



Projekt: Metody rozwiązywania problemu maksymalizacji czasu życia sieci
sensorowej- metody heurystyczne i metaheurystyczne (symulowane
wyżarzanie, algorytmy genetyczne)

Dawid Balikowski, nr albumu: 1233378



2 CZERWCA 2024

UKSW

Spis treści

1. Dokumentacja projektowa	2
Specyfikacja wymagań.....	2
Spis klas, metod i funkcji programu.....	2
Klasy.....	2
Funkcje.....	3
Diagram klas	3
Diagram sekwencji.....	4
2. Dokumentacja użytkownika.....	5
Instrukcja obsługi.....	5
Przykłady użycia.....	6
3. Plan testów	7
1. Testowanie Funkcji Pomocniczych.....	7
2. Testowanie Klasy Field	7
3. Testowanie Funkcji Głównych.....	7
4. Testowanie Interfejsu Użytkownika	8
5. Testy Integracyjne	8
6. Testy Wydajnościowe	8
7. Testy Użyteczności	8
4. Kod źródłowy	9

1. Dokumentacja projektowa:

Specyfikacja wymagań:

1. Program jest aplikacją działającą na systemach Windows, służącą do obliczenia maksymalnego czasu życia sieci sensorowej według niniejszych wymagań:
 - Obszar działania (pole), to kwadrat o wymiarach podanych przez użytkownika. Znajdują się na nim: sensory oraz targety (ich liczbę też podaje użytkownik)
 - Każdy sensor może być aktywny 60 sekund
 - W każdym momencie życia sieci sensorowej, każdy target musi być obserwowany przez co najmniej jeden sensor, jeśli nie jest, sieć jest uważana za martwą
 - Do obliczenia najoptymalniejszego czasu życia sieci użyte jest symulowanie wyżarzenia.
2. Wymagania funkcjonalne:
 - Użytkownik podaje dane wejściowe takie jak: rozmiar pola, liczba targetów, liczba celów, a także zasięg każdego z sensora (są one generowane na polu losowo)
 - Program oblicza najlepszy możliwy czas życia takiej sieci.
3. Wymagania niefunkcjonalne:
 - Interfejs graficzny
 - Wykresy rozlokowania sensorów wraz z ich zasięgami, targetów, i oznaczenie sensorów, które są od początku martwe (nie obserwują żadnego targetu)
 - Wykresy liczby aktywnych sensorów w danej sekundzie życia sieci.

Spis klas, metod i funkcji programu

Klasy

Sensor

__init__(self, x, y, range, status='active') - Inicjalizuje obiekt sensora z podanymi współrzędnymi x, y, zasięgiem oraz statusem (active lub dead).

Target

__init__(self, x, y) - Inicjalizuje obiekt targetu z podanymi współrzędnymi x, y.

Field

__init__(self, nSensors, nTargets, fieldSize, sensorRange) - Inicjalizuje obiekt pola z podaną liczbą sensorów, targetów, rozmiarem pola oraz zasięgiem sensora.

generateField(self) - Generuje pole z sensorami i targetami, ustawiając je początkowo w punkcie (0, 0).

raffleTheElements(self) - Rozmieszcza sensory i targety losowo na polu oraz ustawia status sensorów na 'dead', jeśli nie mogą monitorować żadnego targetu.

setSensorsDeads(self) - Ustawia sensory jako 'dead', jeśli nie mogą monitorować żadnego targetu.

numOfLiveSensors(self) - Zwraca liczbę aktywnych sensorów.

printField(self) - Wyświetla planszę z sensorami i targetami na wykresie.

Interface

__init__(self, master) - Inicjalizuje interfejs użytkownika, tworząc etykiety, pola tekstowe oraz przyciski.

validateAndGetValues(self) - Waliduje wprowadzone dane i uruchamia główny program z podanymi wartościami.

Funkcje

areAllTargetsMonitored(targets, activeSensors, sensorCoverage) - Sprawdza, czy wszystkie targety są monitorowane przez przynajmniej jeden sensor.

calculateDistance(point1, point2) - Oblicza dystans między dwoma punktami na mapie.

calculateMaxLifeTime(field) - Oblicza maksymalny czas życia sieci dla losowego harmonogramu włączania i wyłączania sensorów.

simulatedAnnealing(field) - Implementuje algorytm symulowanego wyżarzania w celu znalezienia najlepszego czasu życia sieci sensorowej.

showResult(value, coords) - Wyświetla wynikowy maksymalny czas życia sieci sensorowej oraz wykres liczby aktywnych sensorów w danej sekundzie.

runProgram(size, nTargets, nSensors, range) - Uruchamia główną logikę programu, generując pole, rozmieszczając elementy, obliczając maksymalny czas życia oraz wyświetlając wyniki.

main() - Inicjalizuje i uruchamia interfejs użytkownika.

Diagram klas:

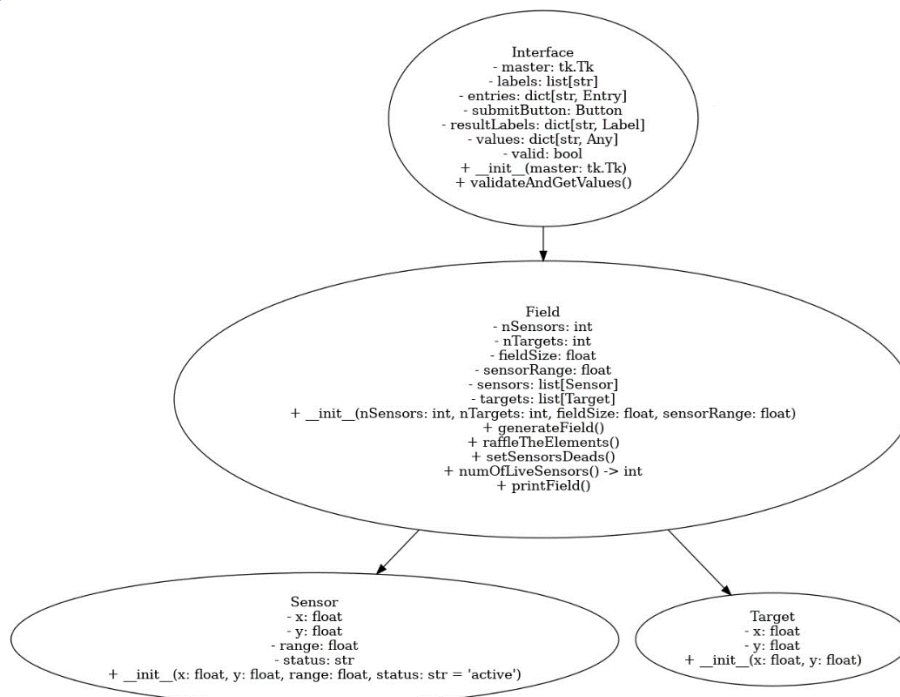
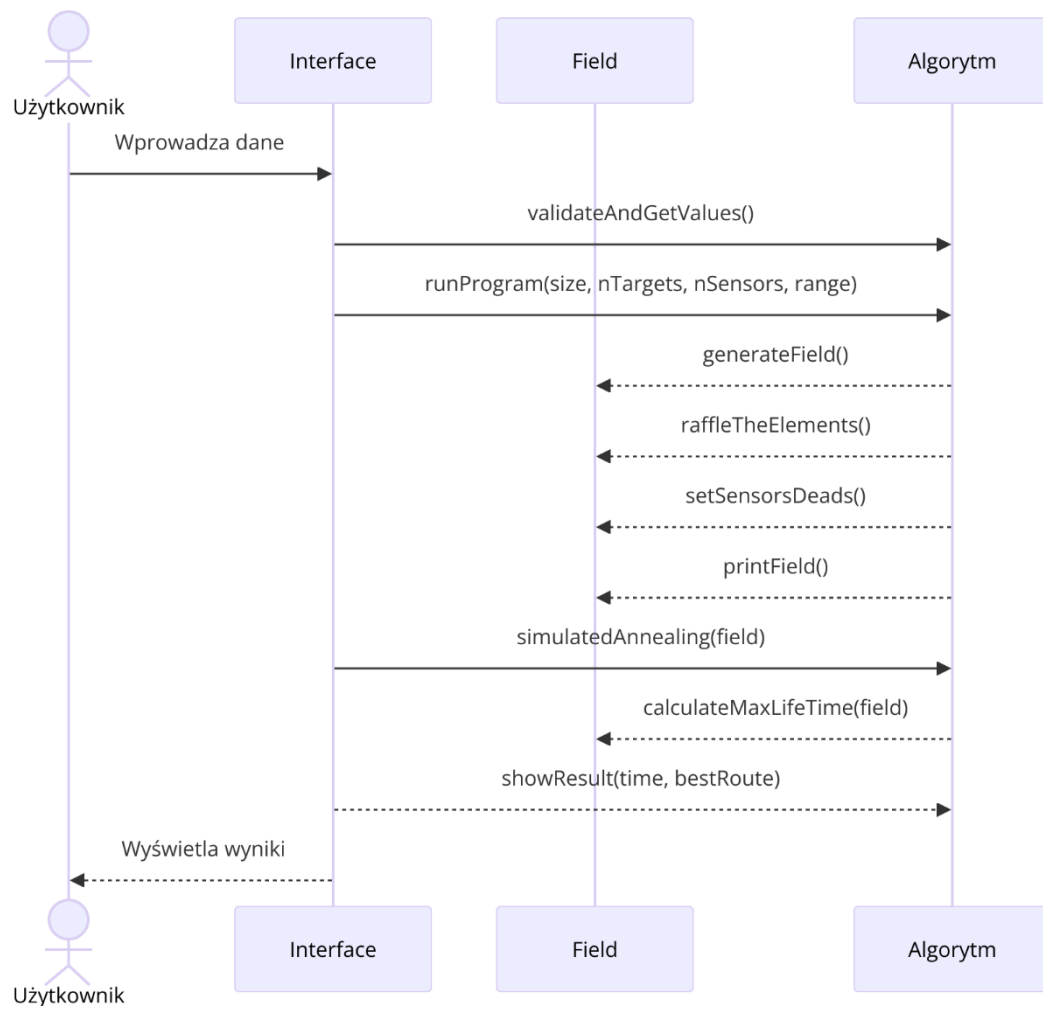


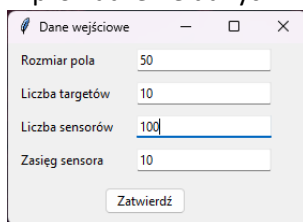
Diagram sekwencji:



2. Dokumentacja użytkownika

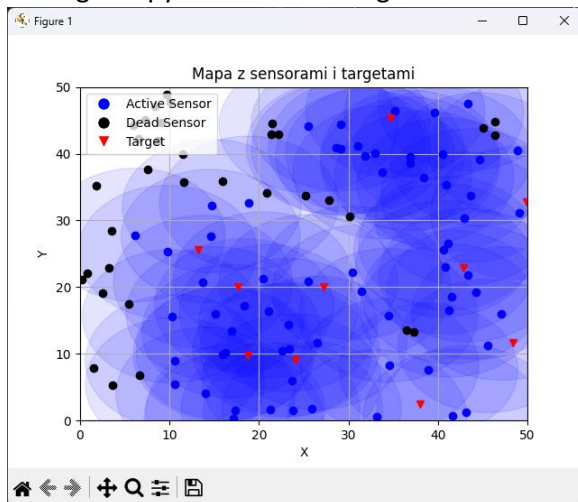
Instrukcja obsługi:

1. Wprowadzenie danych wejściowych w formularzu oraz zatwierdzenie.



Formularz "Dane wejściowe" zawiera cztery pola tekstowe: "Rozmiar pola" (wartość 50), "Liczba targetów" (wartość 10), "Liczba sensorów" (wartość 100) oraz "Zasięg sensora" (wartość 10). Na dole znajduje się przycisk "Zatwierdź".

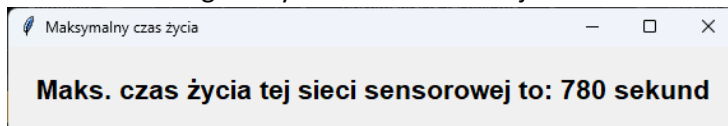
2. Obsługa mapy z sensorami i targetami:



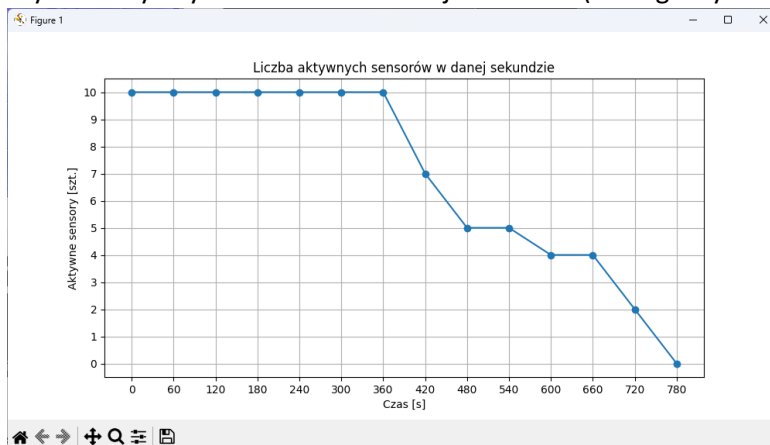
Dolny pasek: od lewej: pozycja wyjściowa, cofnięcie zmiany do tyłu, do przodu, przewijanie po mapie, przybliżenie, konfiguracja mapy, zapis mapy do pliku graficznego.

W celu wyświetlenia okna z czasem maksymalnym i wykresem należy wyłączyć okno z mapą

3. Komunikat o długości życia sieci sensorowej:



4. Wykres aktywnych sensorów w danej sekundzie (obsługa wykresu taka sama jak mapy)



Przykłady użycia:

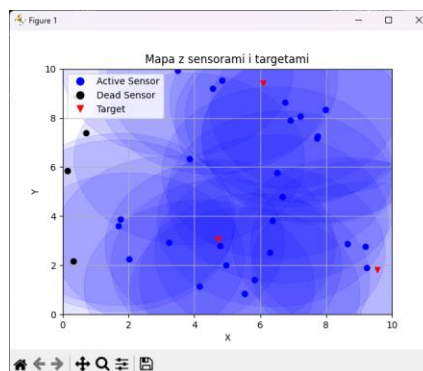
Przykład 1. Chcemy obliczyć czas życia sieci sensorowej, gdzie:

- Rozmiar pola: 10
- Liczba targetów: 3
- Liczba sensorów: 30
- Zasięg sensora: 3.5

Dane wejściowe

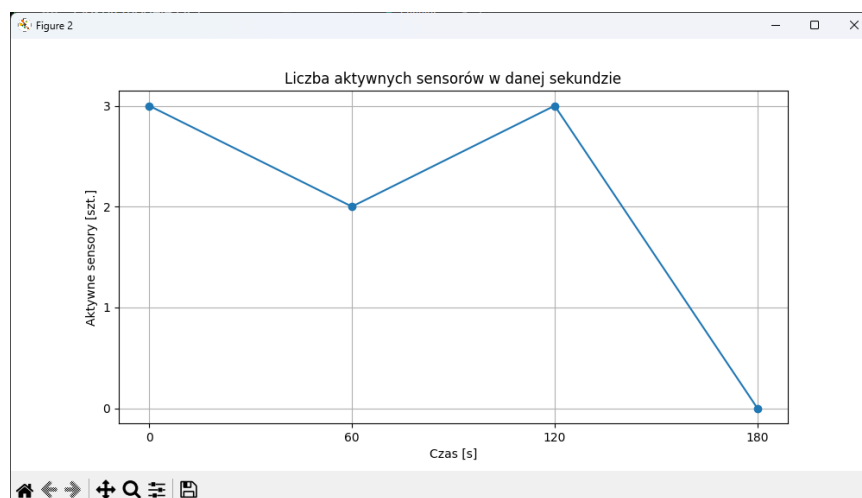
Rozmiar pola	10
Liczba targetów	3
Liczba sensorów	30
Zasięg sensora	3.5

Zatwierdź



Maksymalny czas życia

Maks. czas życia tej sieci sensorowej to: 180 sekund



3. Plan testów:

1. Testowanie Funkcji Pomocniczych

1.1 areAllTargetsMonitored(targets, activeSensors, sensorCoverage)

- Test 1.1.1: Sprawdzenie, czy wszystkie cele są monitorowane, gdy każdy cel jest pokryty przez co najmniej jeden aktywny sensor.
- Test 1.1.2: Sprawdzenie, gdy nie wszystkie cele są monitorowane (brak aktywnego sensora pokrywającego niektóre cele).
- Test 1.1.3: Sprawdzenie, gdy lista aktywnych sensorów jest pusta.

1.2 calculateDistance(point1, point2)

- Test 1.2.1: Sprawdzenie poprawności obliczeń dla różnych punktów (np. (0,0) i (3,4)).
- Test 1.2.2: Sprawdzenie, czy funkcja zwraca zero, gdy oba punkty są takie same.

2. Testowanie Klasy Field

2.1 generateField()

- Test 2.1.1: Sprawdzenie, czy generowane są odpowiednie liczby sensorów i celów.
- Test 2.1.2: Sprawdzenie, czy wszystkie generowane sensory i cele mają początkowe współrzędne (0,0).

2.2 raffleTheElements()

- Test 2.2.1: Sprawdzenie, czy współrzędne sensorów i celów są losowane prawidłowo w granicach pola.
- Test 2.2.2: Sprawdzenie, czy współrzędne są różne od (0,0) po losowaniu.

2.3 setSensorsDeaths()

- Test 2.3.1: Sprawdzenie, czy sensory, które nie mogą monitorować żadnych celów, mają status 'dead'.
- Test 2.3.2: Sprawdzenie, czy sensory, które mogą monitorować przynajmniej jeden cel, mają status 'active'.

2.4 numOfLiveSensors()

- Test 2.4.1: Sprawdzenie, czy funkcja zwraca poprawną liczbę aktywnych sensorów.

2.5 printField()

- Test 2.5.1: Sprawdzenie, czy wykres jest generowany i wyświetlany prawidłowo.

3. Testowanie Funkcji Głównych

3.1 calculateMaxLifeTime(field)

- Test 3.1.1: Sprawdzenie, czy funkcja prawidłowo oblicza maksymalny czas życia sieci.
- Test 3.1.2: Sprawdzenie, czy funkcja prawidłowo tworzy i aktualizuje dziennik aktywności sensorów.

3.2 simulatedAnnealing(field)

- Test 3.2.1: Sprawdzenie, czy funkcja poprawnie implementuje algorytm symulowanego wyżarzania.
- Test 3.2.2: Sprawdzenie, czy funkcja poprawnie znajduje lepsze rozwiązanie w trakcie symulacji.

4. Testowanie Interfejsu Użytkownika

4.1 Interface.__init__(self, master)

- Test 4.1.1: Sprawdzenie, czy wszystkie elementy interfejsu (pola tekstowe, etykiety, przyciski) są poprawnie inicjalizowane.

4.2 validateAndGetValues()

- Test 4.2.1: Sprawdzenie, czy poprawne dane wejściowe są prawidłowo przetwarzane.
- Test 4.2.2: Sprawdzenie, czy błędne dane wejściowe (np. litery zamiast liczb) generują odpowiednie komunikaty o błędach.

4.3 showResult(value, coords)

- Test 4.3.1: Sprawdzenie, czy poprawny wynik jest wyświetlany prawidłowo w nowym oknie.
- Test 4.3.2: Sprawdzenie, czy wykres liczby aktywnych sensorów w czasie jest poprawnie generowany i wyświetlany.

5. Testy Integracyjne

- Test 5.1: Pełna integracja: wprowadzenie danych przez użytkownika, generowanie pola, losowanie elementów, obliczanie maksymalnego czasu życia sieci i wyświetlanie wyników.
- Test 5.2: Sprawdzenie, czy wszystkie komponenty współpracują prawidłowo i czy cały proces działa bez błędów.

6. Testy Wydajnościowe

- Test 6.1: Sprawdzenie wydajności aplikacji dla dużej liczby sensorów i celów.
- Test 6.2: Sprawdzenie czasu odpowiedzi interfejsu użytkownika przy dużym obciążeniu.

7. Testy Użyteczności

- Test 7.1: Sprawdzenie, czy interfejs użytkownika jest intuicyjny i łatwy w użyciu.
- Test 7.2: Sprawdzenie, czy komunikaty o błędach są czytelne i zrozumiałe dla użytkownika.

4. Kod źródłowy:

main.py:

```
import random
import math
import matplotlib.pyplot as plt
import tkinter as tk
from tkinter import ttk, messagebox

BATTERY_TIME = 60 # czas zycia baterii

# funkcja, która sprawdza, czy wszystkie targety są monitorowane przez przynajmniej jeden sensor
def areAllTargetsMonitored(targets, activeSensors, sensorCoverage):
    monitoredTargets = set()
    for sensor in activeSensors:
        for target in sensorCoverage[sensor]:
            monitoredTargets.add(target)
    return all(target in monitoredTargets for target in targets)

# funkcja obliczająca dystans na mapie między dwoma elementami
def calculateDistance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    return distance

# obliczanie maksymalnego czasu zycia sieci dla losowego harmonogramu
def calculateMaxLifeTime(field):
    lifetime = 0
    targets = field.targets
    sensors = []
    for sensor in field.sensors:
        if sensor.status == 'active':
            sensors.append(sensor)

    # tworzymy pustą listę do przechowywania przebiegu włączania i wyłączania sensorów
    sensorsActivityLog = []

    # Definicja zasięgu poszczególnych sensorów (które targety są przez nie monitorowane)
    sensorCoverage = {}

    for sensor in sensors:
        coverage = []
        for target in targets:
            if calculateDistance((sensor.x, sensor.y), (target.x, target.y)) < sensor.range:
                coverage.append(target)
        sensorCoverage[sensor] = coverage
```

```

# Lista dostępnych sensorów (które jeszcze nie zostały wyłączone)
availableSensors = list(sensors)

# Pętla symulująca losowe włączanie i wyłączanie sensorów
while availableSensors:
    activeSensors = []

    for sensor in availableSensors:
        if random.choice([0,0,1]):
            activeSensors.append(sensor)
        if len(activeSensors) == field.nTargets:
            break

    # Sprawdzamy, czy wszystkie targety są monitorowane
    if areAllTargetsMonitored(targets, activeSensors, sensorCoverage):
        # Zapisujemy stan sensorów do listy
        sensorsActivityLog.append(list(activeSensors))
        # Usuwamy użyte sensory z listy dostępnych sensorów
        for sensor in activeSensors:
            availableSensors.remove(sensor)
    # Jeśli nie uda się monitorować wszystkich targetów, pętla może się zakończyć
    else:
        break

stepsWithCoords = {} # przebieg symulacji

# Wyświetlamy wynikową listę aktywności sensorów wraz z ich koordynatami
for step, activeSensors in enumerate(sensorsActivityLog):
    lifetime += BATTERY_TIME

for step, activeSensors in enumerate(sensorsActivityLog):
    temp = []
    for sensor in activeSensors:
        temp.append((sensor.x,sensor.y))
    stepsWithCoords[step] = temp

return lifetime, stepsWithCoords

# obliczanie najlepszego czasu
def simulatedAnnealing(field):

    temp = 1000
    iterations = 5000
    coolingRate = 0.9
    bestTime, bestRoute = calculateMaxLifeTime(field)

    for _ in range(0,iterations):

```

```

    newTime, newRoute = calculateMaxLifeTime(field)
    if newTime > bestTime:
        bestTime = newTime
        bestRoute = newRoute
    elif random.uniform(0,1) < math.exp((newTime-bestTime)/temp):
        bestTime = newTime
        bestRoute = newRoute
    temp *= coolingRate

    return bestTime, bestRoute

# klasa reprezentująca pojeźdźczy sensor
class Sensor:
    def __init__(self, x, y, range, status='active'):
        self.x = x
        self.y = y
        self.range = range
        self.status = status

# klasa reprezentująca pojedynczy taget
class Target:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# klasa reprezentująca pole z sensorami i targetami
class Field:
    def __init__(self):
        self.nSensors = ""
        self.nTargets = ""
        self.fieldSize = ""
        self.sensorRange = ""

    def __init__(self, nSensors, nTargets, fieldSize, sensorRange):
        self.nSensors = int(nSensors)
        self.nTargets = int(nTargets)
        self.fieldSize = float(fieldSize)
        self.sensorRange = float(sensorRange)

# generuje pole z sensorami i targetami (wszystkie te elementy są ustawione w punkcie (0,0))
def generateField(self):
    self.sensors = [Sensor(0, 0, self.sensorRange) for _ in range(self.nSensors)]
    self.targets = [Target(0, 0) for _ in range(self.nTargets)]

# rozlokowanie elementów na planszy w losowym położeniu
def raffleTheElements(self):
    for sensor in self.sensors:
        sensor.x = random.uniform(0, self.fieldSize)
        sensor.y = random.uniform(0, self.fieldSize)

```

```

for target in self.targets:
    target.x = random.uniform(0, self.fieldSize)
    target.y = random.uniform(0, self.fieldSize)
self.setSensorsDeads()

# sprawdza czy sensory mogą monitorować jakikolwiek target, jeśli nie umierają
def setSensorsDeads(self):
    for sensor in self.sensors:
        n = 0
        for target in self.targets:
            if calculateDistance((sensor.x, sensor.y), (target.x, target.y)) > sensor.range:
                n += 1
        if n == len(self.targets):
            sensor.status = 'dead'

# zwraca liczbę aktywnych sensorów
def numOfLiveSensors(self):
    n = 0
    for sensor in self.sensors:
        if sensor.status == 'active':
            n += 1
    return n

# wyświetla na ekranie planszę
def printField(self):
    fig, ax = plt.subplots()
    ax.set_xlim(0, self.fieldSize)
    ax.set_ylim(0, self.fieldSize)

    # wyświetla sensory
    for sensor in self.sensors:
        if sensor.status == 'active':
            ax.add_patch(plt.Circle((sensor.x, sensor.y), sensor.range, color='blue', alpha=0.1))
            ax.plot(sensor.x, sensor.y, 'bo')
        elif sensor.status == 'dead':
            ax.plot(sensor.x, sensor.y, 'ko')
            # ax.add_patch(plt.Circle((sensor.x, sensor.y), sensor.range, color='black', alpha=0.1))

    # wyświetla targety
    for target in self.targets:
        ax.plot(target.x, target.y, 'rv')

    # Tworzy legendę
    sensorPatch = plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='b', markersize=10,
label='Active Sensor')
    deadSensorPatch = plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='k', markersize=10,
label='Dead Sensor')

```

```

    targetPatch = plt.Line2D([0], [0], marker='v', color='w', markerfacecolor='r', markersize=10,
label='Target')
    ax.legend(handles=[sensorPatch, deadSensorPatch, targetPatch])

    ax.set_title('Mapa z sensorami i targetami')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    plt.grid(True)
    plt.show()

# pobierane dane wejsciowe od uzytkownika
class Interface:
    def __init__(self, master):
        self.master = master
        master.title("Dane wejściowe")

        # Etykiety i pola tekstowe
        self.labels = ["Rozmiar pola", "Liczba targetów", "Liczba sensorów", "Zasięg sensora"]
        self.entries = {}

        for idx, label in enumerate(self.labels):
            ttk.Label(master, text=label).grid(row=idx, column=0, padx=10, pady=5, sticky=tk.W)
            entry = ttk.Entry(master)
            entry.grid(row=idx, column=1, padx=10, pady=5)
            self.entries[label] = entry

        # Przycisk "Zatwierdź"
        self.submitButton = ttk.Button(master, text="Zatwierdź", command=self.validateAndGetValues)
        self.submitButton.grid(row=len(self.labels), column=0, columnspan=2, pady=10)

        # Etykiety do wyświetlania wyników
        self.resultLabels = {}
        for idx, label in enumerate(self.labels):
            resultLabel = ttk.Label(master, text="")
            resultLabel.grid(row=idx, column=2, padx=10, pady=5)
            self.resultLabels[label] = resultLabel

        # Zmienna przechowująca wartości formularza
        self.values = {}
        self.valid = False

    def validateAndGetValues(self):
        try:
            self.values["Rozmiar pola"] = float(self.entries["Rozmiar pola"].get())
            self.values["Liczba targetów"] = int(self.entries["Liczba targetów"].get())
            self.values["Liczba sensorów"] = int(self.entries["Liczba sensorów"].get())
            self.values["Zasięg sensora"] = float(self.entries["Zasięg sensora"].get())

            # self.values #

```

```

self.valid = True

#uruchamienie właściwego programu
runProgram(self.values["Rozmiar pola"],self.values["Liczba targetów"],self.values["Liczba
sensorów"],self.values["Zasięg sensora"])

except ValueError as e:
    messagebox.showerror("Błąd", f"Nieprawidłowa wartość: {e}")
    self.valid = False

def showResult(value,coords):
    root = tk.Tk()
    root.title("Maksymalny czas życia")

    # Tworzenie etykiety z podaną wartością
    label = ttk.Label(root, text=f"Maks. czas życia tej sieci sensorowej to: {value} sekund")
    label.config(font=("Baskerville", 16, "bold"))
    label.pack(padx=20, pady=20)

    xValues = list()
    yValues = list()

    for key, value in coords.items():
        # print(f"{key} + {len(value)}")
        xValues.append(key*60)
        yValues.append(len(value))

    xValues.append((key+1)*60)
    yValues.append(0)

    # Tworzenie wykresu
    plt.figure(figsize=(10, 5))
    plt.plot(xValues, yValues, marker='o')

    # Dodanie tytułu i etykiet osi
    plt.title("Liczba aktywnych sensorów w danej sekundzie")
    plt.xlabel("Czas [s]")
    plt.ylabel("Aktywne sensory [szt.]")

    plt.xticks(range(min(xValues), max(xValues) + 1, 60))
    plt.yticks(range(min(yValues), max(yValues) + 1, 1))

    # Dodanie siatki dla lepszej czytelności

```

```

plt.grid(True)

# Wyświetlenie wykresu
plt.show()

# Uruchomienie głównej pętli aplikacji Tkinter
root.mainloop()

def runProgram(size,nTargets,nSensors,range):
    board = Field(nSensors,nTargets,size,range)
    board.generateField()
    board.raffleTheElements()
    board.printField()
    time, bestRoute = simulatedAnnealing(board)
    showResult(time, bestRoute)
    for key, value in bestRoute.items():
        print(f"Key: {key}, Value: {value}")

def main():
    root = tk.Tk()
    app = Interface(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```