WSN Lifetime Maximization Simulator — Dokumentacja

Autor: Jakub Morawski (nr albumu: 126597, Infomatyka II rok, 1 stopień, studia stacjonarne)
Temat projektu: Metody rozwiązywania problemu maksymalizacji czasu życia sieci sensorowej – metody dokładne
Język UI: polski
Środowisko: Python 3.11.9

1. Wprowadzenie

Aplikacja symuluje problem pokrycia przestrzeni dwuwymiarowej (2D) przez bezprzewodową sieć sensorową (WSN). Kluczowym celem jest znalezienie optymalnego rozmieszczenia aktywności sensorów, aby zapewnić pełne pokrycie wszystkich punktów pomiarowych (POI) przy minimalnym zużyciu energii. Program wykorzystuje model programowania liniowego z całkowitymi zmiennymi (ILP) do rozwiązania problemu optymalizacji. Udostępnia także interfejs graficzny, oparty na bibliotece Tkinter, do wizualizacji rozmieszczenia sensorów, POI, ścieżek przesyłania pakietów (komunikacja sensorów) a także do sterowania parametrami symulacji.

2. Architektura systemu

Główne komponenty:

- Model: SensorCover-logika symulacji, algorytmy, komunikacja i zarządzanie energią.

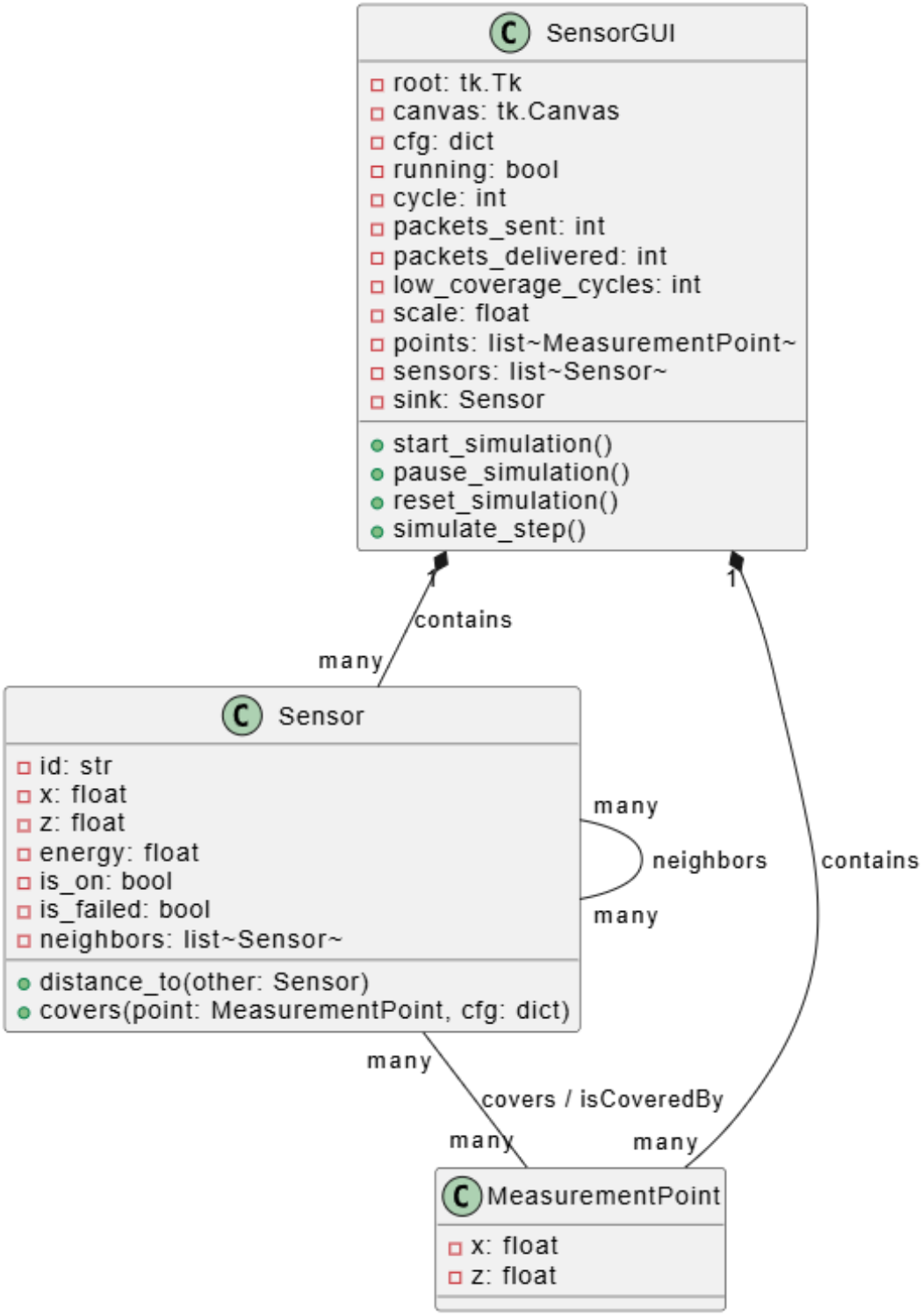
- Encje: Sensor, MeasurementPoint, Sink

- UI: SensorGUI– komponenty Tkinter do parametryzacji, sterowania i wizualizacji.

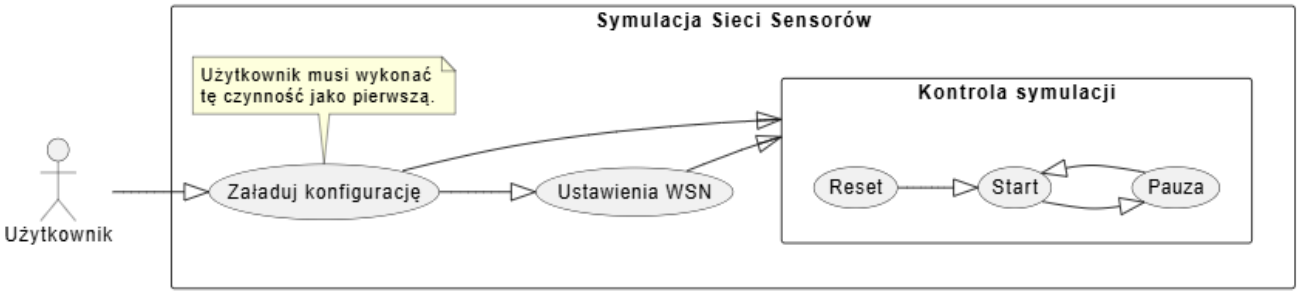
Diagramy zostały wygenerowane i zwizualizowane za pomocą rozszerzeń: PyUML Generator oraz PlantUML.
Linki:
https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml
https://marketplace.visualstudio.com/items?itemName=IuriPavani.python-uml

Aby stworzyć graf pliku .py należy w VS Code w folderze projektu kliknąć prawym przyciskiem na plik i następnie kliknąć „Generate UML Class Diagram”. To stworzy plik .txt, który należy otworzyć i następnie użyć skrótu klawiszowego Alt + D, aby wyświetlić graficznie diagram.

2.1 Diagram klas UML



2.2 Diagram przypadków użycia (Use Case)



3. Technologie i narzędzia

- Język: Python 3.11.9
- Biblioteki: tkinter(GUI), configparser (czytanie danych konfiguracyjnych), pulp (optymalizacja)
- IDE: Visual Studio Code
- Zarządzanie pakietami: pip

4. Instalacja i konfiguracja — krok po kroku

4.1. Wymagania wstępne

- System: Windows 10/11, macOS 12+, Linux (Ubuntu 20.04+)
- Python 3.10 lub nowszy

4.2. Instalacja Pythona

Pobierz instalator ze strony https://www.python.org/downloads/release/python-3137/ i zaznacz „Add Python to PATH” (Windows).

4.3. Utworzenie i aktywacja wirtualnego środowiska

```
# Windows
python -m venv nazwa
nazwa\Scripts\activate

# macOS/Linux
python3 -m venv nazwa
source nazwa/bin/activate
```

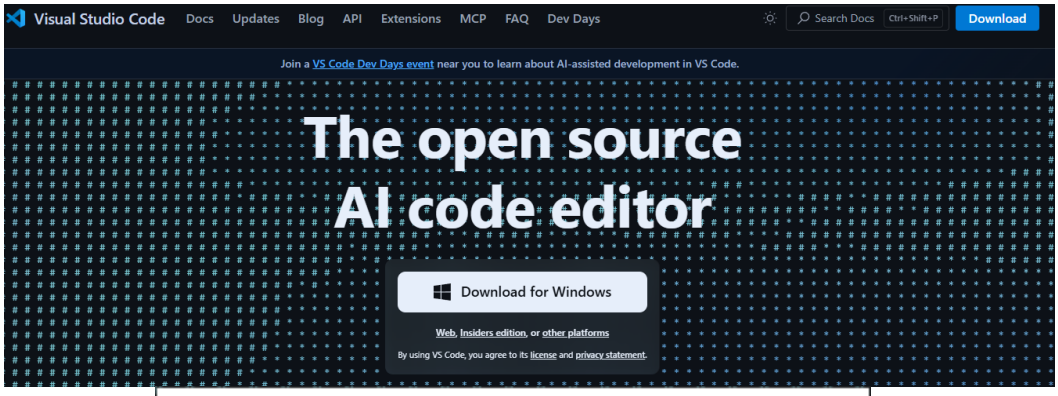
4.4. Instalacja pakietów

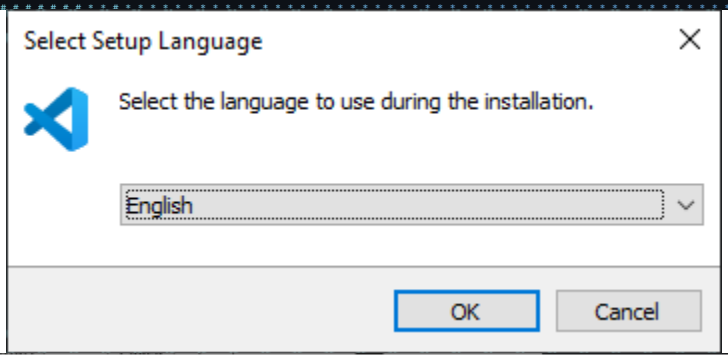
```
pip install --upgrade pip
pip install pulp
pip install configparser
pip install tkinter
```

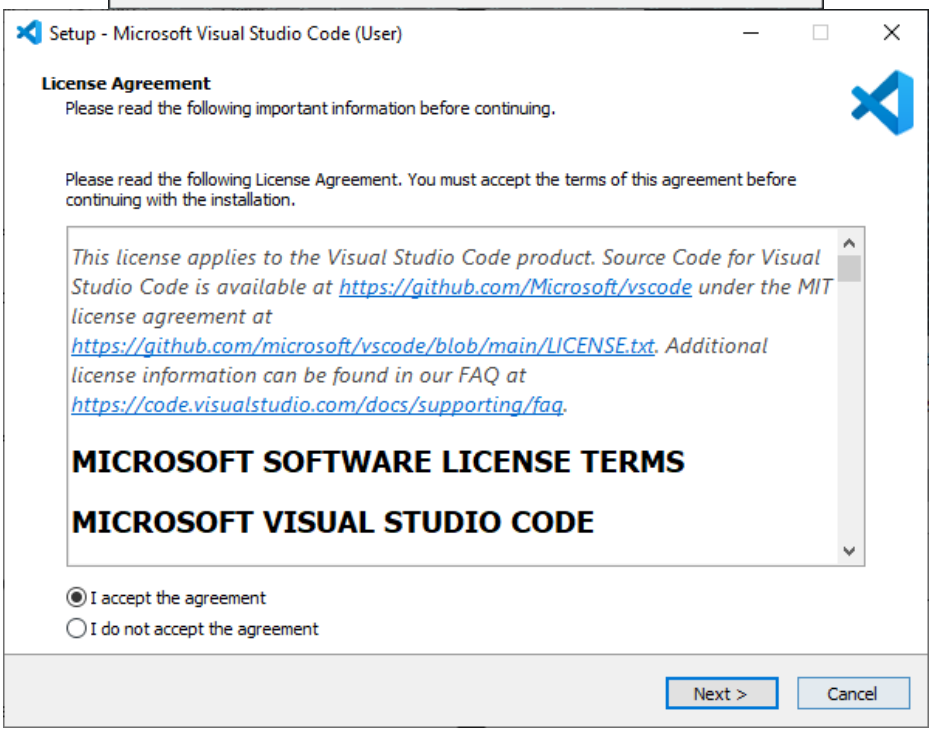
4.5. Instalacja i konfiguracja IDE

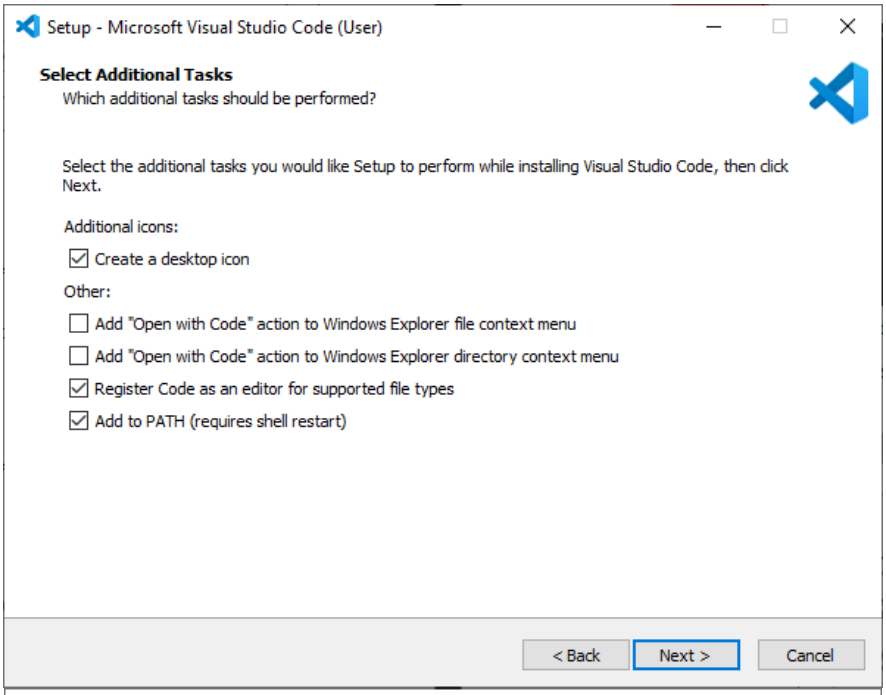
Visual Studio Code:

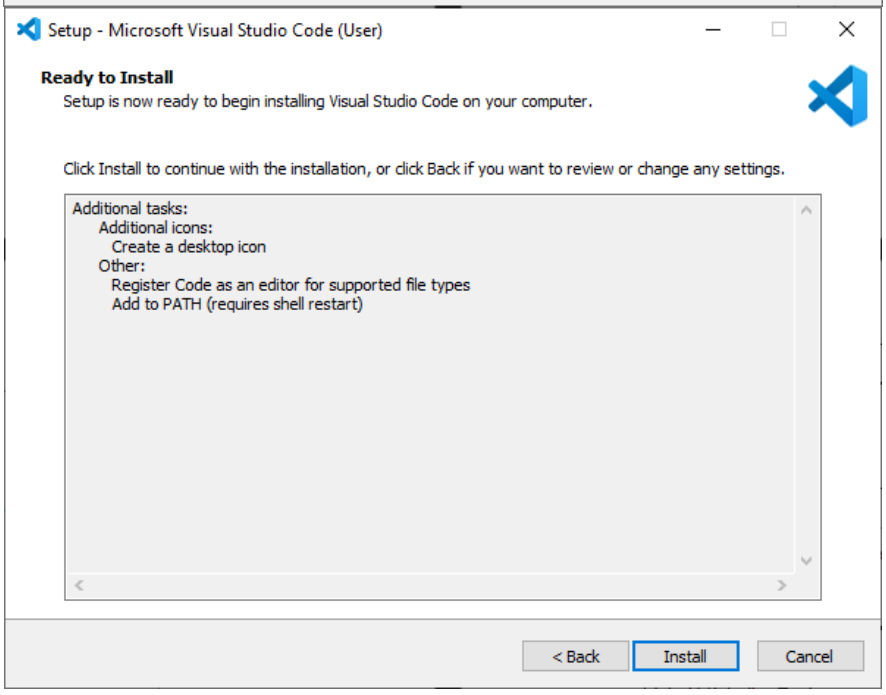
1. Zainstaluj Visual Studio Code https://code.visualstudio.com/











2. Pobranie rozszerzeń
W VS Code, przejdź do zakładki "Extensions" (ikona kwadratu po lewej stronie).
Wyszukaj i zainstaluj następujące rozszerzenia: Python, PyUML Generator oraz PlantUML

4.6. Tworzenie środowiska wymaganego do skompilowania kodu oraz jego uruchomienia.

Kliknąć File (lewy górny róg) -> Otwórz folder projektu -> Kliknąć Terminal (lewy górny róg) -> New Terminal -> Wpisac w terminal: python main.py

4.7. Generowanie dokumentacji z kodu (pdoc)

Dokumentacje wygenerowano w formie HTML i później zamieniono na format PDF używając pdoc oraz przeglądarki (live server). Poniżej instrukcja:

W terminal wpisac:

pip install pdoc
pdoc -o docs sensor_network_simulator.py
W folderze projektu powstanie folder docs, gdzie będzie plik main.html. Należ go otworzyć w przeglądarce (prawy przycisk na plik -> Open with live server) i w przeglądarce kliknąć Drukuj -> Drukuj jako PDF.

5. Instrukcja użytkownika

5.1. Uruchomienie i interfejs

Po uruchomieniu aplikacji, otworzy się okno „Symulacja Sieci Sensorów", gdzie widoczne będzie 5 przycisków: Załaduj konfigurację, Start, Pauza, Reset oraz Ustawienia WSN.

5.2. Kluczowe funkcjonalności

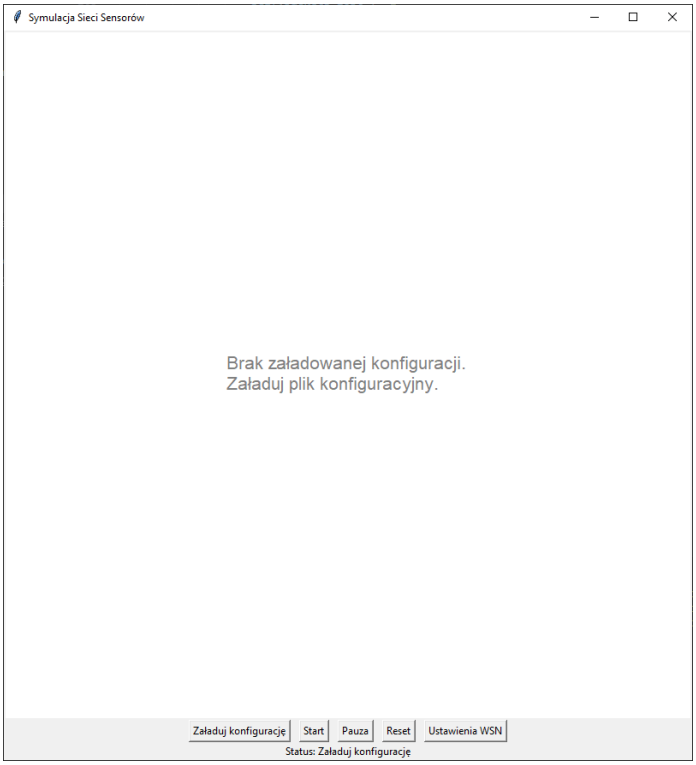
- Parametry symulacji: Rozmiar pola (szerokość i wysokość), liczba POI, liczba sensorów, zasięg sensorów, energia sensorów, minimalny poziom pokrycia punktów, koszt nadawania, odbierania, trybu bezczynności, trybu uśpienia oraz prawdopodobieństwo awarii i prawdopodobieństwo utraty pakietu danych, a także częstotliwość cykli
- Sterowanie: Załaduj konfigurację (wczytanie pliku .ini), Start (rozpoczęcie symulacji), Pauza (zatrzymanie symulacji), Reset (reset symulacji z nowym położeniem POI) oraz Ustawienia WSN (ręczne ustawienie parametrów symulacji).
- Interakcja: klikanie przycisków lewym guzikiem myszy.
- Metryki na żywo: aktualny cykl (krok), aktualne pokrycie POI, pracujące sensory, aktywne sensory (nawet tryb uśpienia – byle nie wyłączone), ilość awarii, które wystąpiły, ilość utraconych pakietów danych
- Eksport: plik .txt z informacją o ilości kroków, PDR, latencji i stanie każdego czujnika (id, poziom energii, kolor na grafie i sąsiedzi)

5.3. Jak korzystać — krok po kroku

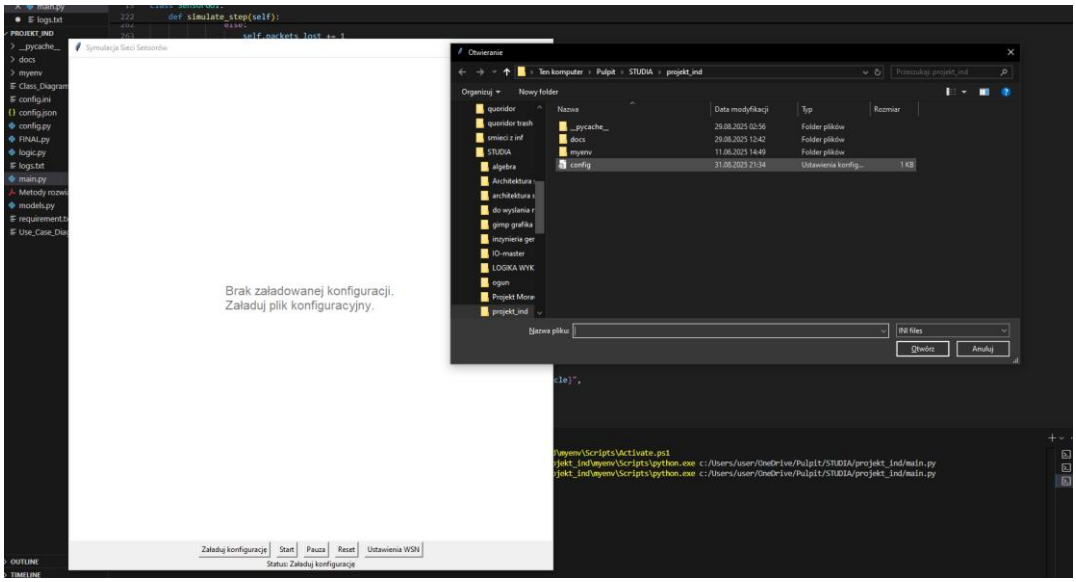
1. Wgraj plik z konfiguracją (parametry dla symulacji)
2. W razie chęci, zmień wartości parametrów przyciskiem „Ustawienia WSN” ręcznie.
3. Kliknij „Start”.
4. Obserwuj grafikę w oknie.
5. Po zakończeniu, opcjonalnie zrestartuj/zmień parametry i rozpocznij nową symulację.

5.4. Zrzuty ekranu aplikacji

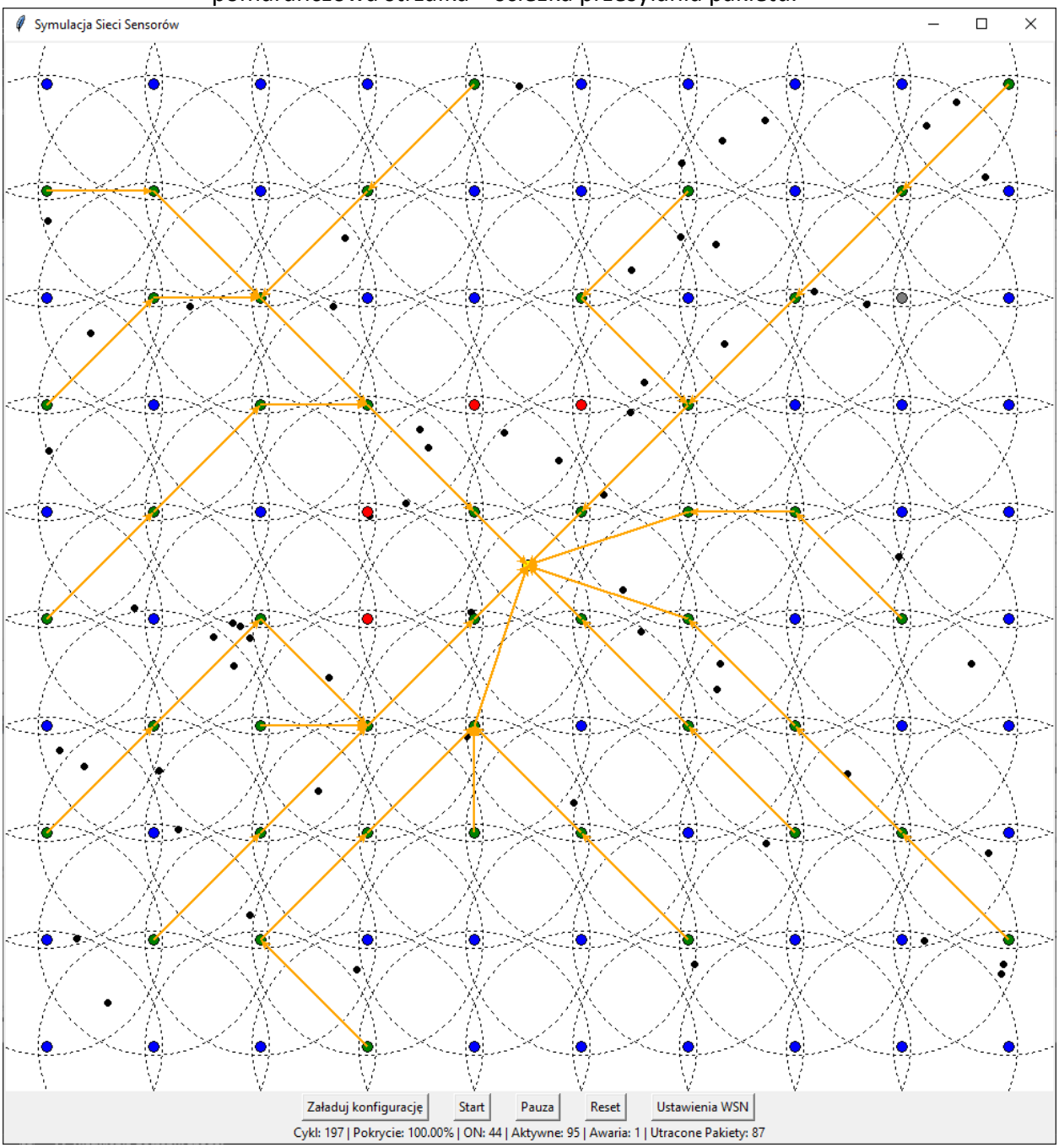
Okno startowe z widocznym informacją na środku co robić oraz dostępnymi przyciskami pod mapą.



Wgrywanie konfiguracji – wybranie pliku rozszerzenia .ini (najlepiej z tego samego folderu).



Mapa z załadowanymi parametrami po starcie symulacji: niebieskie koło – sensor (uśpiony), zielone koło – sensor pracujący, czerwone koło – sensor rozładowany, szare koło – sensor po awarii, żółte koło – SINK, czarne koło – POI, czarny okrąg – zasięg sensora, pomarańczowa strzałka – ścieżka przesyłania pakietu.



6. Dokumentacja API (kluczowe klasy/metody/pola)

6.1. main.py

klasy:

-class SensorGUI: Ta klasa jest kluczowa, ponieważ odpowiada za cały interfejs graficzny i logikę sterującą symulacją.

Metody:

-__innit__(self,root): Inicjalizuje interfejs użytkownika, tworząc główne okno, przyciski i płótno do rysowania.

-load_config_gui(self): Pozwala użytkownikowi wybrać plik konfiguracyjny, a następnie dostosowuje skalę i rozmiar okna.

-draw_scene(self, paths=None): Rysuje na mapie w GUI wszystkie czujniki i punkty do pokrycia, a także ścieżki przesyłania danych.

-simulate_step(self): To serce symulacji, w każdym cyklu symulacji podejmuje kluczowe decyzje, takie jak wybór aktywnych czujników, symulacja awarii i obliczanie statystyk.

-start_simulation(self): Uruchamia symulację, zmieniając stan i rozpoczynając cykliczne wywołania simulate_step.

-reset_simulation(self): Przywraca symulację do stanu początkowego, przywracając sieć czujników i zerując statystyki oraz losując punkty na nowo.

Pola:

-self.cfg: Przechowuje wczytaną konfigurację symulacji.

-self.sensors: Lista obiektów reprezentujących czujniki w sieci.

-self.sink: Obiekt reprezentujący węzeł odbiorczy na środku mapy.

-self.canvas: Płótno, na którym wyświetlana jest sieć.

6.2. logic.py

Metody:

-generate_static_points(cfg): Tworzy stałe punkty w przestrzeni, które czujniki mają za zadanie pokryć.

-generate_grid_sensors_auto(...): Generuje czujniki w równomiernym, siatkowym układzie.

-bfs_paths_to_sink(sink, sensors_with_energy): Wykorzystuje algorytm BFS (przeszukiwanie wszerz), aby znaleźć najkrótsze ścieżki od wszystkich aktywnych czujników do węzła sink.

-solve_ilp(points, sensors, cfg): To najważniejsza funkcja. Korzysta z programowania liniowego (pulp) do optymalizacji. Jej celem jest wybranie minimalnej liczby czujników, które zapewnią odpowiednie pokrycie punktów pomiarowych, jednocześnie biorąc pod uwagę ich energię.

-transmit_data_along_path(path, cfg): Symuluje przesyłanie danych wzdłuż wybranej ścieżki, odejmując energię od czujników i uwzględniając prawdopodobieństwo utraty pakietów.

6.3. config.py

Metody:

-load_config(filename="config.ini"): Główna funkcja, która używa wbudowanej biblioteki Pythona configparser. Otwiera plik konfiguracyjny (config.ini), odczytuje wartości z sekcji [PARAMS] i zwraca je w postaci słownika. Dzięki temu wszystkie parametry, takie jak szerokość pola, liczba czujników czy koszty energetyczne, są łatwo dostępne dla innych modułów.

-cfg: Globalna zmienna, która przechowuje wczytane dane konfiguracyjne. Jest ona importowana i używana w main.py i logic.py.

-OFFSET_X, OFFSET_Y: Stałe wartości używane do obliczania marginesów na płótnie, co zapewnia, że wizualizacja nie będzie zaczynać się w samym rogu.

6.4. models.py

klasy:

-MeasurementPoint: Bardzo prosta klasa. Reprezentuje statyczny punkt w przestrzeni, który ma być monitorowany. Posiada jedynie współrzędne x i z.

Sensor: To bardziej złożona klasa, która reprezentuje pojedynczy czujnik.

Pola:

-id: Unikalny identyfikator czujnika.

-x, z: Współrzędne czujnika w przestrzeni.

-energy: Kluczowy atrybut, który maleje w trakcie symulacji.

-is_on, is_failed: Atrybuty stanu, wskazujące, czy czujnik jest aktywny, czy też uległ awarii.

-neighbors: Lista sąsiednich czujników, z którymi może się komunikować.

-distance_to(other): Metoda obliczająca odległość od innego obiektu.

-covers(point, cfg): Metoda sprawdzająca, czy czujnik jest w stanie pokryć dany punkt, biorąc pod uwagę jego zasięg.

7. Spis znanych błędów i ograniczeń

- Środkowe sensory ulegają szybkiemu rozładowaniu na skutek intensywnej komunikacji sensorowej (tzn. hotspoty)
- Mało interaktywne GUI
- Pomimo, że był to punkt dodatkowy, nie udało mi się opanować tworzenia grafów na koniec
- Jeden algorytm w programie

8. Lista nietypowych zachowań systemu

- Przycisk Reset ustawia na nowo POI, mimo że konfiguracja się nie zmieniła (przydatne przy szukaniu „wygodnego” rozłożenia punktów, ale nie zamierzone)

9. Załączniki i materiały

img/– zrzuty ekranu instalacji (IDE, interpreter, konfiguracja), UI oraz wygenerowane wykresy.

docs/– wygenerowana dokumentacja pdoc.