# Aueb Navigation System: Object Oriented Programming Implementation in Python

Leonidas Ntrekos s6210058
Email: s6210058@aueb.gr
GitHub: LNtrekos

January 2026

**Abstract**

This project is a toy-example navigation system for the Athens University of Economics and Business (AUEB), written in Python using object-oriented programming methods. The university environment is modeled as a weighted graph, where classrooms are represented as nodes and feasible connections between them as edges, whose weights depend on spatial distance and floor-related constraints.

An interactive menu is implemented to allow users to load map data, construct the underlying graph, visualize the university layout, and compute shortest paths between selected classrooms. Shortest-path computation is performed using **Dijkstra**'s algorithm, adapted to the custom graph representation.

## 1 Overview

The whole project consists of 3 different scripts or Python modules (which can be found in the aueb_pathfinding file), and additionally, its main.py file, which is the final implementation.

1. **classes.py** is the core of the whole project as it contains the `Classroom` and `University` classes. These two are the main components upon which we perform each procedure: initiate a graph, calculate the shortest distance between two given nodes (in this case, classrooms), and visualize the whole university.

2. **utils.py** contains the basic logic behind the program. Most important functions here are the **dijkstra** function, which implements the (what a surprise) Dijkstra algorithm, and the **distance** function, which defines the space of the university environment. Last but not least is the clean_values function, which filters out special symbols contained in the raw data provided.

3. **menu.py** carries out the menu "interface". Along with a few utility functions to guarantee a smooth user journey, offers all the procedures needed for user interaction. It basically contains all the "middleman" functions that connect the user to the University class's utilities through a safe space.

The whole program, along with each procedure, can be executed through the main.py. This is the file that assembles all pieces from the different modules into an easily read and straightforward single script.

## 2 Classes

As it has been repeated many times, this is an object-oriented program that is expressed through the two main classes:

- **Classroom Class**

Represents a single classroom (node) within the university. Each classroom is defined by its **name**, its coordinates **(x, y)**, and the **floor** on which it is located. These attributes allow the classroom to be treated as a node in a graph-based representation of the university.

Basic validation is performed during initialization to ensure that classroom names are non-empty strings and that coordinates and floor are integers. The class also implements the **___str___** method, which returns its name (for example, "A21"), while the **___repr___** method offers a more detailed representation intended for debugging and development purposes, which returns the classroom's name and coordinates.

Furthermore, the **___eq___** and **___hash___** methods are implemented so that classrooms can be compared and safely used as keys in dictionaries. Two classrooms are considered equal if they share the same name. In the data that we have, we do not have duplicates. This exception simplifies things and may be considered unnecessary to implement the hash method, but it is something that was discussed in class and thought of implementing.

- **University Class**

As the name suggests, this class represents the university environment as a whole. It basically models the space between the classrooms (previous class) as an undirected weighted graph, where classrooms correspond to nodes and the "walkable" paths between them correspond to edges. Its main parameters are a list of **classroom nodes**, a dictionary of **edges**, a maximum allowable walking distance **max_distance**, and a **floor_weight** parameter that penalizes movement between different floors. The last two are specified by the user during the initialization of the graph (further discussed later) and serve as adjustments to different user preferences.

The class provides the methods to **add classrooms** to the graph and to **create edges** between them. Edge creation relies on a weighted distance function that combines Euclidean distance with an additional penalty when classrooms are located on different floors (further discussed in the section 3). Connections exceeding the maximum distance threshold are discarded.

This class additionally provides a method to retrieve the **neighbor** nodes, which is vital for the Dijkstra algorithm to operate. Lastly, a custom **___str___** method is implemented to display basic information of the graph that is created.

## 3 utils

The `utils` file contains basic helper functions that implement core logical components of the whole program.

- **clean_values**

As the name implies, this function removes special characters from each value passed to it by using a regular expression. It is necessary for the initial raw data loading process. After filtering out tries to convert to an integer, if that fails, it converts the value to a character, implying that it has to be the name of the class. This works because in the raw data available, the first column is the classroom name, while the other three columns are the coordinates and the floor, which are integers.

- **distance**

As the name suggests, the function **distance** calculates the weighted distance between two classrooms. This distance has two parts: the Euclidean distance based on classroom coordinates (x, y), and an extra penalty if the classrooms are on different floors. The penalty starts as the square of the difference between the floors, then it is multiplied by the **floor_weight** parameter. This lets you adjust how much changing floors costs. The function is used when building the graph to decide if two classrooms can be directly connected and to set the edge weights.

- **dijkstra**

This function implements Dijkstra's algorithm for computing the shortest path between two classrooms in the university graph. If the target classroom is unreachable, the function returns an empty path and an infinite distance; otherwise, it reconstructs and returns the shortest path along with its total cost.

# 4 menu

This file contains functions that connect user input with the core logic of the indoor navigation system. It provides the main menu interface, input-driven graph construction, university visualization, and shortest-path calculation.

The main idea behind the menu system is to act as the connecting layer between the `Classroom` and `University` classes and the utility functions described earlier. In this way, all functionalities of the navigation system are made accessible through a single, user-friendly command-line interface.

More specifically, the menu allows the user to:

1. **Load and validate map data**

The user can load classroom data from a text file using the **load_map** function. Before any further action is permitted, initialization checks (**map_init_check** and **uni_init_check**) ensure that the required steps are followed in the correct order, preventing invalid program states.

- **Create and configure the university graph**

Through interactive input prompts, the user specifies the graph parameters that are the maximum allowed distance between two classrooms to be considered neighbors and the floor-weight penalty (that is, multiplied by the square differences of the floor difference between two classrooms). The **create_graph** function then constructs a `University` object, initializes classroom nodes, and creates weighted edges between feasible classroom pairs.

- **Interactively select classrooms**

The function **get_user_node** allows the user to select starting and target classrooms from a dynamically generated list. Input validation ensures that only valid selections are accepted, while providing a clean exit option that returns the user to the main menu.

- **Compute and display shortest paths**

Once the graph is constructed, the menu enables the user to initiate shortest-path computations between selected classrooms. The results are formatted and displayed using the **print_shortest_path** function, presenting both the sequence of classrooms and the total path cost.

- **Visualize the university layout**

The **visualize_graph** function provides a graphical representation of the university map using NetworkX and Matplotlib. Classrooms are displayed as nodes positioned according to their coordinates, while edges represent feasible paths based on distance constraints.

# 5 Main.py

To execute the program, one needs to navigate to the folder that contains the main.py file, and from a command prompt, type: python main.py. If executed correctly should be displayed:

```
Recommendation:
- Maximum Distance: 21
- Floor Weight: 1
These values are suggested when selecting option 2) Create Graph


    ========================
            MENU
    ========================
    1) Load Map
    2) Create Graph
    3) Find the shortest path between two classrooms
    4) Visualize Map
    5) Exit
```

Figure 1: Initial execution of main.py file.

A small hint is prompted to the user to initiate a maximum distance of 21 and a floor weight equal to 1, as the representation of the program is clearer with these.

After the user has gone through option 1, and option 2 (with max distance 21 and floor weight 1) is ready to either find the shortest path between two classrooms of choice or visualize the whole Map. Option 4) Visualize Map option results in:

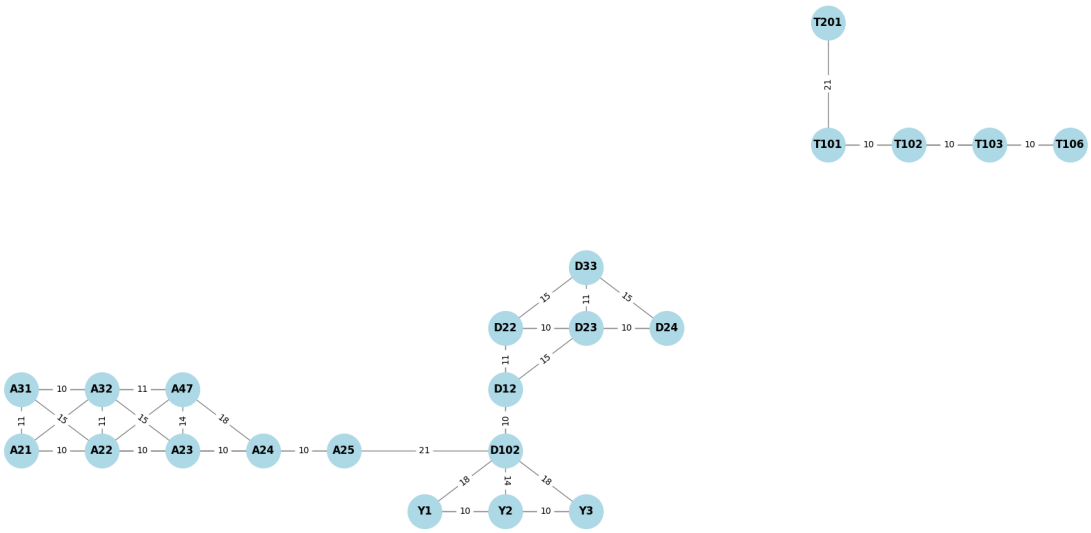



Figure 2: University Map

Then the user selects option 3 in order to find the shortest path between two nodes:

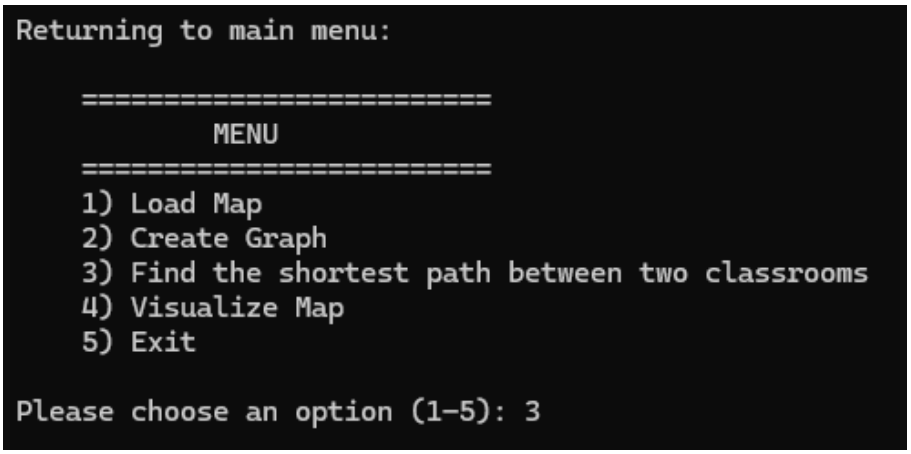
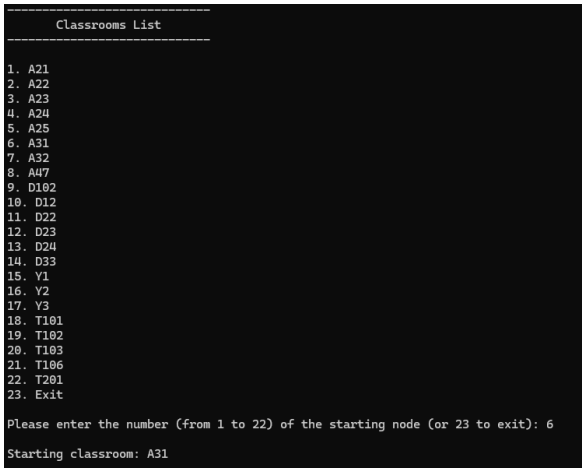
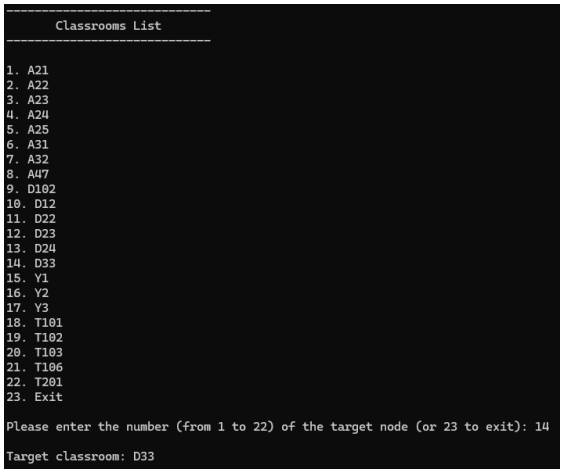



Figure 3: Option 3

which is followed by:




(a) Starting node journey




(b) Target node journey

Figure 4: Shortest-path selection process

which results in:


Figure 5: Shortest Path print

If the path is not feasible, it will display:


Figure 6: Not feasible path