

# Projekt Optymalizacja Wielokryterialna

Realizacja SWD implementującego metody optymalizacji wielokryterialnej

**Autorzy:**

Julia Nowak

Adam Złocki

Jakub Szczypek

## Spis treści

Podział zadań.....	2
Wstęp.....	3
Problem badawczy.....	3
Opis wykorzystywanej bazy danych / zbioru decyzyjnego .....	3
Przygotowanie i weryfikacja danych .....	5
Opis ogólny .....	5
Parametry .....	5
Działanie funkcji .....	9
Implementacje metody wielokryterialnej .....	10
CSO (Cockroach Swarm Optimization) z adaptacją do problemów wielokryterialnych.....	10
Komiwojażer dostosowany do problemów 3D z optymalizacją wag grafu.....	14
Analiza wyników.....	17
Przygotowanie aplikacji .....	18
Podsumowanie.....	19

## Podział zadań

Członek zespołu	Zadania
Julia Nowak	<ul style="list-style-type: none"><li>• Przygotowanie GUI</li><li>• Dokumentacja – wstęp i przygotowanie danych</li><li>• Wstępna wersja kodu PSO</li></ul>
Adam Złocki	<ul style="list-style-type: none"><li>• Poprawki zaimplementowanego już algorytmu PSO</li><li>• Implementacja drugiego algorytmu (jeszcze niezdefiniowany)</li><li>• Praca nad dokumentacją</li></ul>
Jakub Szczypek	<ul style="list-style-type: none"><li>• Praca nad zaimplementowanym już algorytmem CSO–wprowadzanie poprawek i rozwijanie aktualnej wersji algorytmu</li><li>• Praca nad Komiwojażerem 3D</li><li>• Praca nad dokumentacją</li></ul>

## 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 uwzględni 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://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.

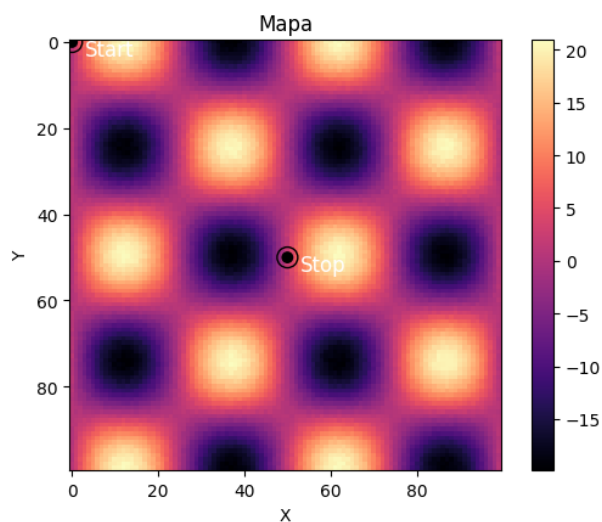
2. **terrain\_size** (krotka, domyślnie (100, 100))

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

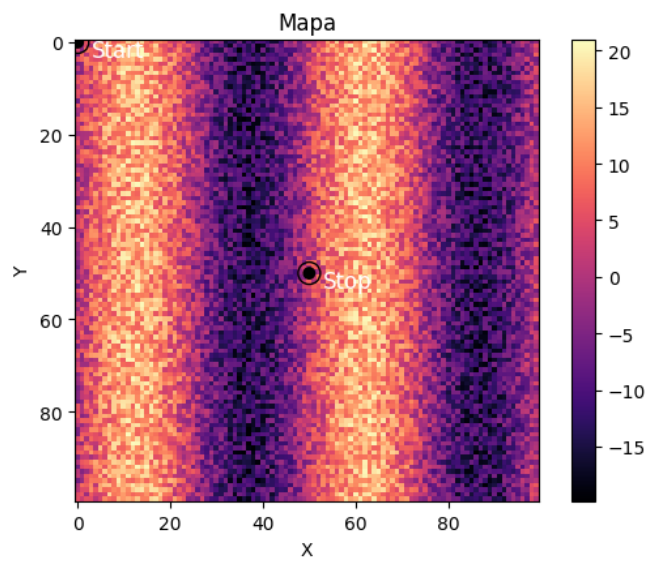
3. **terrain\_type** (łańcuch znaków, domyślnie "hills")

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

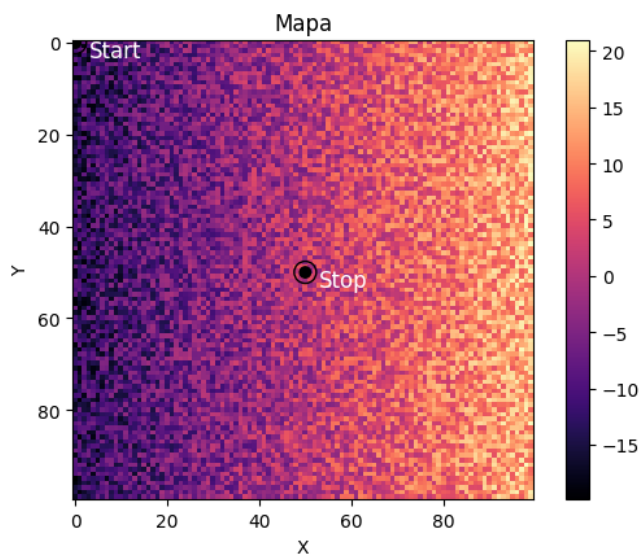
- **"hills"**: Pagórkowata powierzchnia oparta na sinusoidalnych wzorach.



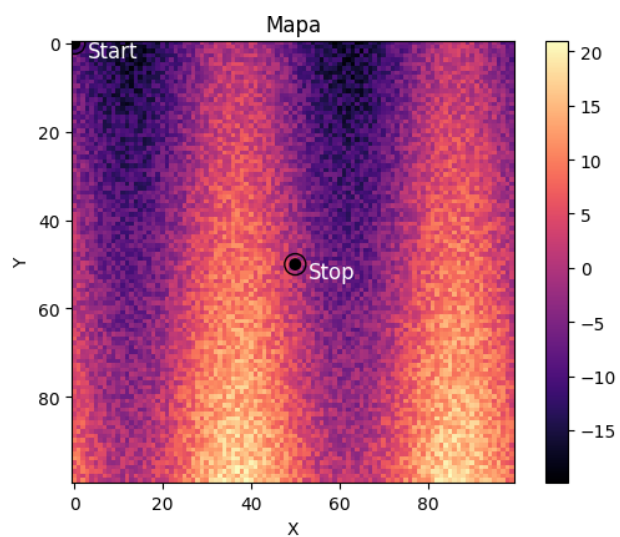
- **"lines"**: Powierzchnia z liniowymi wzorami wzdłuż osi X.



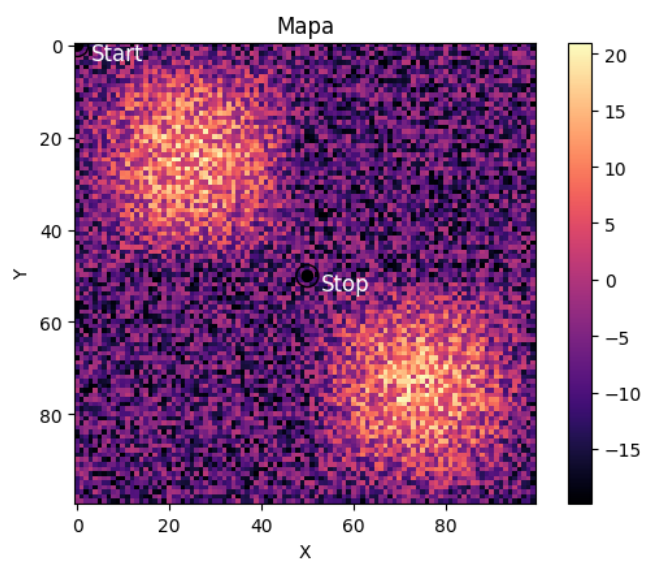
- **"slope"**: Jednostajny spadek wzdłuż osi X.



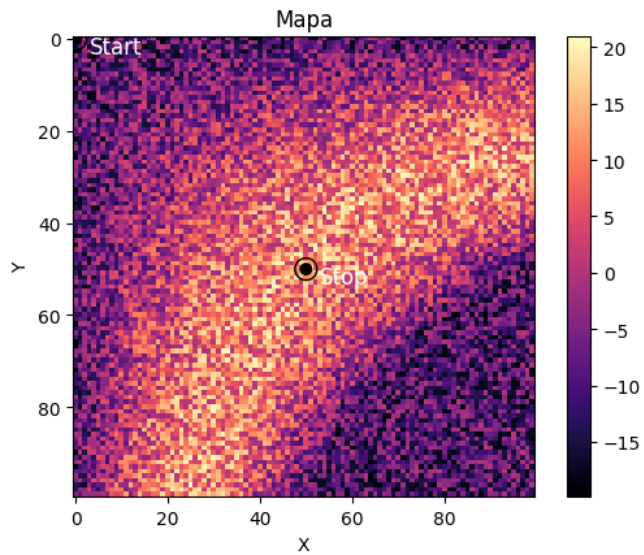
- **"razors"**: Powierzchnia z ostrymi przejściami wzdłuż osi X i Y.



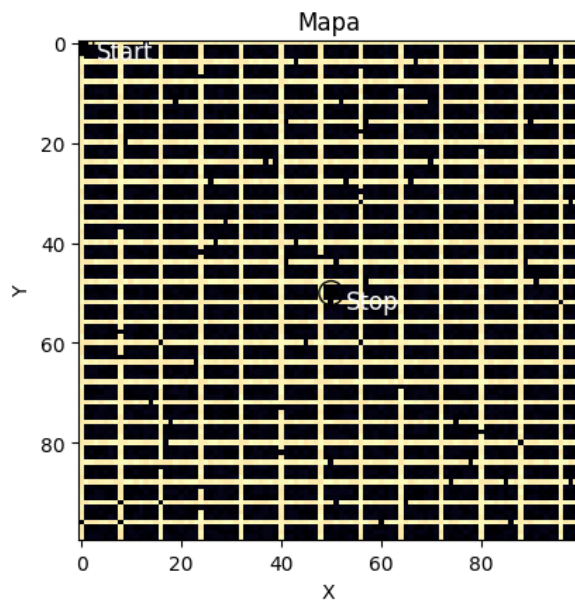
- **"canyon"**: Powierzchnia przypominająca kanion z dodatkowymi dolinami.



- **"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, umożliwiającą 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. Główną modyfikacją 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

Z komentarzem [J1]:

Implementacje metody wielokryterialnej :  
- wdrożenie metod optymalizacji parametrów /  
Implementacja dodatkowej metody jako metody  
odniesienia

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:  $f(x)$ ,  $x \in R^D$ 
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  $\text{abs}(x_i - x_j) < \text{visual}; f(x_j) < f(x_i)$  then
         $p_i = x_j$ 
      end if
    end for
    if  $p_i == x_i$  then
       $x_i = w \cdot x_i + \text{step} \cdot \text{rand} \cdot (p_g - x_i)$ 
    else
       $x_i = w \cdot x_i + \text{step} \cdot \text{rand} \cdot (p_i - x_i)$ 
    end if
    if  $f(x_i) < f(p_g)$  then
       $p_g = x_i$ 
    end if
  end for
  for  $i = 1$  to  $N$  do
     $x_i = x_i + \text{rand}(1, D)$ 
    if  $f(x_i) < f(p_g)$  then
       $p_g = x_i$ 
    end if
  end for
   $k = \text{randint}([1, N])$ 
   $x_k = p_g$ 
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](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 (TSP)** polega na znalezieniu możliwie najkrótszej trasy, która odwiedza każdy wierzchołek (miasto) dokładnie raz i kończy w punkcie początkowym. W trójwymiarowej wersji TSP dodatkowy wymiar (zazwyczaj wysokość terenu lub inna metryka przestrzenna) wpływa na wagę krawędzi łączących wierzchołki. Oznacza to, że jeśli przemieszczamy się z punktu A o współrzędnych  $(x_A, y_A, z_A)$  do punktu B  $(x_B, y_B, z_B)$ , odległość (lub koszt) może być wyznaczany przez odległość euklidesową w 3D:

$$d_{3D}(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

albo poprzez dowolną inną funkcję, uwzględniającą trudność pokonywania wzniesień i zejść (np. różnica wysokości  $|z_B - z_A|$  może dodawać dodatkowy koszt).

Dodatkowo, można wprowadzić wielokryterialny charakter problemu, np. przez:

1. Czas przejazdu między punktami (uwzględniający wysokość),
2. Koszt energii,
3. Bezpieczeństwo trasy,
4. itp.

i skompresować je w jedną funkcję celu (np. metodą wagową – Weighted Sum Method). W efekcie powstaje sparametryzowana funkcja kosztu, na której operuje algorytm.

### 1) Parametry algorytmu

W zależności od wybranej metody rozwiązania TSP 3D (np. algorytm genetyczny, 2-opt, ACO, itp.), definiujemy zestaw parametrów:

1. Liczba iteracji / pokoleń – maksymalna liczba powtórzeń głównej pętli algorytmu.
2. Rozmiar populacji (jeśli algorytm jest populacyjny, np. GA, ACO).
3. Metryka odległości – ustalamy wzór obliczania wag krawędzi w 3D.
4. Metoda wag (w problemach wielokryterialnych) – określa, w jakim stopniu poszczególne kryteria wpływają na funkcję celu.

Inne: np. wielkość kroku (jeżeli mamy podejście swarmowe), prawdopodobieństwo mutacji (w GA) itp.

## 2) Główna pętla algorytmu TSP 3D

Założmy, że stosujemy klasyczne heurystyczne/populacyjne podejście, zainspirowane algorytmem przedstawionym wcześniej (porównywalnym z algorytmem CSO). Główne etapy pracy wyglądają następująco:

### 1. Inicjalizacja

- Ustalamy parametry (liczba iteracji, rozmiar populacji, metoda wag itd.).
- Generujemy początkową populację rozwiązań (ścieżek) – np. losowe permutacje wierzchołków 3D lub heurystyka nearest neighbor.

### 2. Wyznaczenie $P_i$ oraz $P_g$ (minimum lokalne i globalne)

- Każda ścieżka (odpowiednik „karalucha” w CSO) jest porównywana z innymi rozwiązaniami w zasięgu (np. w turnieju selekcyjnym czy w sąsiedztwie permutacji).
- $P_i$  = najlepsze rozwiązanie lokalne (np. w puli turnieju).
- $P_g$  = najlepsze rozwiązanie globalne – minimalny koszt w całej populacji.
- Jeżeli dany osobnik jest „lokalnie najsilniejszy”, wtedy w następnych krokach dąży do globalnego lidera  $P_g$ ; w przeciwnym wypadku do minimum lokalnego  $P_i$ .

### 3. Procedura „Podążanie”

- Ścieżka (rozwiązanie) wykonuje krok w stronę  $P_i$  lub  $P_g$ . W TSP 3D może to oznaczać np. częściową wymianę krawędzi (w GA – crossover), drobną modyfikację permutacji w kierunku innego rozwiązania, czy heurystykę 2-opt/3-opt bazowaną na zbliżeniu do wzorca.
- Możemy zapisać to w stylu:

$$X_i = X_i + step \times rand(0,1) \times (P_g - X_i)$$

lub analogicznie z  $P_i$ .

- W praktyce: zmieniamy kolejność odwiedzanych miast, by przybliżyć się do kolejności (lub fragmentu ścieżki) występującej w  $P_g$  (najlepszym rozwiązaniu globalnym) czy  $P_i$ .

### 4. Rozpraszenie (dyspersja)

- W pewnych iteracjach (z określonym prawdopodobieństwem) wprowadzamy losową modyfikację trasy.
- W TSP 3D może to być np. wylosowanie jednego fragmentu ścieżki i zamiana kolejności miast, wstawienie miasta gdzie indziej, itp.
- Rozprasza to populację rozwiązań i pozwala unikać lokalnych ekstremów.

### 5. Bezwzględne zachowanie

- Można dodatkowo usuwać „najstabsze” rozwiązania (pożerane przez  $P_g$ ) i zastępować je kopią najlepszego.
- W ten sposób zwiększa się presja selekcyjna, bo najstabsze ścieżki zostają wyparte.

### 6. Aktualizacja populacji i sprawdzenie kryterium stopu

- Po wykonaniu kroków podążania i dyspersji, każdy osobnik aktualizuje swój koszt.
- Jeśli liczba iteracji (lub zmiana kosztu jest znikoma) osiągnęła kryterium stopu, algorytm się kończy.

### 3) Struktura danych wyjściowych

- Najlepsza znaleziona ścieżka (permutacja wierzchołków) – czy to cyklicznie (standardowy TSP: wracamy do miasta startowego), czy w wersji „start → end” (otwarta ścieżka).
- Wartość funkcji kosztu (3D distance) – ile wynosi całkowita długość / energia / czas przejścia.
- Lista najlepszych rozwiązań w każdej iteracji – do analizy post-processingu (np. wykres zbieżności).
- Lista wartości kosztu – pozwala ocenić tempo zbieżności algorytmu.

### 4) Podsumowanie

1. **Wstęp:** TSP 3D rozszerza klasyczny problem TSP do trójwymiarowej metryki (np. uwzględniając różnice wysokości).
2. **Parametry:** Liczba iteracji, rozmiar populacji, definicja metody wag (dla problemów wielokryterialnych), widzialność (w CSO), wielkość kroku itd.
3. **Główna pętla:**
  - Wyznaczenie najlepszego lokalnego ( $P_l$ ) i globalnego ( $P_g$ ).
  - Podążanie – modyfikacja ścieżek (pojedyncze kroki w kierunku  $P_l$  lub  $P_g$ ).
  - Rozpraszanie (dyspersja) – losowe zmiany rozwiązań, aby uniknąć utkwienia w lokalnym minimum.
  - Bezwzględne zachowanie (opcjonalnie) – usunięcie najstabszych rozwiązań, zastąpienie ich klonem  $P_g$ .
  - Aktualizacja populacji, sprawdzenie stopu.
4. **Wynik:**
  - Najlepsza ścieżka (kolejność odwiedzin miast) w grafie 3D,
  - Funkcja kosztu tej ścieżki,
  - Historia najlepszych wyników w każdej iteracji.



## **Analiza wyników**

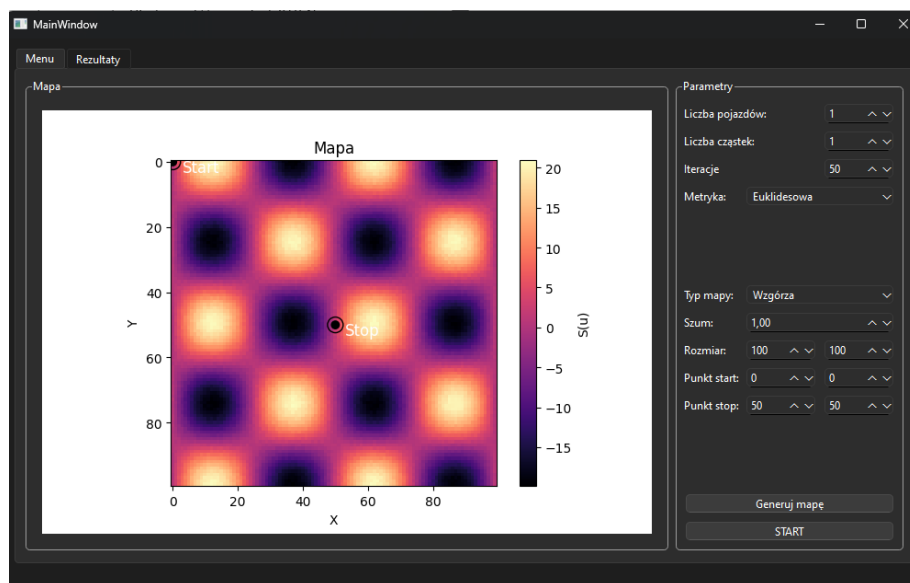
- wyliczone metryki z komentarzami,
- optymalizacja parametrów + uzasadnienie, dlaczego ta została wybrana jako najważniejsza/najlepsza).

## 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.

