algorithms (empirical analysis, O notation), introduction to sorting	1.4
2	Sorting algorithms (insertion sort, mergesort), List, Queue, Stack	2.1, 2.2
3	Hash table, Trees (BST, k-d trees, 2-3 trees)	1.3, 3.4
4	Graphs (undirected; directed; Dijkstra shortest path)	4.1, 4.2, 4.4
5	Dynamic programming; Floyd-Warshall	Not covered by the book, see slides or Cormen
6	Course review and exam simulation	

3 Assessment

The course is tested with two exams: a practical examination and a theoretical examination. The final grade is determined by the practical examination. However, to receive the grade in the practical examination you **must** have a sufficient (i.e. ≥ 5.5) grade in the theoretical examination.

3.1 Theoretical examination

The theoretical examination consists of a **written exam** which covers the topics seen in class. The questions will be both theoretical and about code analysis, such as understanding what a code snippet does, determining its complexity, or finding mistakes in it. The exam lasts two lesson hours (100 minutes). No help is allowed during the exam.
A template of the exam can be found in Attachment 1.

3.2 Practical examination

The practical examination is a **programming assignment**, accompanied by a **practical assessment** (to verify the authorship of code). In the practical assignment, some of the data structures and algorithms seen in class will have to be implemented and applied to a specific case study.
A detailed description of the practical assignment can be found in Attachment 2.

3.3 Retake (herkansing)

If one part of the assessment is not sufficient (theoretical and/or practical examination), then you can repeat that part in the following block:

- In week 3 of the following block you can repeat the written exam.
- The deadline for the delivery of the practical examination is the day before the practical assessment.

Attachment 1: Written exam template

The general shape of a written exam for this course is made up of a series of highly structured open questions.

Question I: complexity

General shape of the question: *Given the following code, what is its (tightest) complexity class using the big-Oh notation?*
Grading: *2 points for correct answer, 0 otherwise.*
Associated learning goals: PERF.

Question II: basic data structures (list, queue, stack, hash tables)

General shape of the question: *Complete the code below so that it produces the desired result on the specified basic data structure.*
Grading: *2 points for correct answer, 1 point for minor mistakes, 0 otherwise.*
Associated learning goals: DS^{A}.

Question III: sorting

General shape of the question: *What is the output of the following algorithm if input is ... ? What is the (tightest) complexity class of the algorithm (using the big-Oh notation)?*
Grading: *2 points for correct complexity and output, 1 point for either correct complexity or output, 0 otherwise.*
Associated learning goals: SORT^{A}.

Question IV: trees

General shape of the question: *Complete the code below so that it produces the desired result (for example, insertion, lookup, traversal, ...) on a tree.*
Grading: *2 points for correct answer, 1 point for minor mistakes, 0 otherwise.*
Associated learning goals: REC^{A}.

Question V: graphs

General shape of the question: *What does the following algorithm produce? What is the (tightest) complexity class of the algorithm (using the big-Oh notation)?*
Grading: *2 points for correct complexity and output, 1 point for either correct complexity or output, 0 otherwise.*
Associated learning goals: GRAPH^{A}.

Attachment 2: Practical examination

General (important) information

- The assignment is individual.
- The assignment must be implemented using C# or F#.
- The only library tools you are allowed to use are: arrays, lists, and math functions. Other data structures or functions on data structures covered by the course must be implemented **by hand**.
- There will be a **practical assessment** on the content of the assignment. During this assessment the teachers reserve the right to ask the students questions concerning the practical assignments and what has been developed during the assessment itself.
- The grade is given by the sum of the grades obtained in each exercise, plus 1.
- **Deadlines and delivery**
  - Each exercise has a (strongly) suggested deadline.
  - You must use *Github* and do frequent commits. The amount/time of the Github commits *could* be checked by the teachers and considered as grading evidence, for example in case of a borderline grade.
  - By the *end of week 7*, you must upload your project on *N@tschool* (through the "inleveropdracht").

Introduction to the framework

The exercises will be based on the simulation of a city, containing houses and special buildings (represented through ancient civilization temples), all connected by streets. The student must implement algorithms to answer some queries on the simulated city. To set up the framework, follow these instructions:

- Download and install Visual Studio 2015 Community (`https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx`); choose "F# language"during *custom setup*;
- Download the project framework from N@tschool (or Github) and open the .sln file; if it is not already set, right click on `EntryPoint` project (in the solution explorer) and select "Set as startup project";
- The functions you have to program are contained in the file `Program.cs`; for now there are stub versions of the function implementations that you have to replace with yours;
- Compile the project in debug mode and run it with Ctrl+F5;
- A window opens asking you which assignment you want to execute: choose a number between 1 and 4 [1] (or q if you want to close it) and press Ok;
- The simulation will start. You can move the visual with WASD and zoom with Z (zoom in) and X (zoom out).
- Exit with ESC.

[1] In the application, assignment 3 corresponds to exercise 3.1; assignment 4 corresponds to exercise 3.2.

Exercise 1 - Sorting

Suggested deadline: End week 3.
Points: 3.

Goal

Sort all special buildings by Euclidean distance from a specified house. The Euclidean distance formula is:

$$d(house, building) = \sqrt{(x_{house} - x_{building})^2 + (y_{house} - y_{building})^2}$$

This means that the connection through roads is not relevant in this exercise (everyone has his private helicopter to move around the city).

Function signature

```
private static IEnumerable<Vector2> SortSpecialBuildingsByDistance(Vector2 house,
    IEnumerable<Vector2> specialBuildings)
```

This function takes as input a house position (`Vector2 house`) and a list of building positions (`IEnumerable<Vector2> specialBuildings`) and returns a sorted list of building positions according to their distance from the house position (`IEnumerable<Vector2>`). Use the **merge sort** as sorting algorithm. Any implementation not using this technique will not be accepted and evaluated.

Result

As you can see from Figure 3.3.1, the selected house is highlighted and there is a number above each special building indicating its position in the sorted list of buildings.

Figuur 3.3.1: Exercise 1 result

Exercise 2 - Trees

Suggested deadline: End week 5.
Points: 3.

Goal

Find all the special buildings within a specified distance from each house. Create a **k-d tree** to organize the special buildings positions in advance (like explained in class). Then look up the tree for each of the requested houses (with the associated distance).

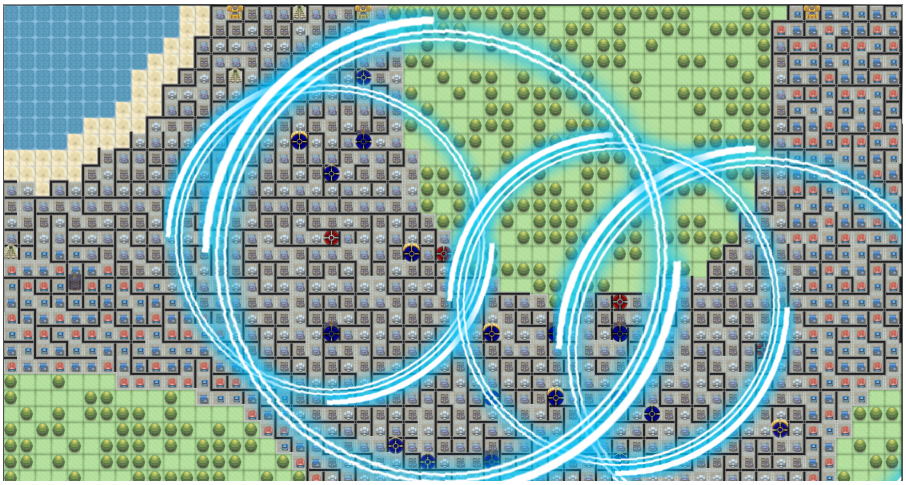
Function signature

```
private static IEnumerable<IEnumerable<Vector2>>
    FindSpecialBuildingsWithinDistanceFromHouse(IEnumerable<Vector2> specialBuildings,
    IEnumerable<Tuple<Vector2, float>> houseAndDistances)
```

This function takes as input a list of special building positions (`IEnumerable<Vector2> specialBuildings`), a list of pairs made of a house position and the maximum distance for a special building to be selected (`IEnumerable<Tuple<Vector2, float>> houseAndDistances`), and returns a list of lists of positions (`IEnumerable<IEnumerable<Vector2>>`) of selected special buildings (one list for each house).

Result

As you can see from Figure 3.3.2, each selected house is surrounded by a circle that should contain all the special buildings within the distance associated to such house. The buildings you return are highlighted in blue, the houses in red.



Figuur 3.3.2: Exercise 2 result

Exercise 3 - Graphs

Suggested deadline: End week 7.
Points: 3.

Goal

Find the shortest path(s) from a specified house to other special building(s). The shortest path is made of the road sections to use in order to drive from the house to the special building.
Remark: For this assignment only, you can choose between two different implementations (Dijkstra or Floyd Warshall).
Hint: In both assignments, you have to build the adjacency matrix using the starting point and endpoint of the road sections. The weight of the edge is the distance between the two points.

Option 1: Dijkstra

In this assignment we want to compute the minimum path between one house and one special building.

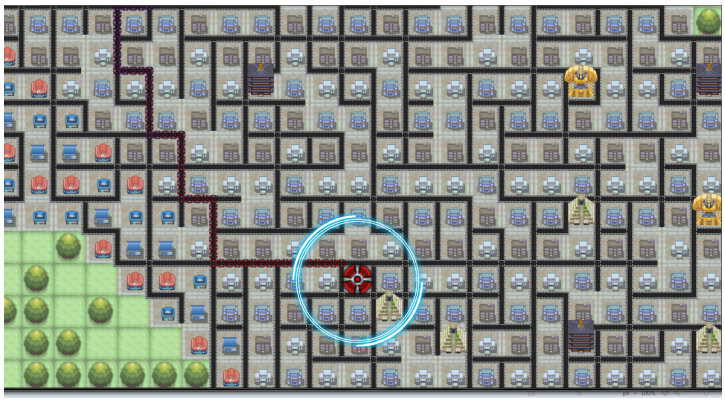
Function signature

```
private static IEnumerable<Tuple<Vector2, Vector2>> FindRoute(Vector2 startingBuilding,
    Vector2 destinationBuilding, IEnumerable<Tuple<Vector2, Vector2>> roads)
```

This function takes as input the position of the house to start from (`Vector2 startingBuilding`), the position of the destination (`Vector2 destinationBuilding`), and a list of road sections, each represented as a pair of starting point and endpoint (`IEnumerable<Tuple<Vector2, Vector2>> roads`), and returns a list of road sections (`IEnumerable<Tuple<Vector2, Vector2>>`) forming the shortest path between the house and the destination.

Result

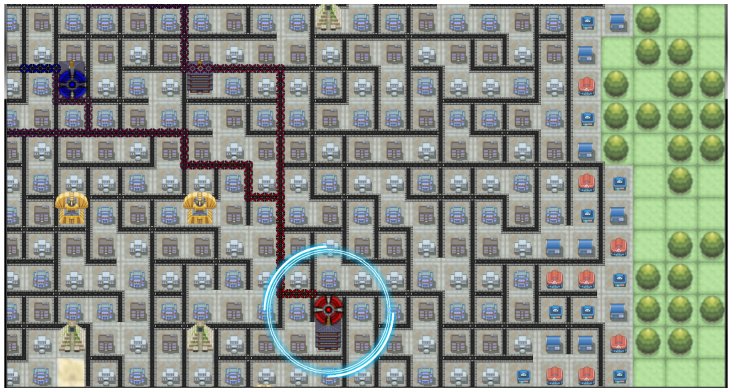
As you can see from Figure 3.3.3, the house is surrounded by a circle and highlighted in red. The path to the destination is highlighted with coloured dots.



Figuur 3.3.3: Exercise 3.1 result

Option 2: Floyd Warshall

In this assignment we want to compute the minimum paths between one house and a list of special buildings.



Figuur 3.3.4: Exercise 3.2 result

Function signature

```
private static IEnumerable<IEnumerable<Tuple<Vector2, Vector2>>> FindRoutesToAll(Vector2
    startingBuilding, IEnumerable<Vector2> destinationBuildings, IEnumerable<Tuple<
    Vector2, Vector2>> roads)
```

This function takes as input the position of the house to start from (`Vector2 startingBuilding`), the positions of all the destinations (`IEnumerable<Vector2> destinationBuildings`), and a list of road sections, each represented as a pair of starting point and endpoint (`IEnumerable<Tuple<Vector2, Vector2>> roads`), and returns a list of shortest paths (`IEnumerable<IEnumerable<Tuple<Vector2, Vector2>>>`). Each shortest path is referred to one specific destination and is made of a list of road sections. The function must precompute the all-pairs shortest path with Floyd Warshall algorithm and then extract only the paths requested by the function from the distance and predecessor matrices.

Result

As you can see from Figure 3.3.4, the house is surrounded by a circle and highlighted in red. The destinations are highlighted in blue. The paths from the house to all destinations are highlighted with coloured dots.