Maximum Lifetime Scheduling in Wireless Sensor Networks

Filip Dąbrowski

Introduction:

The given application focuses on maximizing the lifetime scheduling of the Wireless Sensor Network (WSN). The network consists of many low-cost sensors each equipped with its own battery, scattered randomly in a geographical area of interest and connected by a wireless interface.

The application simulates area or target coverage by the given number of sensors and the sensor’s active range. Due to the deployment cost exceeding the overall sensor cost. The sensors are deployed in larger numbers than it is necessary to cover the given area. It results in the overlap of many sensor area coverage.

Application solves this problem by applying an algorithm that deactivates overlapping sensors and divides them into subsets to prolong the overall WSN lifetime within the designated area.

Requirements:

- The application must consist of the system simulating WSN including sensors, sensor’s range, representation of terrain and targets.

- The simulation algorithm used in the system has to be simulated annealing and genetic algorithms

- Application should perform correct simulation according to the input data such as number of sensors, sensor range, terrain size.

- An interactive graphical user interface is required, enabling users to perform simulations and modify input data such as the number of sensors, sensor range, number of targets, and select simulation mode (target/area coverage).

Used technology:

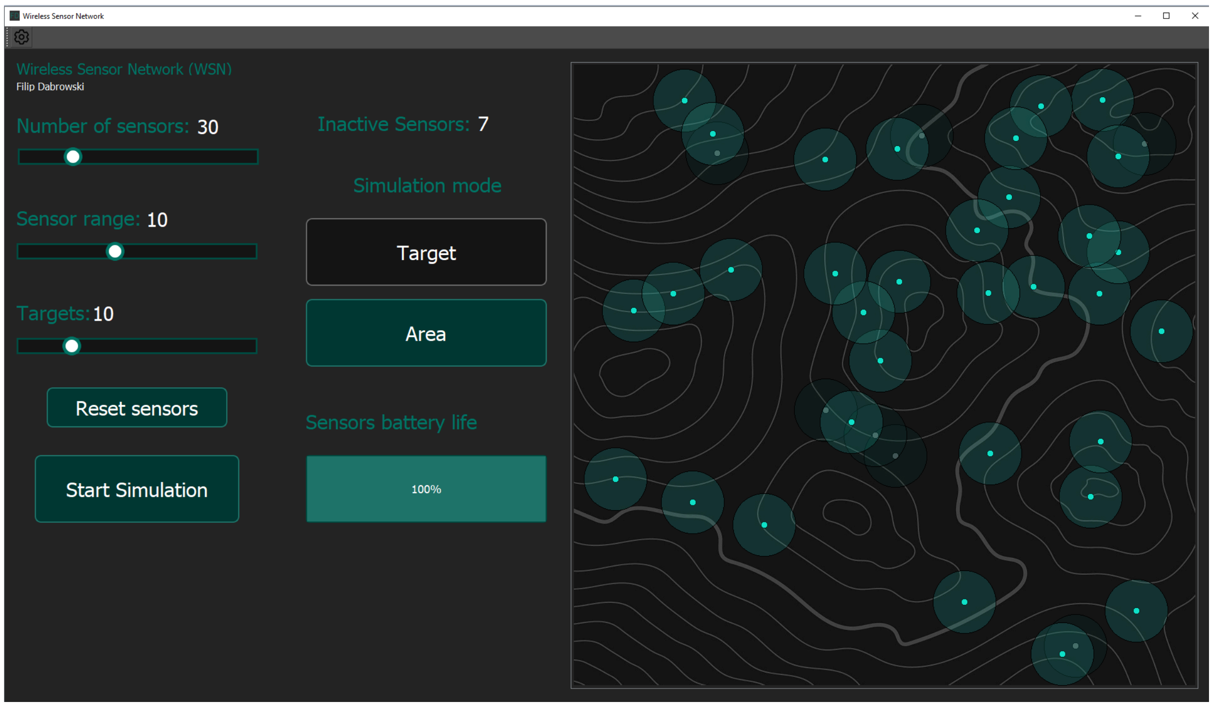
- **Python 3.12.3** - Programming language chosen for the full stack application development including front-end and back-end.
- **PyQt5 5.15.10** - A set of Python bindings for the Qt 5 cross-platform C++ libraries, facilitating API access for modern desktop development and GUI creation.
- **Matplotlib** - Python library used for visualizing math operations containing API similar to MatLab.
- **CSS** - Style sheet language compatible with the objects included in PyQt 5. Used to style the elements within GUI.

Installation:

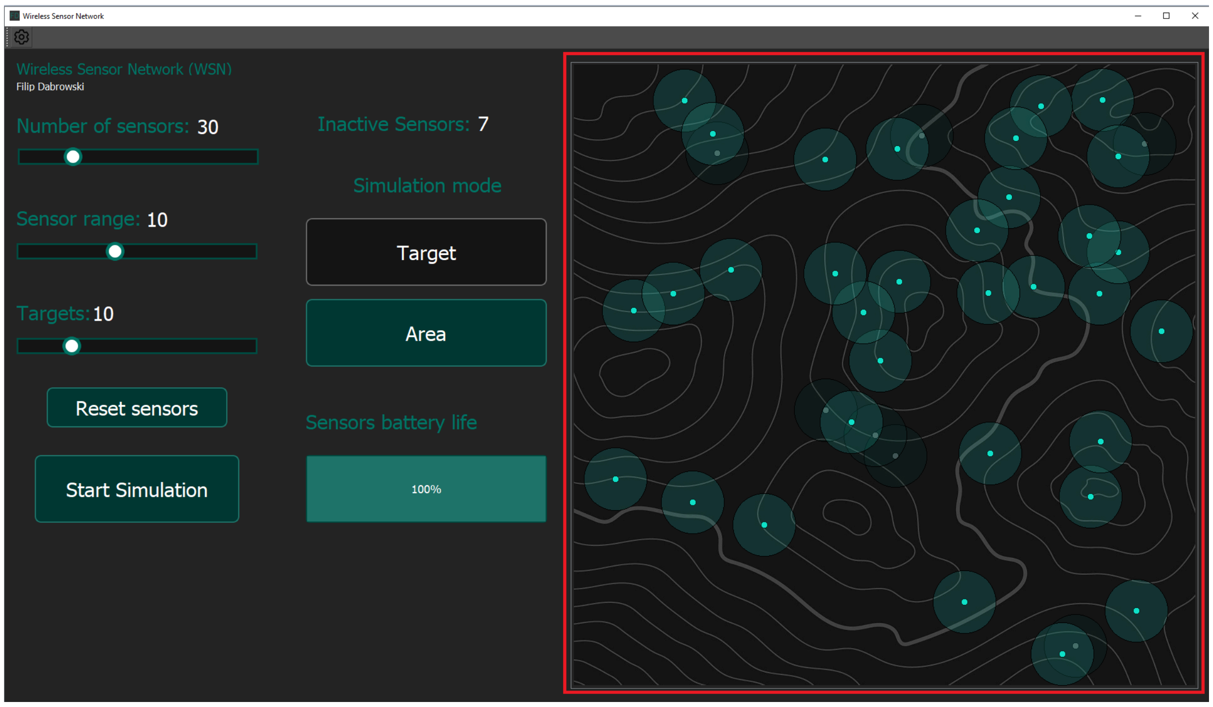
1. Clone the project repository from the GitHub repository:https://github.com/Ezarus12/Wireless-Sensor-Network-WSN-
2. Install python https://www.python.org/
3. Install **PyQt5** and **Matplotlib**
   pip install PyQt5
   pip install matplotlib
4. Run the application
   python main.py

Usage:

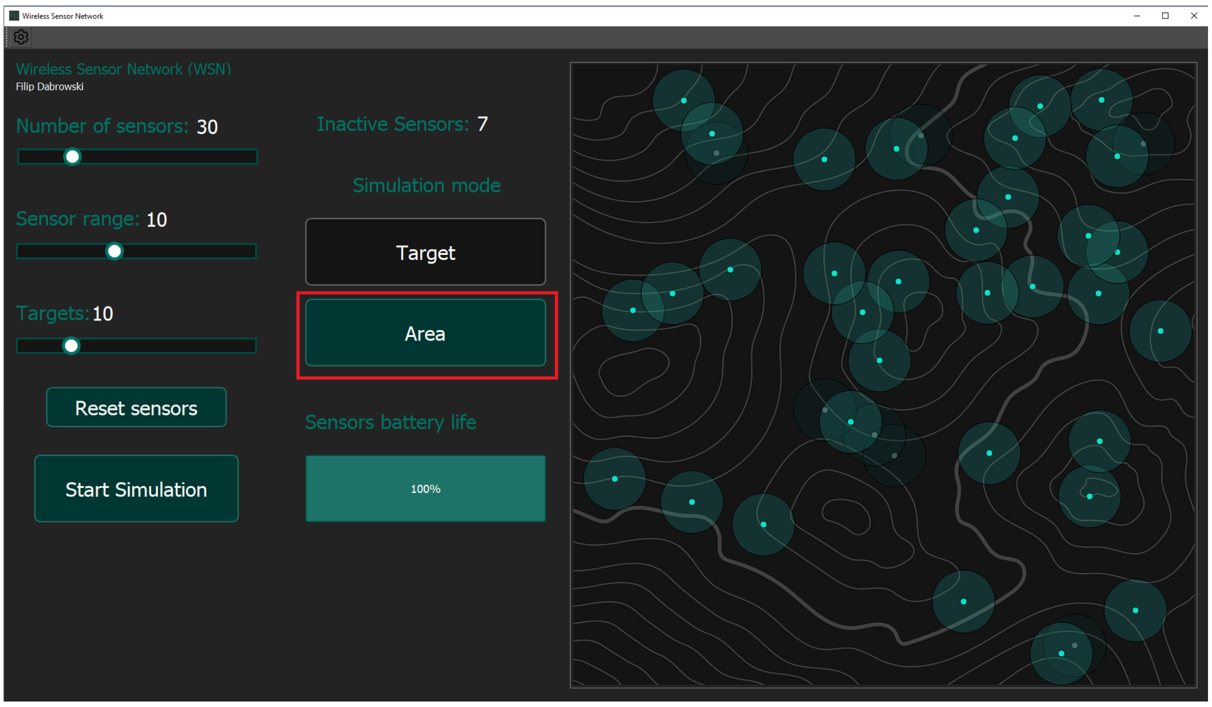
After launching the app, the application window opens on the screen.The main window consists of the adjustable sliders, simulation mode, reset and start button. On the right part of the screen there is and visual representation of the WSN network.



Terrain representation consisting of terrain image, active, inactive and turned off sensors.



Simulation mode buttons. Changes the type of the simulation area/target.


Wireless Sensor Network (WSN)
Filip Dabrowski

Number of sensors: 30

Sensor range: 10

Targets: 10

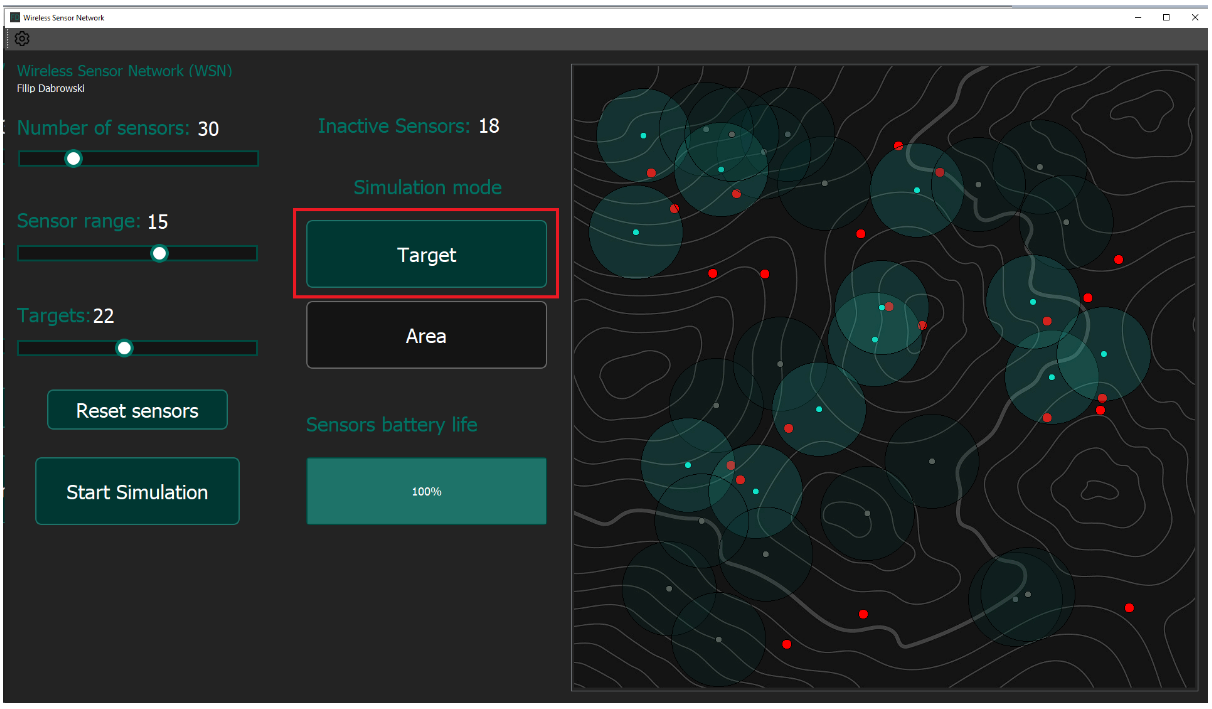
Inactive Sensors: 7

Simulation mode

Sensors battery life

100%

Target simulation mode show also targets represented by the red dot on the terrain


Wireless Sensor Network (WSN)
Filip Dabrowski

Number of sensors: 30

Sensor range: 15

Targets: 22

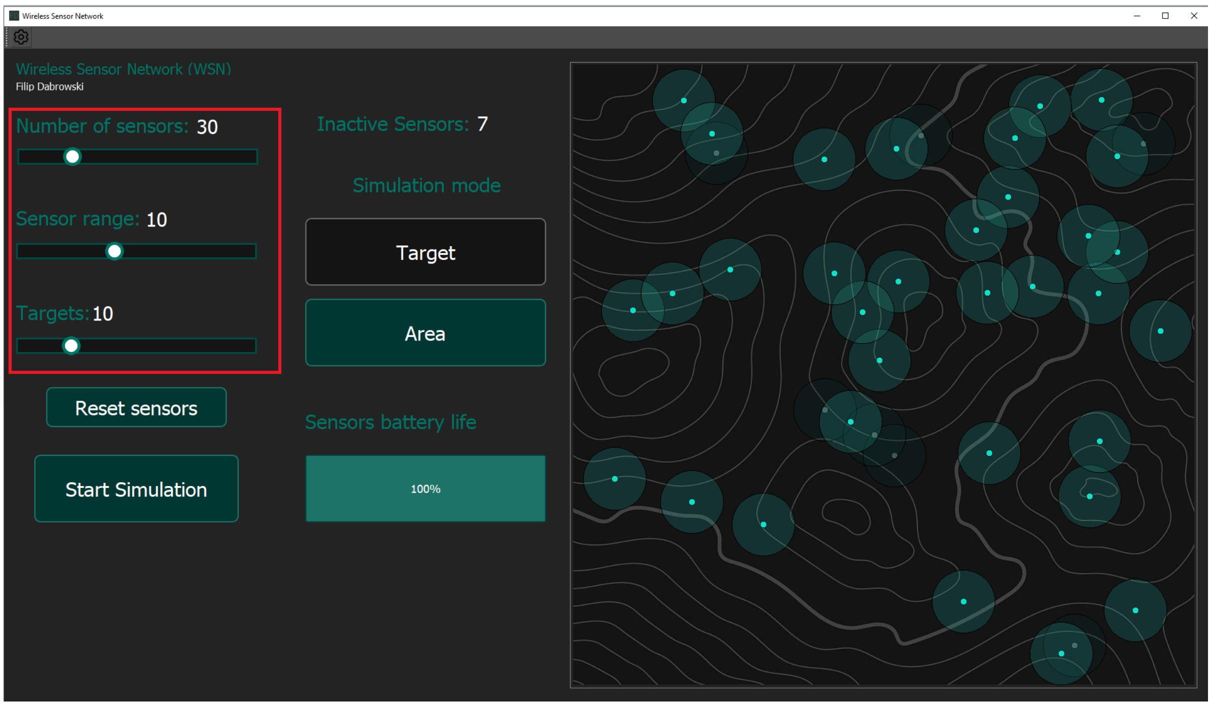
Inactive Sensors: 18

Simulation mode

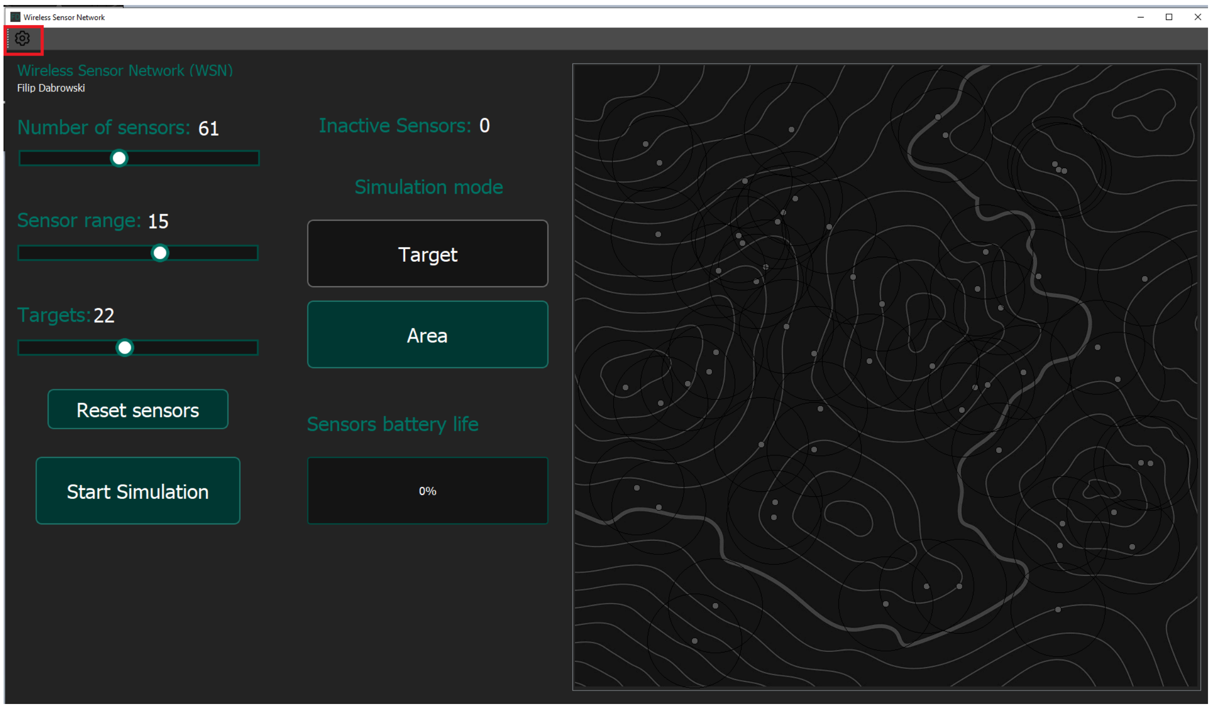
Sensors battery life

100%

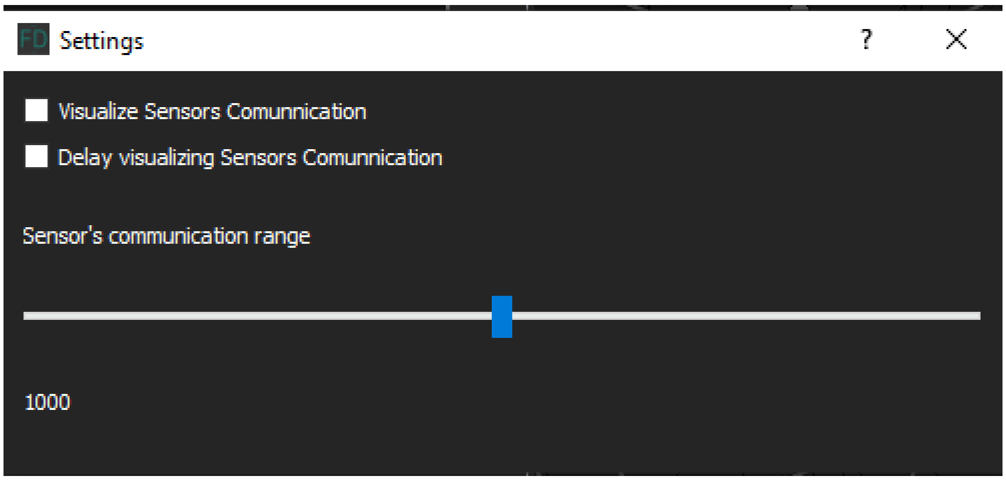
Sliders used for adjusting the number of sensors, their range and number of targets (available only in target simulation mode)



Options button in the toolbar at the top of the screen.



When opened, a new window pops up allowing the user to select if the sensor communication should be visualized and should it be delayed. At the bottom there is also a slider allowing users to change the range that sensors communicate with each other.


Settings

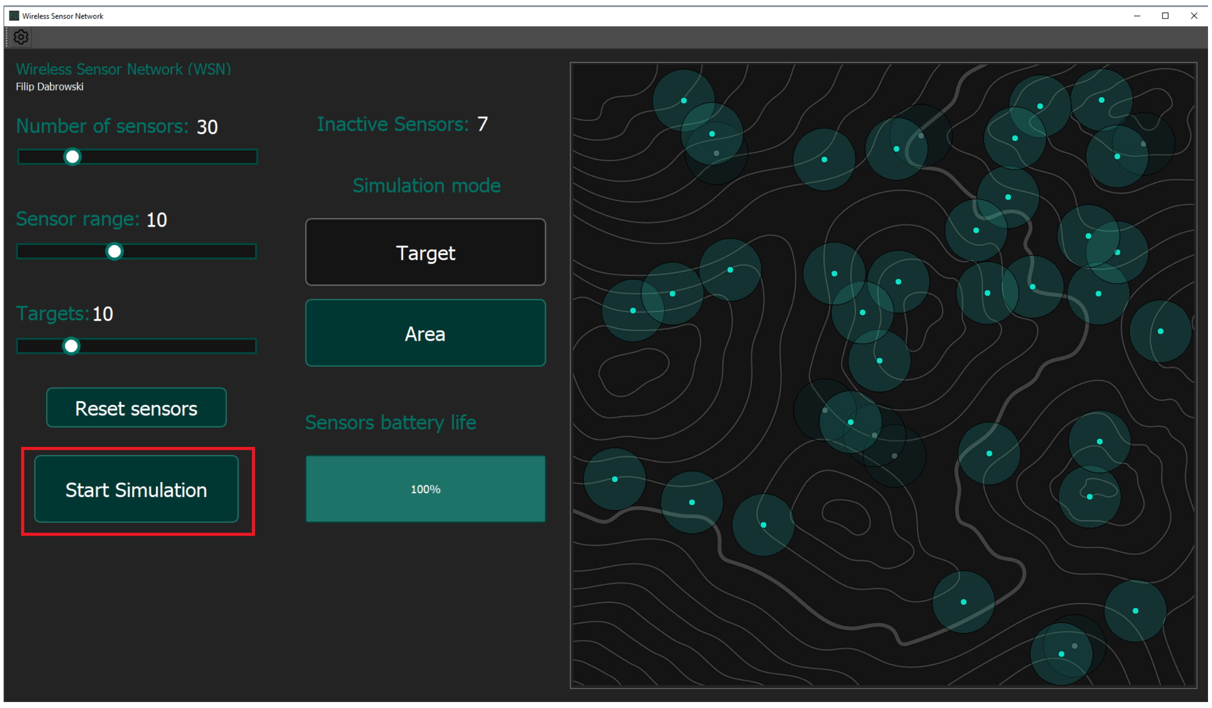
☐ Visualize Sensors Comunnication

☐ Delay visualizing Sensors Comunnication

Sensor's communication range

1000

To start the simulation press the “Start simulation” button


Wireless Sensor Network

Wireless Sensor Network (WSN)
Filip Dabrowski

Number of sensors: 30

Sensor range: 10

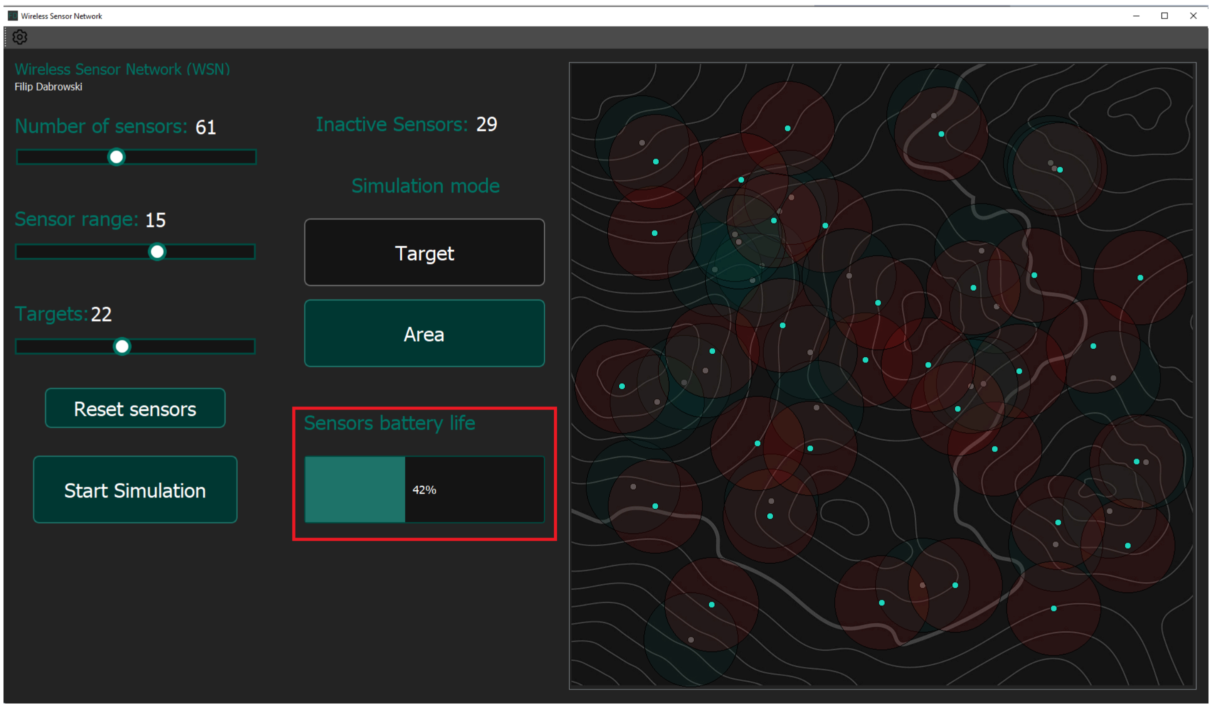
Targets: 10

Inactive Sensors: 7

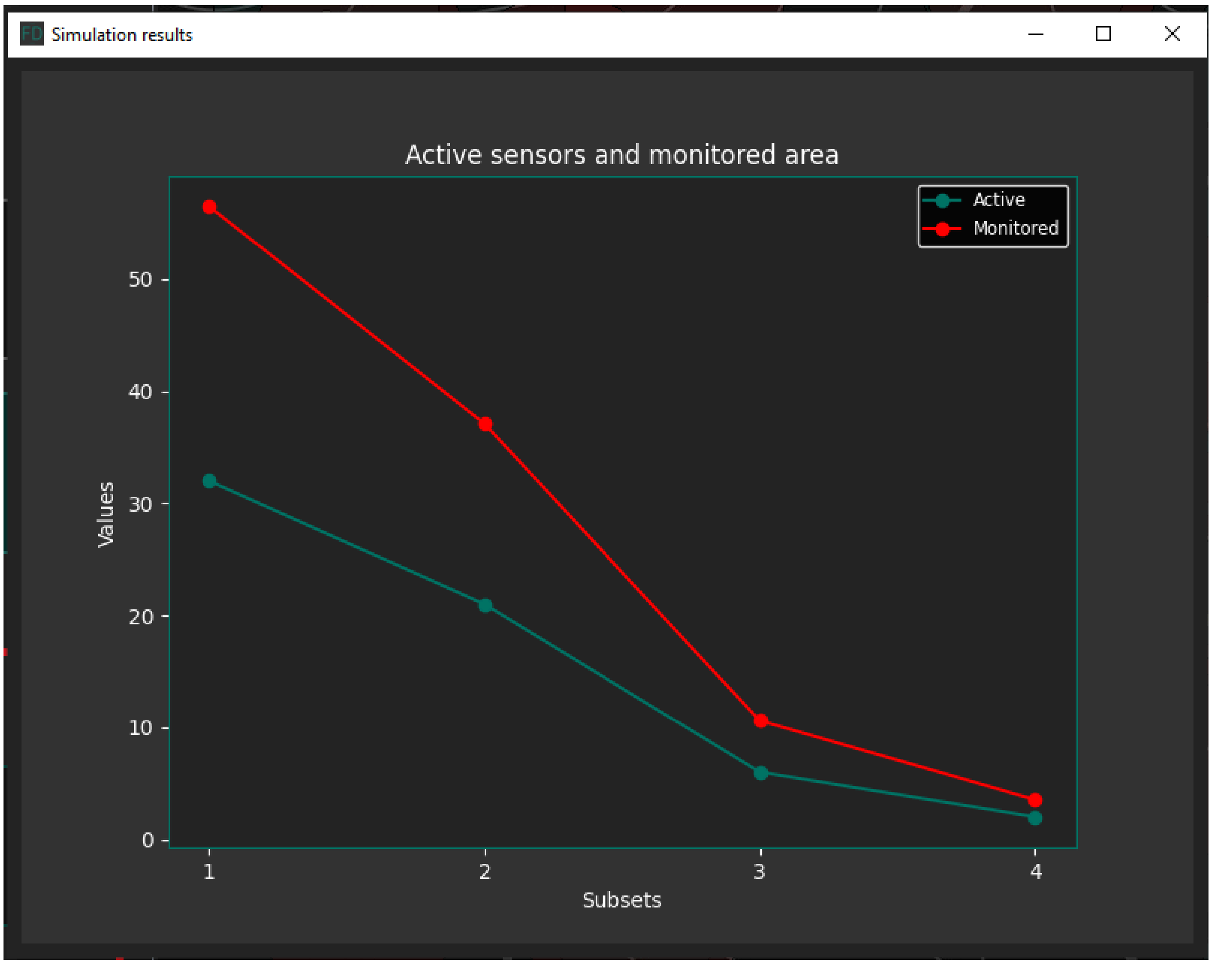
Simulation mode

Sensors battery life: 100%

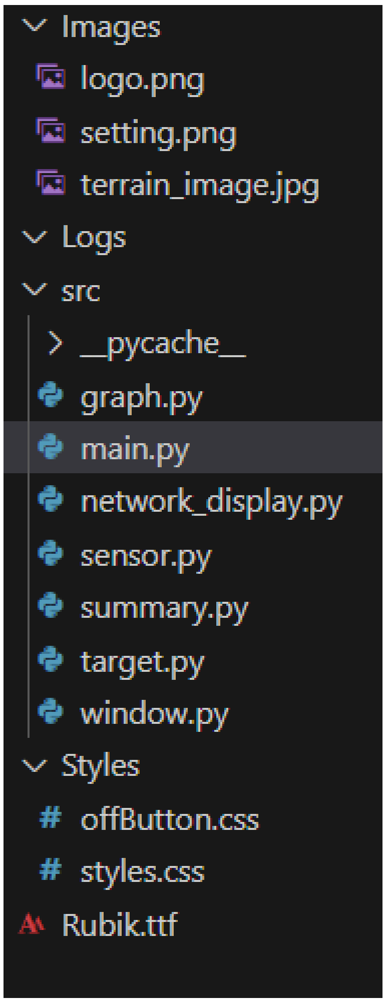
During the simulation the active sensor range turns red and slowly fades according to the current battery level represented by Sensors battery life widget with progress bar.



At the end of the simulation a new window opens showing the graph of the representing runned simulation.



Directory structure:



```
Images
    logo.png
    setting.png
    terrain_image.jpg
Logs
src
    __pycache__
    graph.py
    main.py
    network_display.py
    sensor.py
    summary.py
    target.py
    window.py
Styles
    offButton.css
    styles.css
Rubik.ttf
```

```
WSN/
│
├── Images/
│        ├── terrain_image.jpg
│        ├── setting.png
│        └── logo.png
│
│
├── Logs/
└── …
│
├── src/
│   ├── main.py
│   ├── network_display.py
│   ├── sensor.py
│   ├── target.py
│   ├── summary.py
│   ├── graph.py
│   └── window.py
│
├── Styles/
│   ├── styles.css
│   └── offButton.css
│
├── Rubik.ttf
│
│
└── tests/
   ├── test_network_display.py   y
   ├── test_sensor.py
   └── …
```

Code documentation:

Modules:

<u>main.py:</u>

Main module used to create QApplication, Window and executes them.

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Window()
    window.show()
    sys.exit(app.exec_())
```

<u>window.py</u>

Main module used mostly for creating, adjusting and styling window along with its elements and widgets. Connects all of the UI elements with the objects used to simulate WSN in other modules

Classes:

.**Window**: Represents main application window, initializes the UI elements and connects them with the simulation objects. Manages user input from the UI elements like buttons, slider, checkboxes and connects them to the correct methods.

Methods:

- **create_toolbar() -** Creates toolbar inside of the window and initializes object inside of it like QAction used to access the setting window.

```
def create_toolbar(self):
    toolbar = QToolBar()
    Settings = QAction(QIcon("setting.png"), "Settings", self)
    Settings.triggered.connect(self.open_settings_window)
    toolbar.setStyleSheet("background-color: #4F4F4F;")
    toolbar.addAction(Settings)
    self.addToolBar(toolbar)
```

- **open_settings_window() -** Creates a new window on a screen with other application settings not accessible from the main window like: visualizing sensor communication , delaying visualization of the sensor's communication and slider used to change sensor's communication range.

- **create_widgets() -** Method used to initialize and add to the main window all of the widgets. Every widget is initialized with proper text, parameters and objects.Most of

the widgets take their stylesheet from style.css file but few of them take in place styling in form of a string as an argument.
Example widget:

```
    #Number slider
    self.numberSlider = QSlider(Qt.Horizontal, self.centralwidget)
    self.numberSlider.setGeometry(QtCore.QRect(20, 290, 311, 51))
    self.numberSlider.setMinimum(0)
    self.numberSlider.setMaximum(150)
    self.numberSlider.setTickInterval(1)
    self.numberSlider.setValue(30)
    self.numberSlider.setStyleSheet(str(stylesheet, encoding='utf-8'))
```

- **resize_event()** - Method handling widgets size, position and font size during window resizing. All of the widgets are placed according to their percentage position and size relating to window size.
  Example resized widget:

```
self.numberSlider.resize(math.floor(width*0.2),math.floor(height*0.04))
self.numberSlider.move(math.floor(width*0.011),math.floor(height*0.14))
```

- **reset()** - Method connected to the reset button. Clears three lists used in the graph, disables the UI and generates new WSN representation on the screen. After generation is finished, it enables UI.

- **draw_network()** - Updates the inactiveSensorNum label and passes the method to the networkDispaly object.
- **update_label(label_widget, value)** - Updates the label text to the **value** in the given widget **label_widget.**

- **decreaseBatteryLife(fileName) -** Decreases battery life by 1 and calls fade_range area method of the sensor object. If battery life is <= 0 stops the timer starts the next simulation by calling simulation() method on the NetworkDisplay object and updates inactiveSensorsNum label. If next subsets are available, call startBatteryDecrease for the next subset, otherwise set battery life to 0, enable the UI and show the calls graphWindow().

- **startBatteryDecrease(fileName)** - Method disables the UI for the time of the simulation. If the simulation is finished it shows the correct message window using QMessageBox according to the simulationMode. Else sets progressBar to 100, creates timer, counts subsets, according to the simulationMode calls formatDataGraph and creates Summary object for log files. Finally connects the timer to the decreaseBatteryLife method and starts the timer.

- **changeUIstate(state)** - Enables or disables the widgets on the screen according to the given **state**

- **changeUIstateAllowReset(state)** - Calls **changeUIstate(state)** and sets resetButton state to true.

- **startSimulation() -** Method resets subset count to 0, creates new fileName using the current date and time finally calls the sttartBatteryDecrease method.

```
def startSimulation(self):
    self.subset = 0
    fileName = "Simulation_" + datetime.now().strftime("%Y-%m-%d_%H-%M-%S") + ".txt"
    self.startBatteryDecrease(fileName)
```

- **formatDataGraph() -** Formats data from the lists self.activeSensorsGraph, self.monitoredAreaGraph, self.monitoredTargetsGraph setting them accordingly to the GraphWindow class.

- **changeSimulationMode(mode, stylesheet, buttonStylesheet) -** Method connected to the Area and Target buttons. Changing simulationMode according to the **mode** disables visualizing sensor communication for the target area. Finally changes button style sheets.

```
    def changeSimulationMode(self, mode, stylesheet, buttonStylesheet):
        if mode == "Target":
            self.network_display.simulationMode = 'T'
            self.network_display.delayVSN = False
            self.network_display.visualizeSensorsCommunication = False
            self.targetButton.setStyleSheet(str(stylesheet, encoding='utf-8'))
            self.areaButton.setStyleSheet(str(buttonStylesheet, encoding='utf-8'))
        elif mode == "Area":
            self.network_display.simulationMode = 'A'
            self.targetButton.setStyleSheet(str(buttonStylesheet, encoding='utf-8'))
            self.areaButton.setStyleSheet(str(stylesheet, encoding='utf-8'))
        else:
            print("Mode must be \"Target\" or \"Area\"")
```

network_display.py

Consists of NetworkDisplay class managing the simulation algorithm, simulation state and date used to perform it. NetworkDisplay class also represents the terrain, sensors, sensor ranges, targets, and their state during the simulation inside the QGraphicScene object.

Methods:

- **draw_network() -** draws the graphical representation of the currently generated WSN. Adds to the scene terrain images, targets (if the simulation mode is set to Target), sensors and their ranges accordingly to the sensorNum attribute. Each sensor's position is generated randomly within the given area and according to the simulation mode the correct subset is being called**.**

- **generateTargets()** - randomly generates the position of the given number of targets and adds them to the graphic scene.

- **visualizeNetwork()** - visualizes the sensor's communication within WSN by drawing a line between the currently checked sensor and second one. If the delay flag is True, the drawing of each line is delayed by 10 ms.

- **createSubset()** - For each sensor calculates the distance to the all of the other sensors and according to the detectionRange runs the main algorithm deciding if the sensors should remain active. If visualizeSensorsCommunication flag is True class visualizeNetwork method.

```python
def createSubset(self):
    for i, sensor in enumerate(self.sensors):
        for j, other_sensor in enumerate(self.sensors[i+1:], start=i+1):
            distance = math.sqrt(((sensor.xPos - other_sensor.xPos) ** 2) + ((sensor.yPos -
other_sensor.yPos) ** 2))
            if distance <= self.detectionRange:
                if distance <= (self.sensorRange / 2) and sensor.isActive and sensor.hasPower:
                    self.inactive_sensors += 1
                    sensor.isActive = False
                    sensor.change_color_inactive()
                if self.visualizeSensorsCommunication:
                    self.visualizeNetwork(sensor, other_sensor)
```

- **createSubsetTarget()** - Similar method to the createSubset but uses an algorithm adjusted for the target monitoring in the WSN network.’

```python
def createSubsetTarget(self):
    for sensor in self.sensors:
        for target in self.targets:
            distance = math.sqrt(((sensor.xPos - target.xPos) ** 2) + ((sensor.yPos -
target.yPos) ** 2))
            if distance <= (self.sensorRange / 2) and not target.monitored:
                target.monitored = True
                sensor.monitoring = True
                self.monitoringAnyTarget = True
        if not sensor.monitoring:
            self.inactive_sensors += 1
            sensor.isActive = False
            sensor.change_color_inactive()
```

- **nextSubset()** - Method prepares the next simulation subset by turning off active sensors at the end of subset life indicated by the battery life. Change their color to off. Rest of the sensors are set to active and turned off sensor’s objects are being deleted.When all of the sensors state has been up[dated method class proper createSubset method according to the simulationMode.

- **simulation()** - Method calls nextSubset() and updates UI by calling processEvents() method on QApplication object.

- **ResetSensors (sensorNum, range, targetNum) -** Method clears UI scene representing WSN and sets sensorNum, sensorRange, targetNum according to the arguments. Finally the method calls draw_network().

- **load_terrainImage(name) -** Method load terrain image from a file path **name.**

- **Setters:**

```
def set_sensorNum(self, num):
    self.sensorNum = num

def set_sensorRange(self, num):
    self.sensorRange = num*10

def set_targetNum(self, num):
    self.targetNum = num

def set_detectionRange(self, num):
    self.detectionRange = num
```

sensor.py

Includes Sensor class inheriting from QGraphicsEllipseItem. Constructor parameters x, y, size, range used to init parent object except the "range". Consists also of rangeOpacity set to 50% and three flags:

- isActive - set True when sensor is active
- hasPower - set True when sensor has power and False when the sensor has been used in the current subset
- monitoring - set True when sensor is monitoring at least one target

  Methods:
  - draw_range() - draws the sensor's range according to range attribute and aligns the range with the center of the sensor.
  - Three UI methods changing the sensor and sensor range color accordingly:
    - change_color_inactive()
    - change_color_active()
    - change_color_off()

```
def change_color_active(self):
        self.setBrush(QBrush(sensor_color_active))  # Sensor color
        self.range_area.setBrush(QBrush(QColor(32, 179, 162,
self.rangeAreaOpacity))) # Range area color
```

- fade_range_area(value) - changes the opacity of the range area by the given value, where: 100 = 50% of opacity.

```
    def fade_range_area(self, value):
        opacity = math.floor(self.rangeAreaOpacity*((value/100)))
        self.range_area.setBrush(QBrush(QColor(255, 0, 0,
opacity)))
```

target.py

Small module consisting of one class Target inheriting from QGraphicsEllipseItem class for the graphical representation. Constructor parameters x, y, size used to init parent object. Attributes: xPos, yPos and one flag “monitored” set to True when the target is being monitored by at least one sensor and False otherwise..

Methods:

- Only simple getters

summary.py

Consist of Summary class used to create, write and save log_files for each performed simulation.

Methods:

- **simulation_log_message_area(sensors, fileName, subset)** - Creates a new file for the current area simulation writing: subset, active sensors and area coverage.
- **simulation_log_message_area(sensors, fileName, subset)** - Creates a new file for the current target simulation writing: subset, active sensors and monitored targets.

graph.py

Module responsible for creating a new window at the end of simulation. The window consists of a graph representing current simulation and shows data like active sensors and monitored area/targets during each subset.

Classes:

GraphWindow(QMainWindow) - Class responsible for creating a new window at the center of the screen. According to the simulation mode, it creates a graph object. Inherits functionality from a QMainWindow class.

GraphBase(FigureCanvas) - Base class for other graph classes. Inherits from FigureCanvas class. Constructor takes as arguments all of the necessary data used to create a graph.

Methods:

- **plot()** - Creates graph for the current simulation inside of the window. The graph consists of two axis displaying subsets and active sensors/monitored data.

GraphArea(GraphBase) - Child class of the GraphBase used to create graphs for area simulation mode.

GraphTarget(GraphBase) - Child class of the GraphBase used to create graphs for target simulation mode.

Testing:

Through the whole development stage of the application the manual tests were executed for every feature integrated into the application. The only input allowed by the user comes from interacting with the built in widgets like buttons, sliders and checkboxes so most of the major issues were easily detected during the manual testing.

Unit tests have been created for the target.py and sensor.py modules and put in the “Tests” directory. To conduct the unit testing the python **unittest** module has been used.

The tests enforced disabling the UI widgets for the user during the few stages of application operations. The UI is disabled during visualizing sensor network communication and simulation itself.

After the tests application is stable and fully functional without major bugs and issues with performance.

Author:

Filip Dąbrowski
Responsibilities:

- Design and implementation of the Wireless Sensor Network (WSN) simulation algorithms.
- Design of the UI.
- Development of the core modules including sensor management, target tracking, and network visualization.
- Integration of the graphical user interface using PyQt5.
- Implementation of logging mechanisms and performance optimization.
- Writing and maintaining project documentation.
- Conducting testing and debugging to ensure the robustness of the application.