Projekt Optymalizacja Wielokryterialna

Realizacja SWD implementującego metody optymalizacji wielokryterialnej

Autorzy:

Julia Nowak

Adam Złocki

Jakub Szczypek

Spis treści

Podział zadań

Członek zespołu	Zadania
Julia Nowak	• Przygotowanie GUI • Dokumentacja – wstęp i przygotowanie danych • Wstępna wersja kodu PSO
Adam Złocki	• Poprawki zaimplementowanego już algorytmu PSO • Implementacja drugiego algorytmu (jeszcze niezdefiniowany) • Praca nad dokumentacją
Jakub Szczypek	• Praca nad zaimplementowanym już algorytmem PSO – wprowadzanie poprawek i rozwijanie aktualnej wersji algorytmu • Praca nad dokumentacją

Wstęp

Problem badawczy

Tematem projektu jest **optymalizacja przejazdu zespołu robotów przez przeszkodę**, co stanowi przykład problemu wielokryterialnego w optymalizacji. Problem ten dotyczy znalezienia kompromisu między dwoma sprzecznymi kryteriami: minimalizacją uszkodzeń robotów a minimalizacją czasu potrzebnego na pokonanie przeszkody przez cały zespół.

Kluczowe założenia problemu to:

- **Zakłócenia w ruchu robotów** na przeszkodzie mogą prowadzić do ich uszkodzeń, a ryzyko to rośnie w przypadku wysokiej gęstości robotów w jednym obszarze.
- **Dystans między robotami** wpływa na poziom zakłóceń i ryzyko uszkodzeń – większy dystans zmniejsza ryzyko, ale zwiększa czas przejazdu.

Głównym celem badawczym jest zaprojektowanie i wdrożenie efektywnego algorytmu wielokryterialnej optymalizacji, który uwzględnia oba te kryteria.

Opis wykorzystywanej bazy danych / zbioru decyzyjnego

W projekcie do symulacji i analizy danych planujemy wykorzystać różne źródła danych:

1. **Generowane dane testowe**
   Przygotujemy sztuczne dane reprezentujące różne konfiguracje przeszkód i zakłóceń. Dane te będą obejmować:
   - Mapy 3D z różnorodnymi układami przeszkód (np. linie proste, labirynty, przeszkody dynamiczne).
   - Zespoły robotów o różnej liczebności i właściwościach.

2. **Dane z literatury naukowej**
   Skorzystamy z prac dotyczących swarm robotics oraz optymalizacji trajektorii, które dostarczają przykładów i modeli problemów podobnych do analizowanego w naszym projekcie.

3. **Publiczne bazy symulacyjne**
   Wykorzystamy otwarte źródła, takie jak:
   - Open Robotics,
   - ROS (Robot Operating System),
   - Bazy danych i środowiska symulacyjne dostępne w repozytoriach naukowych, np. artykuły z *ScienceDirect*.

   Dzięki połączeniu danych generowanych, literaturowych oraz z publicznych baz symulacyjnych, zapewnimy szeroki kontekst i realistyczne warunki do optymalizacji oraz weryfikacji opracowanego algorytmu.

Przykładowymi źródłami danych mogą być:

- https://library.fiveable.me/swarm-intelligence-and-robotics
- https://en.wikipedia.org/wiki/Particle_swarm_optimization
- https://www.sciencedirect.com/science/article/pii/S0304389424004114

Przygotowanie i weryfikacja danych

Opis ogólny

Do przygotowania mapy terenu, która jest główną informacją wejściową naszego algorytmu, przygotowaliśmy funkcję generującą mapę wysokościową terenu. Proces generowania mapy terenu służy do symulacji różnych typów wysokości terenu w formie macierzy danych, które mogą być wizualizowane w postaci mapy wysokości. Funkcja umożliwia generowanie różnych struktur terenu, takich jak pagórki, linie, nachylenia, kaniony, wzory przypominające labirynty oraz inne nieregularne kształty.

Parametry

Funkcja posiada następujące argumenty wejściowe:

1. **noise_num** (liczba, domyślnie 1)

   Określa intensywność losowych zakłóceń (szumu) dodanych do mapy terenu. Większe wartości generują bardziej chaotyczne powierzchnie.

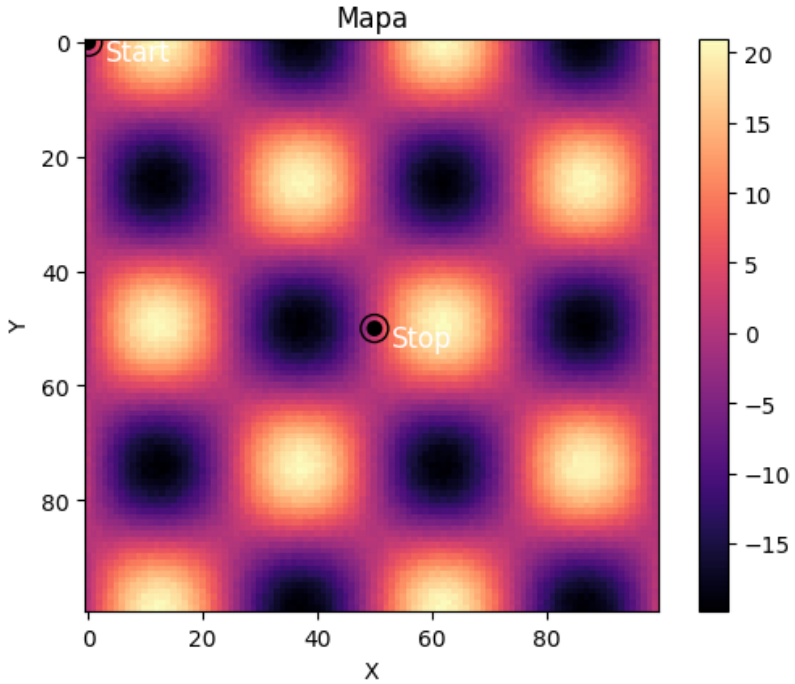
2. **terrain_size** (krotka, domyślnie (100, 100))

   Rozmiar generowanego terenu w postaci liczby punktów w osiach X i Y. Na przykład (100, 100) oznacza macierz 100x100 pikseli.

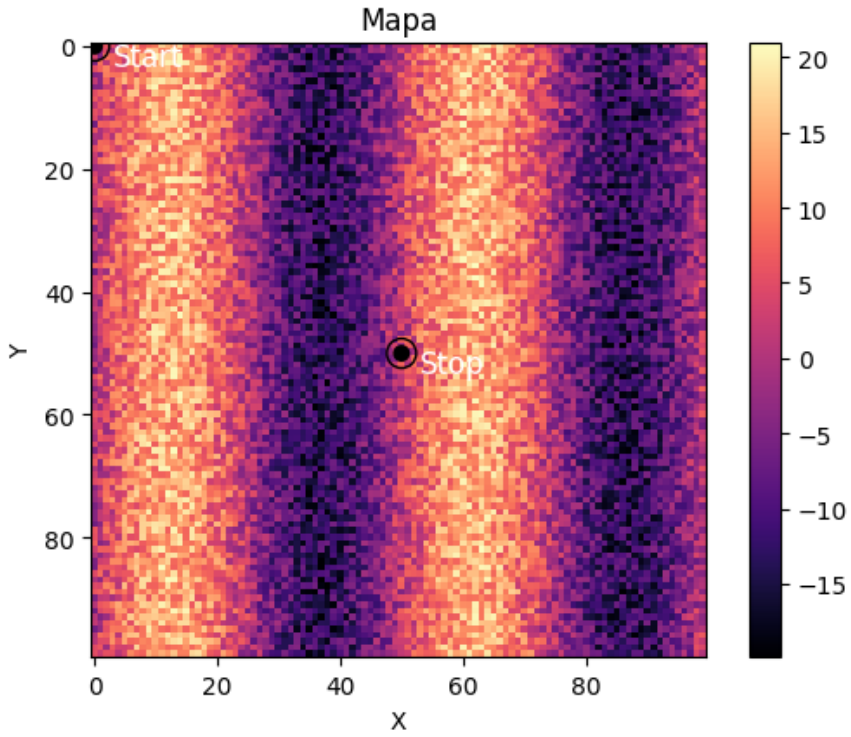
3. **terrain_type** (łańcuch znaków, domyślnie "hills")

   Typ struktury terenu, który ma zostać wygenerowany. Dostępne tryby:

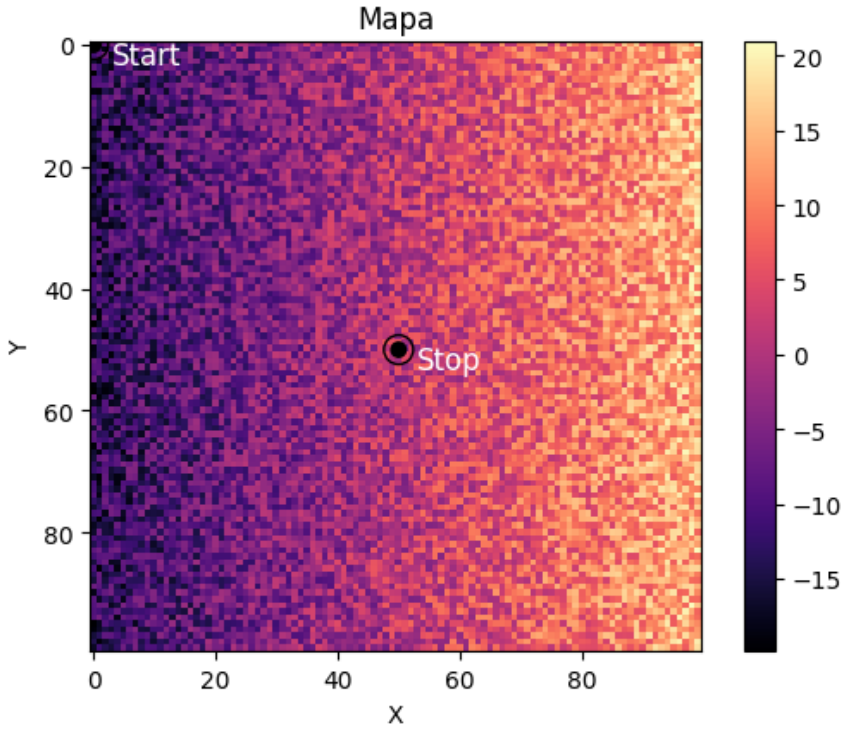
   - **"hills":** Pagórkowata powierzchnia oparta na sinusoidalnych wzorach.



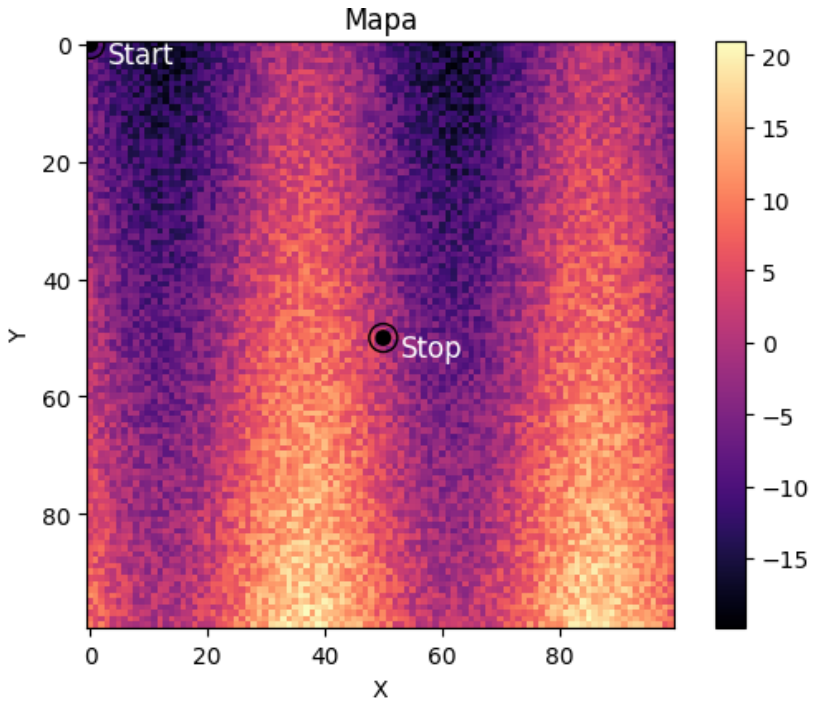
- **"lines":** Powierzchnia z liniowymi wzorami wzdłuż osi X.



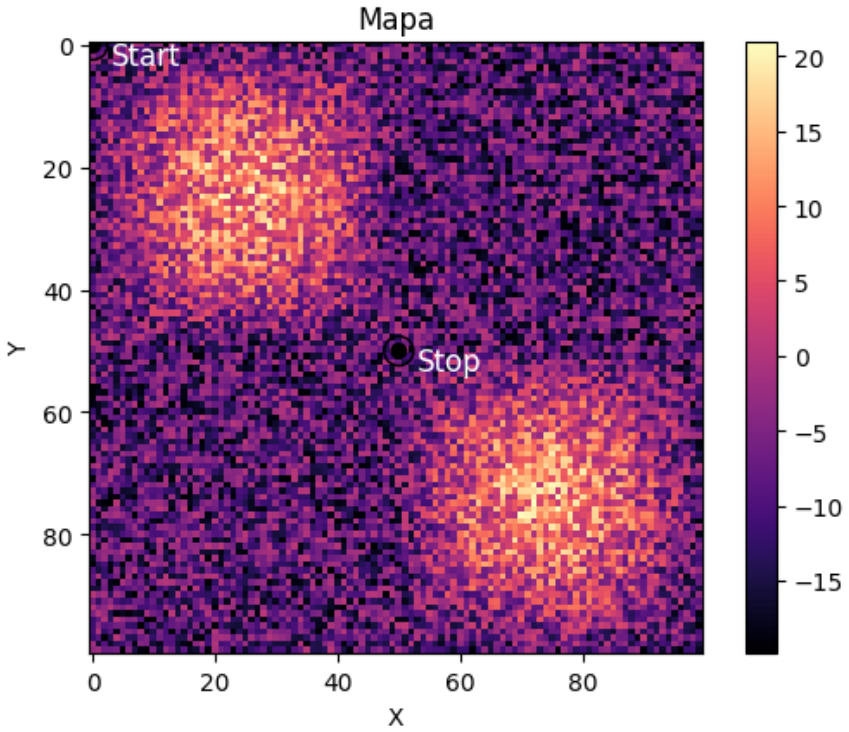
- **"slope":** Jednostajny spadek wzdłuż osi X.



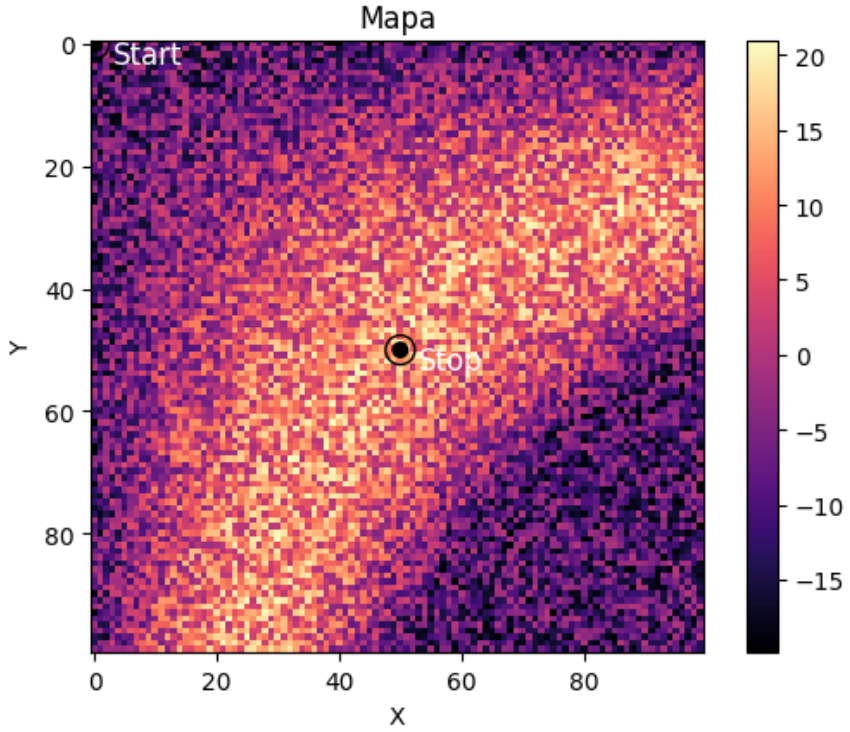
- **"razors":** Powierzchnia z ostrymi przejściami wzdłuż osi X i Y.



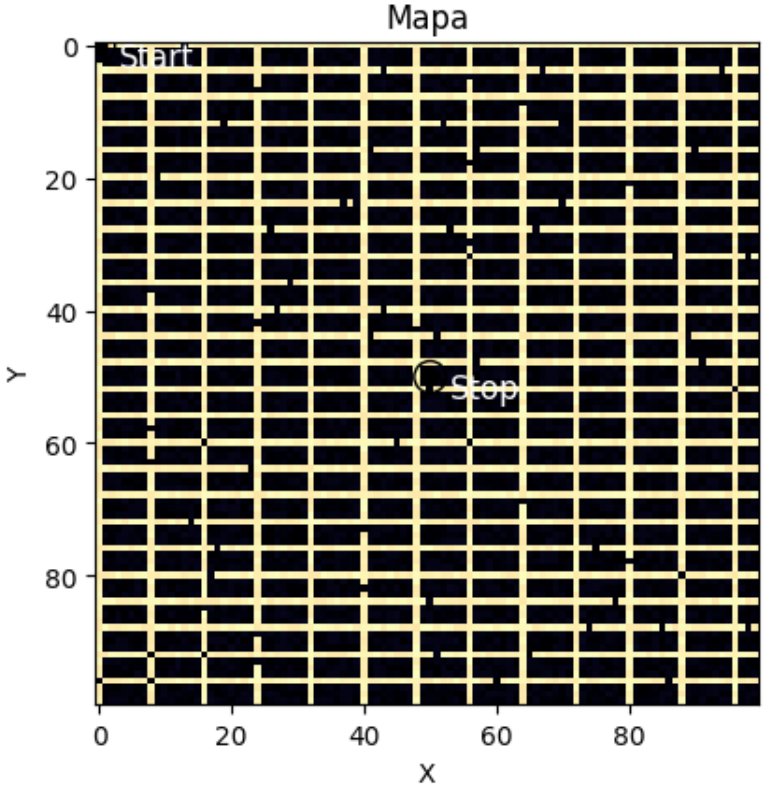
- **"canyon":** Powierzchnia przypominająca kanion z dodatkowymi dolinami.



- **"bow":** Powierzchnia w kształcie łuku oparta na iloczynie sinusoidalnym.



- **"maze":** Wzór przypominający labirynt, zawierający ściany oraz przypadkowe otwarcia w ścianach.



Działanie funkcji

Na początku generowane są losowe wartości w postaci macierzy o rozmiarze określonym przez terrain_size. Te wartości reprezentują losowy szum, który będzie bazą dla struktury terenu. Następnie generowana jest siatka współrzędnych dla osi X i Y, umożliwiająca modelowanie wzorów sinusoidalnych lub liniowych dla wybranej struktury terenu.

Na podstawie wartości terrain_type tworzony jest odpowiedni wzór, np. pagórki, linie lub nachylenia lub generowany jest labirynt. Wygenerowana struktura jest łączona z zakłóceniami w celu stworzenia ostatecznej mapy terenu.

Wybór rozmiaru macierzy (terrain_size) wpływa na poziom szczegółowości wygenerowanej mapy. Większe rozmiary mogą zwiększyć dokładność, ale wydłużają czas obliczeń. Intensywność zakłóceń (noise_num) może być regulowana w celu uzyskania bardziej realistycznych lub abstrakcyjnych wyników.

Implementacje metody wielokryterialnej

CSO (Cockroach Swarm Optimization) z adaptacją do problemów wielokryterialnych

Algorytm Rojowy Karalucha CSO (Cockroach Swarm Optimization) jest metodą optymalizacji inspirowaną zachowaniami karaluchów, które skutecznie umożliwiają eksplorację przestrzeni rozwiązań w poszukiwaniu optymalnych wartości. Wykorzystuje on kilka różnych zachowań charakterystycznych dla karaluchów, takich jak poruszanie się w rojach oraz rozpraszanie się w momencie wystąpienia bodźców, aby zapewnić ciągłe przemieszczanie się osobników i uniknąć utknięcia w lokalnych ekstremach. Stanowi skuteczne narzędzie optymalizacji, które wykorzystuje zróżnicowane zachowania inspirowane naturą, aby przeszukiwać przestrzeń rozwiązań i znajdować optymalne wartości.

Algorytm CSO jest najczęściej wykorzystywany do problemów optymalizacji jednokryterialnej, jednak możliwe jest zastosowanie modyfikacji, które umożliwiają jego adaptację do problemów wielokryterialnych. Głównej modyfikacji ulega wówczas funkcja kosztu – wagowo możemy zdefiniować w jakim stopniu dane kryterium wpływa na ostateczny wynik działania algorytmu (Weighted Sum Method). Jest to w pewnym sensie „spłaszczenie" problemu N-kryterialnego do problemu optymalizacji funkcji kosztu. Jest to jedno z możliwych rozwiązań, jednak kosztowne obliczeniowo.

1) Parametry algorytmu

W przypadku CSO, głównymi parametrami są:

- Maksymalna liczba iteracji,
- Rozmiar populacji początkowej,
- Prawdopodobieństwo dyspersji,
- Widzialność (liczba kroków),
- Wielkość kroku.

2) Główna pętla algorytmu

Praca algorytmu karalucha zawiera się w głównej pętli wykonywanej do momentu osiągnięcia kryterium stopu, przyjętego jako maksymalna liczba iteracji. Po zainicjalizowaniu algorytmu parametrami początkowymi oraz wygenerowaniu populacji początkowej algorytm przechodzi do głównej pętli. Jest ona wykonywana do momentu przekroczenia

maksymalnej liczby iteracji. Można w niej wyróżnić cztery główne etapy, z których każdy został szerzej opisany w kolejnych podpunktach:

1. Wyznaczenie P_i oraz P_g
2. Procedura „Podążanie w roju”
3. Rozpraszanie
4. Bezwzględne zachowanie

Po zakończeniu pracy, algorytm zwraca:

- Najlepsze rozwiązanie,
- Wartość funkcji kosztu dla najlepszego rozwiązania,
- Listę najlepszych rozwiązań (po jednym z każdej iteracji),
- Listę wartości kosztu najlepszych rozwiązań (po jednym z każdej iteracji).

Prace całego algorytmu oparto na pseudokodzie pokazanym na Rys. 1. Uwzględnia on inercję osobnika **w**. Jednakże, w implementacji będącej obiektem tej pracy z niej zrezygnowano, co równało się przyjęciu jej wartości równej 1.

```
INPUT: Fitness function: f(x), x ∈ R^D
set parameters and generate an initial population of cockroach
set p_g = x_1
for i = 2 to N do
  if f(x_i) < f(p_g) then
    p_g = x_i
  end if
end for
for t = 1 to T_max do
  for i = 1 to N do
    for j = 1 to N do
      if abs(x_i − x_j) < visual; f(x_j) < f(x_i) then
        p_i = x_i
      end if
    end for
    if p_i == x_i then
      x_i = w · x_i + step · rand · (p_g − x_i)
    else
      x_i = w · x_i + step · rand · (p_i − x_i)
    end if
    if f(x_i) < f(p_g) then
      p_g = x_i
    end if
  end for
  for i = 1 to N do
    x_i = x_i + rand(1, D)
    if f(x_i) < f(p_g) then
      p_g = x_i
    end if
  end for
  k = randint([1, N])
  x_k = p_g;
end for
Check termination condition
```

Rys. 1 Pseudokod algorytmu karalucha bez mechanizmu głodu (opracowano na podstawie https://www.researchgate.net/figure/An-improved-cockroach-swarm-optimization-algorithm_fig1_263354974)

3) Zmienne P_i i P_g

W tym kroku każdy karaluch (rozwiązanie) X_i zostaje porównany ze wszystkimi karaluchami X_j będącymi w jego zakresie widzialności, który określa parametr *visual*. Rozwiązania są porównywane na podstawie przypisanego do każdego z nich czasu przejazdu wszystkich pojazdów przez wyznaczone wierzchołki, rozpoczynając i kończąc w stacji początkowej. Zmiennej P_i zostaje przypisany ten osobnik, który jest lepszy, czyli ten o mniejszej wartości czasu rozwiązania.

Po przeprowadzeniu porównania ze wszystkimi karaluchami X_j w zasięgu widzialności

karalucha X_i, zostaje wyznaczone minimum lokalne, które jest przypisane do zmiennej P_i. Globalnie najlepszy osobnik P_g zostaje wybrany z całej przestrzeni rozwiązań i nie wykonuje kroku w żadną stronę.

4) Procedura "Podążanie w Roju"

Jeżeli karaluch X_i okazał się lokalnie najsilniejszy, wówczas podąża za globalnie najsilniejszym karaluchem P_g według zależności:

$$X_i = X_i + step * rand < 0{,}1 > * (P_g - X_i)$$

W przeciwnym przypadku, karaluch podąża za lokalnie najsilniejszym osobnikiem P_i, zgodnie z następującą zależnością:

$$X_i = X_i + step * rand < 0{,}1 > * (P_i - X_i)$$

Podążanie samo w sobie polega na wykonywaniu kroków (*step*) przez karalucha X_i w stronę najsilniejszego dla niego karalucha X_j, który jest lokalnie/globalnie najsilniejszym osobnikiem. Każdy pojedynczy krok równoznaczny jest z pojedynczą modyfikacją X_i w kierunku osiągnięcia X_j. Na podstawie liczby takich przejść określana jest odległość pomiędzy X_i i X_j oraz to, czy karaluchy znajdują się w swoich zakresach widzialności.

5) Rozpraszanie (dyspersja)

Jest to operacja przeprowadzana na pojedynczym karaluchu według zależności:

$$X_i = X_i + step * rand(1, D)$$

Dyspersja to niewielka modyfikacja rozwiązania, która polega na poruszeniu się karalucha w kierunku innego osobnika wybranego losowo z całej przestrzeni rozwiązań. Zostało to zrealizowane poprzez wymianę jednego elementu z listy wierzchołów odwiedzanych przez jeden z samochodów, będący częścią rozwiązania. Element wybierany jest losowo, a następnie zostaje przeniesiony do innego, również losowo wybranego samochodu na losowe miejsce. Jeśli wybrany element został już wcześniej przeniesiony, wówczas zostaje wybrany inny, losowy element z danego samochodu. Ponadto, jeśli wybrany samochód ma

tylko jeden element, to operacja przeniesienia elementu nie zostaje zrealizowana i następuje przejście do kolejnego samochodu.

6) **Bezwzględne zachowanie**

Karaluchy słabsze są “pożerane” przez najsilniejszego karalucha P_g. Zostało to zrealizowane w taki sposób, że losowy karaluch, który jest gorszy od globalnie najlepszego karalucha, jest przez niego zastępowany, co odpowiada „pożeraniu”.

Komiwojażer dostosowany do problemu 3D z optymalizacją wag grafu

Problem komiwojażera (ang. Traveling Salesman Problem, TSP) jest jednym z klasycznych problemów optymalizacyjnych w teorii grafów. Polega na znalezieniu najkrótszej możliwej trasy, która odwiedza każdy wierzchołek grafu dokładnie raz i wraca do punktu początkowego. W przypadku rozszerzenia problemu na przestrzeń 3D, do klasycznych odległości między wierzchołkami dochodzi dodatkowy wymiar w postaci różnic wysokości. Algorytm musi uwzględniać trudność pokonywania wzniesień i zejść, co wymaga optymalizacji wag grafu.

Punkty na mapie są traktowane jako wierzchołki grafu, a krawędzie łączące te wierzchołki posiadają wagi obliczane według odpowiedniego wzoru, uwzględniającego pozycję robota i wysokość terenu (trudność poruszania się).

Przykładowy pseudokod rozwiązania przedstawiono poniżej.

Funkcja: MultiCriteriaTSP(G, K1, K2) Wejście: G - graf, gdzie wierzchołki reprezentują miasta, a krawędzie reprezentują połączenia z przypisanymi kosztami K1 - funkcja oceny pierwszego kryterium (np. koszt) K2 - funkcja oceny drugiego kryterium (np. czas)

Wyjście: Najlepsza ścieżka uwzględniająca oba kryteria (wielokryterialna optymalizacja)

1. Zainicjuj populację rozwiązań początkowych P:
   - P = generuj_randomowe_trasy(G)
2. Powtarzaj, dopóki nie osiągniemy kryterium zakończenia (np. liczba iteracji, brak poprawy): a. Oceń każde rozwiązanie w populacji: - Dla każdego rozwiązania r ∈ P oblicz: - koszt_1 = K1(r) - koszt_2 = K2(r) - dodaj (koszt_1, koszt_2) do zbioru rozwiązań Pareto
   b. Selekcja: wybierz najlepsze rozwiązania na podstawie reguł dominacji Pareto: - Dla każdej pary rozwiązań (r1, r2) w P: - Jeśli r1 dominuje r2 (r1 lepsze w obu kryteriach lub przynajmniej w jednym), zachowaj r1 - Jeśli r2 dominuje r1, zachowaj r2
   c. Crossover i mutacja: - Stwórz nową populację poprzez crossover i mutację: - crossover(r1, r2) - wymień fragmenty tras pomiędzy dwoma rozwiązaniami - mutacja(r) - wprowadź drobne zmiany w trasie (np. zamiana miejscami miast) - Dodaj do populacji nowych osobników
3. Po zakończeniu algorytmu zwróć zbiór rozwiązań Pareto:
   - ParetoFront = wybierz rozwiązania, które nie są dominowane przez inne rozwiązania w populacji
4. Zwróć najlepsze rozwiązanie lub zbiór rozwiązania (zależnie od aplikacji):
   - Zwróć rozwiązanie z ParetoFront z najlepszym kompromisem między kryteriami K1 i K2

Analiza wyników

- wyliczone metryki z komentarzami,

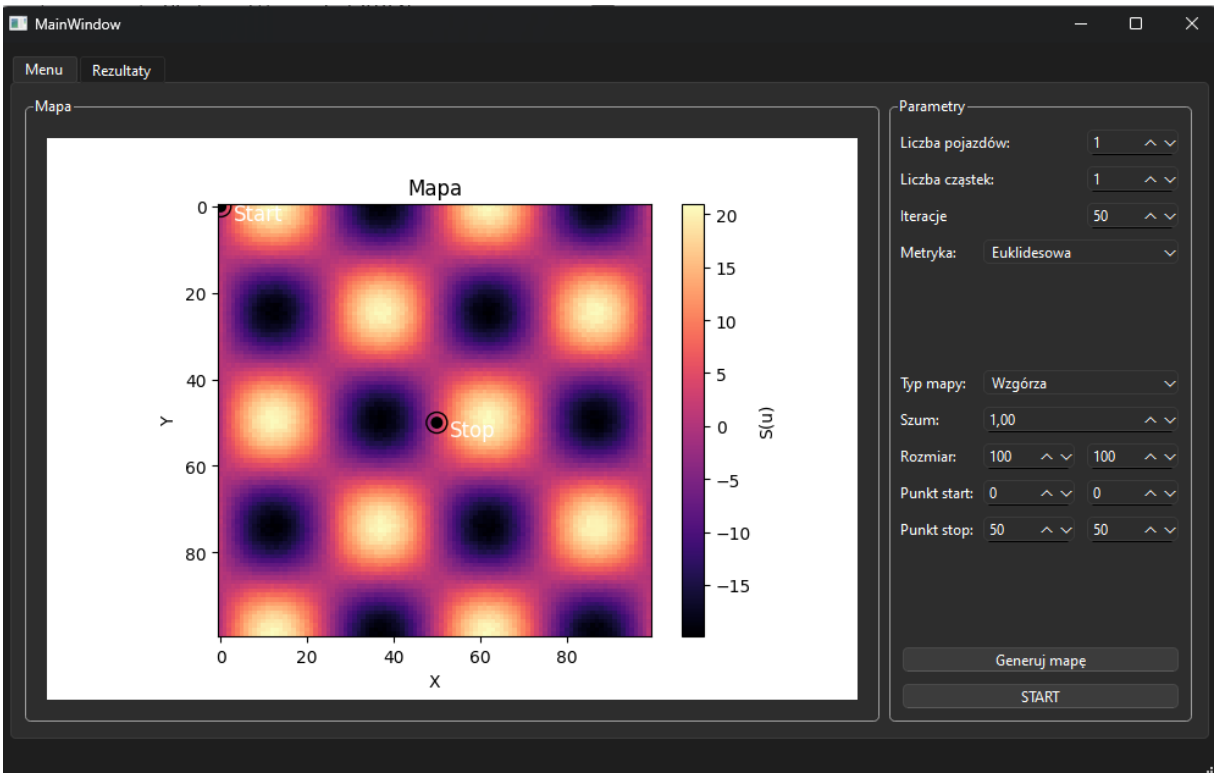
- optymalizacja parametrów + uzasadnienie, dlaczego ta została wybrana jako najważniejsza/najlepsza).

Przygotowanie aplikacji

Aplikacja została przygotowana z wykorzystaniem Pythona i biblioteki PyQt, która umożliwiła utworzenie okna z możliwością ustawiania parametrów, trybów oraz przedstawiania wizualizacji.

Aby uruchomić program należy pobrać kod lub paczkę „.zip” z repozytorium. Repozytorium znajduje się pod linkiem https://github.com/Julnowak/Optymalizacja-wielokryterialna/Projekt, jednak o dostęp należy poprosić autorów kodu. Jeżeli rozmiar plików na to pozwoli, wówczas pliki zostaną załączone również na platformie Upel.

Aby program działał prawidłowo należy po otwarciu w edytorze Pythona zainstalować wszystkie biblioteki zawarte w pliku „requirements.txt” poprzez komendę „pip install requirements.txt”. Gdy wszystkie biblioteki będą już zainstalowane, należy przejść do folderu „GUI” i uruchomić plik „**mainwindow.py**”. Podjęcie tej akcji powinno skutkować pojawieniem się okna aplikacji. W razie problemów, prosimy o kontakt z autorami.



Rys. 2 Zrzut ekranu przedstawiający interfejs

Podsumowanie

- Co zostało zrobione?

- Implementacja generatora terenu,
- Implementacja zmodyfikowanego CSO z funkcjami pomocniczymi,
- Implementacja zmodyfikowanego TSP z funkcjami pomocniczymi,
- Testowanie funkcji i analiza wyników algorytmów,
- Przygotowanie funkcjonalnego GUI

- Jakie wyniki wyszły?

- Znacznie wolniejszy TSP, ale lepsze wyniki
- CSO – duża zależność ustawień parametrów, problemy szukania trasy

- Czy wyniki są satysfakcjonujące?

- Algorytmy spełniają swoje zadanie, jednak są bardzo złożone obliczeniowo,

- Propozycje tego, co można zrobić, żeby poprawić wyniki lub rozwinąć projekt dalej.

- Zrównoleglić obliczenia,
- Dodać optymalizację parametrów algorytmów,

Wybrano algorytm CSO, gdyż założony PSO był dostosowany bardziej do detekcji problemów ciągłych. PSO z pustą mapą i niezerowymi przeszkodami są dość powszechne, jednak naszym zdaniem nie daje pełnego obrazu terenu. Zastosowanie mapy wysokościowej jest lepsze pod kątem planowania przyszłych tras pojazdów z wyprzedzeniem. Dzięki temu, roboty będą mogły sprawniej pokonywać trasy na znanym terenie i jedynie korygować trasę.