HOGESCHOOL ROTTERDAM / CMI

Algorithms

INFDEV03-6A

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

Module description

Module name:	Algorithms
Module code:	INFDEV03-6A
Study points and hours of effort for full-time students:	This module gives 4 ects, in correspondence with 112 hours: • 3 x 8 hours frontal lecture • the rest is self-study
Examination:	Written exam and practical assessment
Course structure:	Lectures
Prerequisite knowledge:	Object oriented programming
Learning tools:	• Book: *Introduction to Algorithms*; authors C.E. Leiserson, C. Stein, R. Rivest, and T.H. Cormen • 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: • Analyse the performance of an algorithm [PERF] • Infer the behavior of algorithms involving basic data structures and analyse their performance [$\texttt{DS}^{\texttt{A}}$] • Implement sorting algorithms [$\texttt{SORT}^{\texttt{I}}$] • Infer the behavior of sorting algorithms and analyse their performance [$\texttt{SORT}^{\texttt{A}}$] • Implement recursive data structures [$\texttt{REC}^{\texttt{I}}$] • Infer the behavior of algorithms manipulating recursive data structures and analyse their performance [$\texttt{REC}^{\texttt{A}}$] • Implement graphs representation and algorithms [$\texttt{GRAPH}^{\texttt{I}}$] • Infer the behavior of algorithms manipulating graphs and analyse their performance [$\texttt{GRAPH}^{\texttt{A}}$]
Course owners:	G. Costantini
Date:	10 november 2017

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 for every application. 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 efficiently and effectively.
In this course we are going to explain some popular algorithms and data structures used in a variety of scenario's encountered in practice when dealing with structuring and traversal of data.

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. Moreover, the whole Informatica program is not based on the "workshop" philosophy but rather on building solid foundations preparing students for the future evolutions of computer science (see for instance the module description of Development 1).

2 Course program

In the following table you can see the program of the course, divided in lesson units. Each lesson unit is also associated with the corresponding book paragraphs. The last lesson unit of the course is reserved for a summary in preparation for the exam.
Note: Lesson units are intented as collections of topics and do not necessarily correspond to study weeks (for example, a lesson unit could span two study weeks).

Lesson unit	**Topics**	**Book chapters and paragraphs**
1	Introduction to algorithms Arrays Complexity of algorithms (empirical analysis, O notation)	3
2	Sorting algorithms - Insertion sort - Merge sort	2.1, 2.3
3	List Queue Stack Hash table	10.1, 10.2, 11.1 until 11.4
4	Trees - BST - k-d trees - 2-3 trees	12.1 until 12.3, 18
5	Graphs - undirected - directed - Dijkstra's shortest path	22.1 until 22.3, 24.3
6	Dynamic programming Floyd-Warshall	15.3, 25.2
7	Course review and exam simulation	

3 Assessment

The course is tested with two exams: a practical assessment and a theoretical examination. The final grade is determined by the practical assessment. However, to be admitted to the practical assessment, you **must** have a sufficient (i.e. ≥ 5.5) grade in the theoretical examination. If your grade in the theoretical examination is not sufficient, then we will register a 0 (zero) in Osiris as grade of the practical assessment.
The correspondence between learning goals and assessment parts is shown in the following table. For more details, see the examination matrix in Attachment 1.

Learning goal	Theory	Practice
PERF	V	
DS^A	V	
$SORT^I$		V
$SORT^A$	V	
REC^I		V
REC^A	V	
$GRAPH^I$		V
$GRAPH^A$	V	

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.

The exam consists of:

- 6 questions about complexity of algorithms ("What is the complexity of the following code/algorithm?")
- 4 questions about inference of the behaviour of a given algorithm (for example, "What does this algorithms do?" or "Does this algorithm find the minimum element?", etc.)

3.2 Practical assessment

The practical examination is a **practical assessment** during which the student is asked to implement (completely or in part) algorithms seen during the course or strictly connected to course topics. The practical assessment consists of:

- 5 assignments about filling in code of given partial algorithms

The programming language used for the practical assessment will be one of the languages covered by a previous Development course. More details will be given during the lessons.

As preparation for the assessment, the students are strongly suggested to complete a **formative programming assignment** (see Attachment 2). In this programming assignment, some of the data structures and algorithms seen in class will have to be implemented and applied to a specific case study.

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 educational period:

- In week 10 of the following OP you can repeat the written exam.
- In week 10 of the following OP you can repeat the practical assessment (if the written exam has been already passed).

Attachment 1: Examination matrix

- Module: INFDEV03-6A (Algorithms)
- Module responsible: G. Costantini
- Study points: 4
- Exam form: Written exam [W.E.] (prerequisite) + Practical Assessment [P.A.] (100% final grade)
- Exam date: OP 2

Learning goals	Knowledge	Insight	Apply	Analyse [W.E.]	Synthesize [P.A.]	Evaluate	Total points [W.E]	[P.A.]
1 Analyse the performance of an algorithm [PERF]				20%			20%	
2 Infer the behavior of algorithms involving basic data structures and analyse their performance [DS^{A}]				20%			20%	
3 Implement sorting algorithms [SORT^{I}]					30%			30%
4 Infer the behavior of sorting algorithms and analyse their performance [SORT^{A}]				20%			20%	
5 Implement recursive data structures [REC^{I}]					30%			30%
6 Infer the behavior of algorithms manipulating recursive data structures and analyse their performance [REC^{A}]				20%			20%	
7 Implement graphs representation and algorithms [GRAPH^{I}]					40%			40%
8 Infer the behavior of algorithms manipulating graphs and analyse their performance [GRAPH^{A}]				20%			20%	
Total points:				100% = prereq	100%		100%	100%
Cesure:	Cesure of written exam: 5.5, which is prerequisite to be admitted to the practical assessment. Cesure is there also 5.5. The student receives a grade only for the assessment.							
Retake:	For both written exam and practical assessment there is a retake (within the scholastic year).							

Attachment 2: Formative project for practicum exam

General (important) information

- 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**.

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 the latest version of Visual Studio 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

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

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).

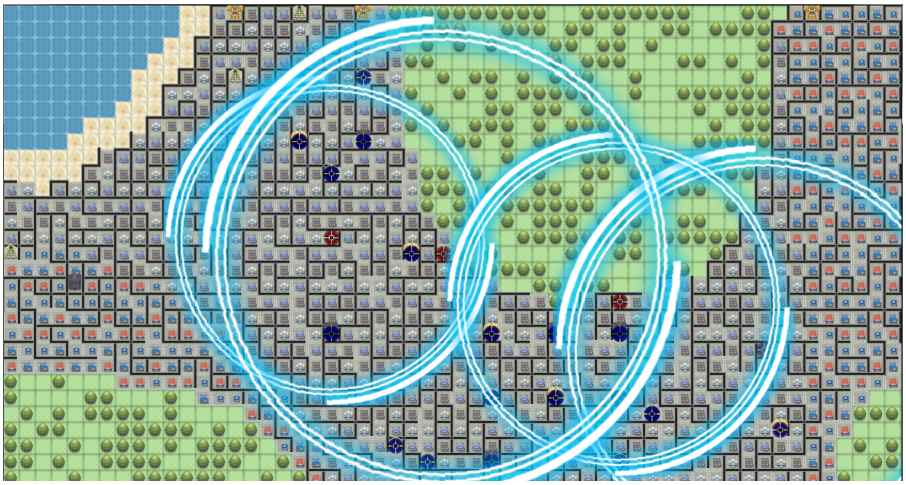
Function signature

```
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

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: This assignment can be solved in two ways, that is using 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.

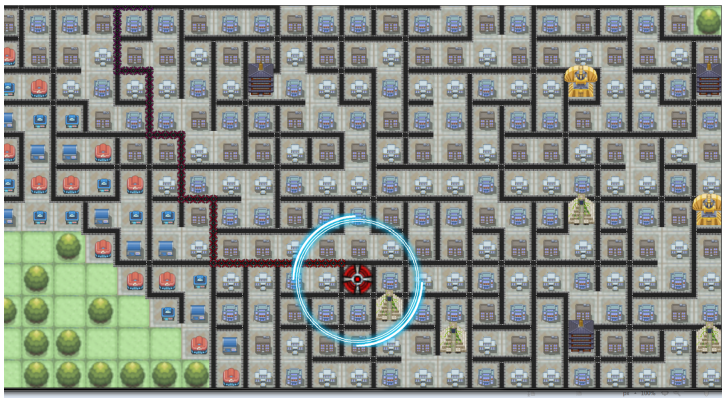
Function signature

```
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.



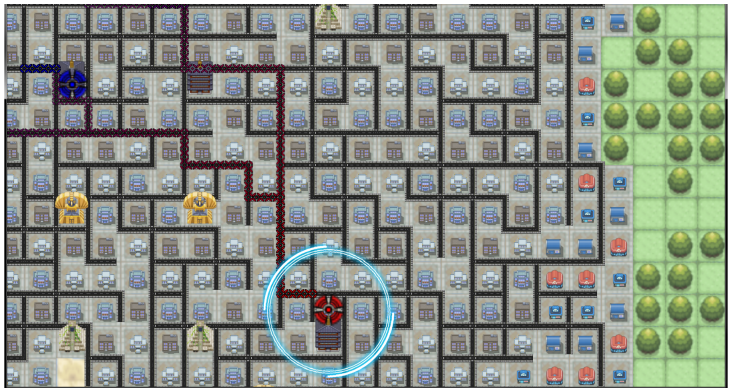
Figuur 3.3.3: Exercise 3.1 result

Option 2: Floyd Warshall

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

Function signature

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



Figuur 3.3.4: Exercise 3.2 result

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.