Kalibracja modelu EPANET algorytmem White Shark Optimizer

Igor Swat Rafał Piwowar

1 Środowisko symulacyjne EPANET

EPANET to oprogramowanie opracowane przez Amerykańską Agencję Ochrony Środowiska (EPA) służące do modelowania systemów dystrybucji wody. Umożliwia ono symulację zachowania hydraulicznego oraz jakościowego w sieciach ciśnieniowych, obejmujących rury, węzły, pompy, zawory, zbiorniki magazynujące oraz rezerwuary.

EPANET oferuje trzy podstawowe funkcjonalności:

- **Modelowanie hydrauliki** – symulacja przepływu, ciśnienia, stanów pomp i zaworów;
- **Modelowanie jakości wody** – analiza wieku wody oraz ogólnej jakości w poszczególnych punktach sieci;
- **Modelowanie bezpieczeństwa i odporności wodnej** – analiza scenariuszy zagrożeń zewnętrznych, wspierana przez rozszerzenia EPANET-MSX i EPANET-RTX.

1.1 Elementy modelu sieci wodociągowej

W EPANET dostępne są następujące typy obiektów:

Węzeł (Junction) Reprezentuje punkt sieci, do którego podłączone są rury. Służy jako miejsce wejścia i wyjścia przepływu wody.

- **Dane wejściowe (hiperparametry):** podniesienie (elevation), zapotrzebowanie na wodę (water demand), początkowa jakość wody (initial water quality).
- **Dane wyjściowe:** wysokość hydrauliczna (hydraulic head), ciśnienie (pressure), jakość wody (water quality).

Rura (Pipe) Łączy dwa węzły. Rury są zawsze pełne, a kierunek przepływu zależy od różnicy wysokości hydraulicznej.

- **Dane wejściowe:** węzły początkowy i końcowy, średnica, długość, współczynnik szorstkości, status (otwarta/zamknięta/zawór zwrotny).
- **Dane wyjściowe:** przepływ (flow rate), prędkość (velocity), straty ciśnienia (headloss), współczynnik tarcia (Darcy-Weisbach friction factor), średnia szybkość reakcji chemicznych (average reaction rate), średnia jakość wody (average water quality).

Zbiornik naturalny (Reservoir) Reprezentuje nieskończone źródło lub ujście wody (np. jezioro, rzeka).

- **Dane wejściowe:** wysokość hydrauliczna, początkowa jakość wody.

Zbiornik sztuczny (Tank) Reprezentuje magazyn wody o skończonej pojemności.

- **Dane wejściowe:** podniesienie dolne, średnica, początkowy/minimalny/maksymalny poziom wody, początkowa jakość wody.
- **Dane wyjściowe:** wysokość hydrauliczna, jakość wody.

Pompa (Pump) Element zapewniający przepływ wody między węzłami, działający w oparciu o krzywą pompy (zależność przepływu i wysokości).

Szczegółowy opis komponentów znajduje się w dokumentacji EPANET 2.2: *3. The Network Model*. Lista jednostek i parametrów: *Units of Measurement*.

2 White Shark Optimizer – implementacja (Python)

Pierwszym punktem naszych prac była implementacja algorytmu White Shark Optimizer. Zgodnie z planem prac, implementacja miała zostać wykonana zarówno w Pythonie, jak i Elixirze (biblioteka `nx`) dla celów porównawczych.

Przed przystąpieniem do implementacji dokonaliśmy przeglądu istniejących już i dostępnych w sieci rozwiązań:

- Autorzy oryginalnego artykułu nie udostępniają gotowej implementacji, jedynie pseudokod.

- Istnieje implementacja w MATLAB (MathWorks File Exchange).
- Niektóre dostępne implementacje różnią się od opisu z artykułu, np. w sposobie aktualizacji pozycji.

Poniżej zamieszczona została implementacja algorytmu WSO w języku Python wraz z wyszczególnieniem najważniejszych fragmentów.

```python
# Main WSO mechanism
# - Directly connected with a given problem by taking evaluator, dimentionality and parameter ranges as input
# - We assume that parameters are integers / floats in form of numpy array
class Optimizer:

    def __init__(self):
        # Initialize hyperparameters - according to WSO paper
        self.p_min = 0.5
        self.p_max = 1.5
        self.tau = 4.125
        self.mu = 2 / abs(2 - self.tau - np.sqrt(self.tau ** 2 - 4 * self.tau))
        self.f_min = 0.07
        self.f_max = 0.75
        self.a0 = 6.25
        self.a1 = 100.0
        self.a2 = 0.0005

        self.b_scale = 0.9


    def optimize(self, problem: Problem, no_sharks: int = 10, steps: int = 10,
                 verbose: bool = False, logging_freq: int = 10) -> tuple[np.ndarray, float]:
        ''' Performs WSO to find a solution that minimizes problem.evaluate() function values

            Returns a pair of (best_solution, best_solution_eval).
            - logging_freq: number of iterations between logs
        '''

        # Step 0 - clear history tables
        self.best_fitness_history.clear()

        # Step 1 - Generate initial population with respect dimensionality
        # - W for shark positions
        # - v for shark velocities
        W = np.random.uniform(problem.lb, problem.ub, (no_sharks, problem.dim))
```

```
        v = np.zeros_like(W)      # zeros_like() automatically copies dimensionality of an array

        with ProcessPoolExecutor(max_workers=self.no_workers) as executor:
            # Step 2 - Evaluate initial population fitness
            # - Iterates over population ranks (position of a single shark) and creates 1D fitness vector
            # - Parallel execution
            from .worker import evaluate_with_local_worker

            fitness = np.empty(no_sharks)

            future_to_idx = {
                executor.submit(
                    evaluate_with_local_worker,
                        W[i, :],
                        self.model_filepath,
                        self.tmp_filepath,
                        problem.time_hrs,
                        problem.measured_df,
                        problem.dim,
                        problem.lb,
                        problem.ub
                ): i for i in range(no_sharks)
            }

            for future in as_completed(future_to_idx):
                i = future_to_idx[future]
                try:
                    fitness[i] = future.result()
                except Exception as exc:
                    print(f'Exception generated by shark {i}: {exc}')
                    fitness[i] = np.inf

            fitness_min = np.min(fitness)
            W_best = W.copy()                   # 2D matrix
            W_gbest = W[np.argmin(fitness)]     # 1D vector

            # Main WSO loop
            for k in tqdm(range(1, steps + 1)):
                # Calculate adaptive parameters
```

```python
p1 = self.p_max + (self.p_max - self.p_min) * np.exp(-(4 * k / steps)**2)
p2 = self.p_min + (self.p_max - self.p_min) * np.exp(-(4 * k / steps)**2)
mv = 1 / (self.a0 + np.exp((steps / 2.0 - k) / self.a1))
s_s = abs(1 - np.exp(-self.a2 * k / steps))

# Step 3 - update shark velocities
# - NOTE: Can be additionally vectorized
nu = np.random.randint(0, no_sharks, no_sharks)
for i in range(no_sharks):
    c1 = random.random()
    c2 = random.random()
    v[i, :] = self.mu * (v[i, :] + p1 * c1 * (W_gbest - W[i, :]) + p2 * c2 * (W_best[nu[i], :] - W[i, :]))

# Step 4 - update positions with wavy motion or random allocation
f = self.f_min + (self.f_max - self.f_min) / (self.f_max + self.f_min)
for i in range(no_sharks):
    a = W[i, :] > problem.ub
    b = W[i, :] < problem.lb
    w0 = np.logical_xor(a, b)
    if random.random() < mv:
        W[i][w0] = problem.ub[w0] * a[w0] + problem.lb[w0] * b[w0]
    else:
        shift = v[i, :] / f
        W_new = W[i, :] + shift

        # Bounce back if needed
        if not np.any((problem.lb > W[i, :]) | (problem.ub < W[i, :])) and np.any((problem.lb > W_new) | (problem.ub < W_new)):
            # Adjust position and velocity
            P = np.clip(W_new, problem.lb, problem.ub)
            back_shift = P - W_new

```

```
            W[i, :] = P + back_shift * self.b_scale
            v[i, :] = -v[i, :] * self.b_scale
        else:
            W[i, :] = W_new

# Step 5 - school movement update
for i in range(no_sharks):
    # TODO: Is this thing even correct?
    if random.random() <= s_s:
        D = np.abs(np.random.rand() * (W_gbest -
        ↪  W[i, :]))
        if i == 0:
            sgn =
            ↪  np.sign(np.random.rand(problem.dim)
            ↪  - 0.5)
            W[i, :] = W_gbest +
            ↪  np.random.rand(problem.dim) * D *
            ↪  sgn
        else:
            sgn =
            ↪  np.sign(np.random.rand(problem.dim)
            ↪  - 0.5)
            tmp = W_gbest +
            ↪  np.random.rand(problem.dim) * D *
            ↪  sgn
            W[i, :] = (W[i, :] + tmp) / (2 *
            ↪  random.random())

# Step 6 - return the sharks to the original
↪  solution space
W = np.clip(W, problem.lb, problem.ub)

# Step 7 - evaluate and update best positions
# - Multithreading enabled
future_to_shark = {
    executor.submit(
        evaluate_with_local_worker,
        W[i, :],
        self.model_filepath,
        self.tmp_filepath,
        problem.time_hrs,
        problem.measured_df,
        problem.dim,
```

```
                        problem.lb,
                        problem.ub
                    ): i for i in range(no_sharks)
                }

                for future in as_completed(future_to_shark):
                    i = future_to_shark[future]
                    try:
                        fit = future.result()
                        if fit < fitness[i]:
                            W_best[i, :] = W[i, :]
                            fitness[i] = fit
                        if fitness[i] < fitness_min:
                            fitness_min = fitness[i]
                            W_gbest = W_best[i].copy()
                    except Exception as exc:
                        print(f'Exception generated by shark {i}: {exc}')

                # Save optimization progress to history tables
                self.best_fitness_history.append(fitness_min)

                # Verbose logging
                if verbose and k % logging_freq == 0:
                    progress = k / steps
                    print(f"\nProgress: {int(progress*100)}% | Best fitness: {fitness_min:.8f}")

        return W_gbest, fitness_min
```

2.1 Hiperparametry

Wartości hiperparametrów używanych w algorytmie zostały zaczerpnięte z artykułu jako propozycje autorów. Dodatkowo przyjmujemy arbitralnie:

- rozmiar populacji,
- liczbę iteracji.

2.2 Inicjalizacja populacji

Pozycje rekinów – wektorów w przestrzeni parametrów – inicjalizujemy z rozkładu jednostajnego w zadanym przedziale:

$$x_i \sim \mathcal{U}(l_i, u_i),$$

gdzie l_i, u_i to odpowiednio dolna i górna granica dla i-tego parametru.

2.3 Ewaluacja pozycji

Poszczególne pozycje rekinów wartościowane są funkcją straty opisującą dany problem. Algorytm ma na celu minimalizowanie wartości tejże funkcji. W tym celu na wejściu algorytmu przekazywana jest abstrakcyjna reprezentacja problemu, gdzie metoda evaluate() implementuje funkcję straty.

2.4 Iteracyjna ewolucja populacji

Główna pętla algorytmu składa się z:

1. Aktualizacji prędkości rekinów.
2. Przemieszczania zgodnie z prędkością (lub losowo).
3. Przemieszczania względem gromady.
4. Ewaluacji i odrzucania rozwiązań poza zakresem.

Ponieważ docelowo fragment ten jest najbardziej czasochłonnym etapem algorytmu w kontekście uruchamiania symulacji w środowisku EPANET, podjęliśmy decyzję o ignorowaniu rozwiązań (rekinów) wykraczających poza dopuszczalny obszar, co zmniejszy liczbę potencjalnych wywołań symulacji.

2.5 Mechanizm odbijania rekinów od granic przestrzeni poszukiwań

W algorytmie White Shark Optimizer (WSO) należy zapewnić, aby pozycje rekinów (potencjalnych rozwiązań) nie wychodziły poza dopuszczalny zakres przestrzeni poszukiwań. Dolne i górne ograniczenia przestrzeni są zapisane jako wektory:

$$\mathbf{lb} = [lb_1, lb_2, \ldots, lb_D], \quad \mathbf{ub} = [ub_1, ub_2, \ldots, ub_D],$$

gdzie D oznacza wymiar przestrzeni.

Nowa pozycja rekina i-tego po kroku aktualizacji obliczana jest jako:

$$\mathbf{W}_{new}^{(i)} = \mathbf{W}^{(i)} + \frac{\mathbf{v}^{(i)}}{f},$$

gdzie f to współczynnik wygładzający (ang. wave factor). Jeżeli nowa pozycja wychodzi poza dozwolone granice, zostaje ona skorygowana w oparciu o mechanizm odbicia.

Korygowana pozycja oraz prędkość są wyznaczane na podstawie następujących wzorów:

$$\mathbf{P}^{(i)} = \text{clip}(\mathbf{W}_{new}^{(i)}, \mathbf{lb}, \mathbf{ub}),$$
$$\Delta^{(i)} = \mathbf{P}^{(i)} - \mathbf{W}_{new}^{(i)},$$
$$\mathbf{W}^{(i)} = \mathbf{P}^{(i)} + b_{scale} \cdot \Delta^{(i)},$$
$$\mathbf{v}^{(i)} = -b_{scale} \cdot \mathbf{v}^{(i)},$$

gdzie $b_{scale} \in (0, 1)$ to współczynnik strat energii w momencie odbicia. W implementacji przyjęto $b_{scale} = 0{,}9$.

Znak minus w aktualizacji prędkości odpowiada odbiciu od granicy, natomiast tłumienie przez b_{scale} pozwala ograniczyć zasięg kolejnych ruchów rekina i stabilizować eksplorację. Dzięki temu algorytm nie tylko unika niepoprawnych rozwiązań, ale również zwiększa efektywność przeszukiwania w pobliżu granic przestrzeni decyzyjnej.

2.6 Testy porównawcze

W celu oceny efektywności i stabilności algorytmu *WSO* (White Shark Optimization) przeprowadziliśmy serię testów porównawczych z algorytmem *PSO* (Particle Swarm Optimization). Testy wykonano na dwóch standardowych funkcjach benchmarkowych:

- **Rastrigin** (wymiarowość: $D = 2$, obszar poszukiwań $[-5.12, 5.12]^2$),
- **Rosenbrock** (wymiarowość: $D = 2$, obszar poszukiwań $[-5.12, 5.12]^2$).

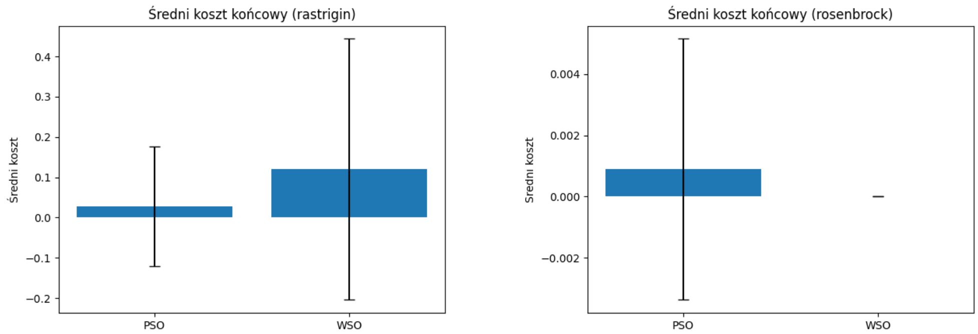
Dla każdego problemu uruchomiono $N = 200$ niezależnych prób. Parametry algorytmów dobrano następująco:

- PSO: liczba cząstek $n_{pso} = 30$, współczynniki przyspieszeń $c_1 = 0.5$, $c_2 = 0.3$, współczynnik bezwładności $w = 0.9$, maksymalna liczba iteracji $T = 100$.
- WSO: liczba rekinów (białych rekinów) $n_{wso} = 30$, liczba kroków optymalizacji $T = 100$.

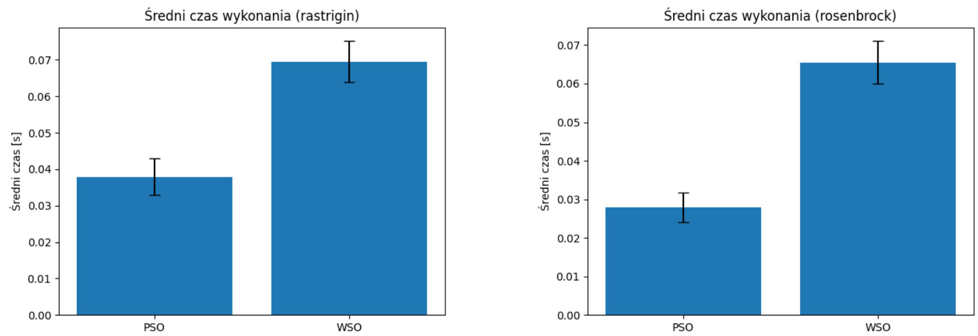
Jako miary jakości porównania przyjęto:

1. *Średni najlepszy koszt* uzyskany po T iteracjach,
2. *Odchylenie standardowe* wartości końcowych kosztów,
3. *Średni czas* wykonania algorytmu.

Poniższe wykresy przedstawiają wyniki porównań:



Rysunek 1: Porównanie średnich najlepszych kosztów dla algorytmów PSO i WSO na funkcjach Rastrigin i Rosenbrock.



Rysunek 2: Porównanie średniego czasu wykonania algorytmów PSO i WSO dla różnych funkcji testowych.

Dyskusja wyników: Na podstawie przeprowadzonych testów można zauważyć wyraźne różnice w skuteczności obu algorytmów w zależności od funkcji testowej.

Dla funkcji **Rastrigin**, która charakteryzuje się dużą liczbą lokalnych minimów, lepsze wyniki osiągnął algorytm *PSO*, uzyskując niższe średnie wartości końcowego kosztu oraz mniejsze odchylenie standardowe. Może to wskazywać na jego większą zdolność do eksploracji wielomodalnych przestrzeni poszukiwań.

Z kolei dla funkcji **Rosenbrock**, znanej z wąskiej i zakrzywionej doliny globalnego minimum, lepiej poradził sobie algorytm *WSO*. Osiągał on niższe wartości końcowe funkcji celu, co sugeruje jego lepsze właściwości eksploatacyjne w tego typu krajobrazach optymalizacyjnych.

Pod względem czasowym, w obu przypadkach *PSO* okazał się nieco szybszy od *WSO*. Różnica ta wynika prawdopodobnie z prostszej struktury PSO i mniejszego kosztu obliczeniowego na iterację, podczas gdy WSO, odwzorowując bardziej złożone mechanizmy zachowania (np. ruchy rekina), wymaga większych zasobów obliczeniowych.

Podsumowując, *PSO* cechuje się lepszą wydajnością w problemach wielomodalnych i krótszym czasem wykonania, natomiast *WSO* oferuje większą precyzję w problemach wymagających dokładnej eksploracji wąskich obszarów przestrzeni rozwiązań.

2.7 Zrównoleglenie obliczeń WSO z użyciem procesów i workerów

Docelowy algorytm WSO wykorzystywany podczas optymalizacji modelu sieci wodociągowej został wzbogacony o aspekt równoległości obliczeń.

W języku Python implementacja ta została zrealizowana z użyciem biblioteki `multiprocessing`, która pozwala uruchamiać niezależne procesy systemowe, a tym samym omija ograniczenie tzw. *Global Interpreter Lock* (GIL). GIL jest mechanizmem zapewniającym wzajemne wykluczanie w interpreterze CPythona, przez co nawet na systemach wielordzeniowych tylko jeden wątek może wykonywać kod bajtowy Pythona w danej chwili. W związku z tym tradycyjny threading w Pythonie nie przynosi realnych korzyści w kontekście obliczeń CPU-zależnych. Zastosowanie procesów (zamiast wątków) pozwala na równoczesne uruchomienie wielu symulacji EPANET.

Dla każdego procesu tworzony jest osobny worker klasy `EpanetWorker`, który działa we własnym, odizolowanym katalogu tymczasowym o nazwie:

`worker_{pid}`,

gdzie `pid` to identyfikator procesu operacyjnego. Wymóg ten wynika z faktu, że biblioteka EPANET tworzy pliki tymczasowe w bieżącym katalogu roboczym, co mogłoby prowadzić do kolizji między równolegle uruchomionymi symulacjami.

Każdy worker posiada własną kopię pliku modelu sieci, ładowaną przy jego inicjalizacji:

```
worker_model_path = os.path.join(worker_dir, model_filename)
shutil.copy(model_path, worker_model_path)
self.model = epanet(worker_model_path)
```

oraz instancję klasy `EpanetProblem`, która obsługuje ocenę jednej pozycji w przestrzeni decyzyjnej:

```python
self.problem = EpanetProblem(dim=dim, lb=lb, ub=ub,
                             model=self.model,
                             time_hrs=time_hrs,
                             measured_df=measured_df.copy())
```

Zrównoleglenie pozwala na równoczesne uruchomienie wielu takich instancji EPANET, co istotnie skraca czas jednej iteracji algorytmu WSO. Ocena wszystkich osobników (rekinów) w danej iteracji może być wykonywana równolegle jako mapowanie funkcji `evaluate_with_local_worker` na zbiór wektorów parametrów:

```python
with multiprocessing.Pool(processes=N) as pool:
    fitness = pool.map(evaluate_with_local_worker, population)
```

Po zakończeniu działania każdy worker automatycznie zwalnia zasoby i zamyka model EPANET:

```python
def __del__(self):
    self.model.unload()
```

Rozwiązanie to zapewnia pełną niezależność symulacji oraz skalowalność względem liczby dostępnych rdzeni CPU, co znacząco zwiększa efektywność optymalizacji.

3 White Shark Optimizer – implementacja (Elixir)

3.1 Biblioteka Nx

Nx to biblioteka obliczeń numerycznych dla języka Elixir, umożliwiająca pracę z wielowymiarowymi strukturami danych (tensorami) oraz przeprowadzanie wydajnych operacji matematycznych. Dzięki integracji z backendami takimi jak Google XLA czy LibTorch, Nx pozwala na szybkie i efektywne przetwarzanie danych, co czyni go idealnym narzędziem do zastosowań w sztucznej inteligencji, uczeniu maszynowym oraz analizie dużych zbiorów danych.

3.2 Implementacja

3.2.1 Hiperparametry i Definicja Problemu

```elixir
defmodule Hyperparameters do
  @type t :: %Hyperparameters{
```

```
          p_min: float(),
          p_max: float(),
          tau: float(),
          f_min: float(),
          f_max: float(),
          a0: float(),
          a1: float(),
          a2: float(),
          n: integer(),
          rand_fun: (() -> float()),
          mu: float() | nil,
          f: float() | nil
        }

  defstruct p_min: 0.5,
            p_max: 1.5,
            tau: 4.125,
            f_min: 0.07,
            f_max: 0.75,
            a0: 6.25,
            a1: 100,
            a2: 0.0005,
            n: 100,
            rand_fun: &:rand.uniform/0,
            mu: nil,
            f: nil

  defp compute_mu(tau) do
    2 / (abs(2 - tau - :math.sqrt(tau * tau - 4 * tau)))
  end

  defp compute_f(f_max, f_min) do
    f_min + (f_max - f_min) / (f_max + f_min)
  end

  @spec new(map()) :: t()
  def new(opts \\ %{}) do
    tau = Map.get(opts, :tau, %__MODULE__{}.tau)
    f_max = Map.get(opts, :f_max, %__MODULE__{}.f_max)
    f_min = Map.get(opts, :f_min, %__MODULE__{}.f_min)


    %__MODULE__{}
```

```
    |> struct(opts)
    |> Map.update!(:mu, fn _ -> compute_mu(tau) end)
    |> Map.update!(:f, fn _ -> compute_f(f_max, f_min) end)
  end


end

defmodule Problem do
  @type t :: %Problem{
          name: String.t() | nil,
          d: integer(),
          fun: (Nx.Tensor.t() -> float()) | nil,
          l: Nx.Tensor.t() | nil,
          u: Nx.Tensor.t() | nil,
          minimize: boolean(),
        }

  defstruct name: nil,
            d: nil,
            fun: nil,
            l: nil,
            u: nil,
            minimize: true

  @spec new(integer()) :: t()
  def new(opts \\ %{}) do
    d = Map.get(opts, :d, 3)
    fun = case Map.get(opts, :minimize, true) do

      true -> Map.get(opts, :fun, nil)
      false -> fn tensor -> -Map.get(opts, :fun, nil).(tensor) end
    end

    computed_fields = %{
      l: Nx.broadcast(-10, {d}),
      u: Nx.broadcast(10, {d}),
      fun: fun

    }
    %__MODULE__{}
    |> struct(Map.merge(computed_fields, opts))
    |> struct(opts)
```

```
  end
end
```

3.2.2 White Shark Optimizer

```
defmodule WhiteSharkOptimizer do
  @moduledoc """
  Implements the White Shark Optimization algorithm for solving optimization problems in Nx.
  Based on
  https://www.sciencedirect.com/science/article/pii/S0950705122001897
  https://www.mathworks.com/matlabcentral/fileexchange/107365-white-shark-optimizer-wso
  """
  require Nx
  @type t :: %WhiteSharkOptimizer{
          problem: Problem.t() | nil,
          hyperparams: Hyperparameters.t | nil,
          key: integer() | nil,
          w: Nx.Tensor.t | nil,
          v: Nx.Tensor.t | nil,
          k: integer(),
          max_iterations: integer(), #called K in the paper
          p1: float() | nil,
          p2: float() | nil,
          wgbestk: Nx.Tensor.t() | nil,
          best_g_fitness: float() | nil,
          w_best: Nx.Tensor.t() | nil,
          best_fitness: Nx.Tensor.t | nil,
          fitness_results: Nx.Tensor.t | nil,
          verbose: boolean(),
        }

  defstruct problem: nil,
            hyperparams: nil,
            key: nil,
```

```
            w: nil,
            v: nil,
            k: 0,
            max_iterations: 100,
            p1: nil,
            p2: nil,
            wgbestk: nil,
            best_g_fitness: :infinity,
            w_best: nil,
            best_fitness: nil,
            fitness_results: nil,
            verbose: true


  @spec compute_ps(t()) :: t()
  defp compute_ps(wso) do
    p_min = wso.hyperparams.p_min
    p_max = wso.hyperparams.p_max
    p1 = p_max + (p_max - p_min) * :math.exp(-:math.pow( 4 * wso.k / wso.max_iterations, 2))
    p2 = p_min + (p_max - p_min) * :math.exp(-:math.pow( 4 * wso.k / wso.max_iterations, 2))
    %{wso | p1: p1, p2: p2}
  end

  @doc """
  Initializes a new instance of the WhiteSharkOptimizer struct with the provided problem, hyperparameters, and optional configuration.

  ### Parameters:
  - `problem`: A struct defining the optimization problem to be solved. It must include the following fields:
    - `d`: Dimensionality of the problem.
    - `l`: Lower bounds for the search space as an Nx.Tensor of shape `{d}`.
    - `u`: Upper bounds for the search space as an Nx.Tensor of shape `{d}`.
    - `fun`: A fitness function of the form `(Nx.Tensor.t() -> float())`, used to evaluate the quality of solutions.
    - The dimensionality (`d`) must match the bounds (`l` and `u`).
```

```
- `hyperparams`: A struct specifying the algorithm's hyperparameters such as the population size (`n`), learning rate (`mu`), and others necessary for controlling the behavior of the optimizer.
- `opts`: A map of optional configuration values. These can include:
  - `key`: Random key for generating initial population and randomness. Defaults to a key generated by `Nx.Random.key(0)` if not provided.
  - `verbose`: If `false`, the best solution will be printed every iteration. Defaults to `true`.

### Raises:
- `ArgumentError`: Raised in the following cases:
  - `problem` is `nil`, as the optimizer cannot operate without a defined problem.
  - `problem` is missing required fields (`d`, `l`, `u`, or `fun`).
  - The dimensions of the bounds (`l` and `u`) do not match the dimensionality (`d`).
  - The `fun` field is not a valid function.

### Returns:
- A `WhiteSharkOptimizer` struct initialized with computed fields.
"""
@spec new(Problem.t(), Hyperparameters.t(), map()) :: t()
def new(%{d: _, l: _, u: _, fun: _} = problem, hyperparams, opts \\ %{}) do
  validate_problem(problem)

  key = Map.get(opts, :key, Nx.Random.key(0))

  {random_tensor, key} = Nx.Random.uniform(key, shape: {hyperparams.n, problem.d})
  w_initial = random_tensor
        |> Nx.multiply(Nx.subtract(problem.u, problem.l))
        |> Nx.add(problem.l)
  computed_fields = %{
    w: w_initial,
    v: Nx.broadcast(0, {hyperparams.n, problem.d}),
    key: key,
    problem: problem,
    hyperparams: hyperparams,
```

```
      w_best: w_initial,
      best_fitness: Nx.broadcast(Nx.Constants.infinity(), {hyperparams.n})
    }

    %__MODULE__{}
    |> struct(Map.merge(computed_fields, opts))
    |> compute_ps()

  end

  defp validate_problem(%{d: d, l: l, u: u, fun: fun}) do
    unless is_integer(d) and d > 0 do
      raise ArgumentError, "`d` must be a positive integer"
    end

    unless is_function(fun, 1) do
      raise ArgumentError, "`fun` must be a valid function of the form `Nx.Tensor.t() -> float()`"
    end

    unless Nx.axis_size(l, 0) == d and Nx.axis_size(u, 0) == d do
      raise ArgumentError, "The dimensions of `l` and `u` must match `d` in the problem struct"
    end
  end

  defp validate_problem(_) do
    raise ArgumentError, "Problem struct must include fields `d`, `l`, `u`, and `fun`"
  end

  @spec fitness_function(t()) :: t()
  defp fitness_function(wso) do

    # Process each row and compute the fitness results
    fitness_results =
      Enum.map(0..(wso.hyperparams.n - 1), fn i ->
        wso.w
        |> Nx.slice([i, 0], [1, Nx.axis_size(wso.w, 1)])
        |> Nx.squeeze()
        |> wso.problem.fun.()
      end)
```

```elixir
    |> Nx.tensor()

    # Update the struct with computed fitness results
    %{wso | fitness_results: fitness_results}
  end

  @spec find_wgbestk(t()) :: t()
  defp find_wgbestk(wso) do
    gbestk = Nx.argmin(wso.fitness_results)
    gbestk_fitness_value = wso.fitness_results
      |> Nx.slice([gbestk], [1])
      |> Nx.reshape({})
      |> Nx.to_number()
    if gbestk_fitness_value < wso.best_g_fitness do
      %{wso | wgbestk: Nx.slice(wso.w, [gbestk, 0], [1, Nx.axis_size(wso.w, 1)]),
      best_g_fitness: gbestk_fitness_value}
    else
      wso
    end
  end

  @spec find_wbest(t()) :: t()
  defp find_wbest(wso) do
    # Create a mask for rows where fitness_results < best_fitness
    mask = Nx.less(wso.fitness_results, wso.best_fitness)

    # Expand the mask to align dimensions (add an axis to match {n, d})
    mask_expanded = Nx.new_axis(mask, -1)  # Shape: {n} -> {n, 1}

    # Broadcast the mask to match the shape of w and w_best
    mask_broadcasted = Nx.broadcast(mask_expanded, Nx.shape(wso.w)) # Shape: {n, 1} -> {n, d}

    # Perform conditional updates with Nx.select
    updated_w_best = Nx.select(mask_broadcasted, wso.w, wso.w_best)
    updated_best_fitness = Nx.select(mask, wso.fitness_results, wso.best_fitness)

    # Return the updated struct
    %{wso |
      w_best: updated_w_best,
```

```
      best_fitness: updated_best_fitness}
  end

  @spec movement_speed_towards_prey(t()) :: t()
  defp movement_speed_towards_prey(wso) do

     {c1, new_key} = Nx.Random.uniform(wso.key, shape: {wso.hyperparams.n, 1})
     {c2, new_key} = Nx.Random.uniform(new_key, shape: {wso.hyperparams.n, 1})

     {rand, new_key} = Nx.Random.uniform(new_key, 0.0, wso.hyperparams.n, shape: {wso.hyperparams.n})

     nu = Nx.floor(rand) |> Nx.as_type(:s64)
     selected_wbest = Nx.take(wso.w_best, nu, axis: 0)

     new_v = Nx.multiply(wso.hyperparams.mu, (wso.v
          |> Nx.add(wso.p1
            |> Nx.multiply(c1)
            |> Nx.multiply(Nx.subtract(wso.w_best, wso.w)))
          |> Nx.add(wso.p2
            |> Nx.multiply(c2)
            |> Nx.multiply(Nx.subtract(selected_wbest, wso.w)))
        ))

     %{wso |
      v: new_v,
      key: new_key}

  end

  @spec movement_speed_towards_optimal_prey(t()) :: t()
  defp movement_speed_towards_optimal_prey(wso) do
    rand = wso.hyperparams.rand_fun.()
    mv = 1 / (wso.hyperparams.a0 +
      :math.exp( (wso.max_iterations/2.0 - wso.k)/ wso.hyperparams.a1 ))

      w_new = case rand < mv do
      true ->
        a = wso.w
```

```
        |> Nx.subtract(Nx.broadcast(wso.problem.u, {wso.hyperparams.n, wso.problem.d}))
        |> Nx.greater(0)
        |> Nx.select(0, 1)

      b = wso.w
        |> Nx.subtract(Nx.broadcast(wso.problem.l, {wso.hyperparams.n, wso.problem.d}))
        |> Nx.less(0)
        |> Nx.select(0, 1)

      w0 = Nx.logical_and(a, b)
      # NOT XOR (a, b) = AND (NOT a, NOT a) note that a and b cannot both be 1

      wso.w
         |> Nx.multiply(w0)
         |> Nx.add(Nx.multiply(wso.problem.u, a))
         |> Nx.add(Nx.multiply(wso.problem.l, b))

    false -> wso.w |> Nx.add(Nx.divide(wso.v, wso.hyperparams.f))
  end
  %{wso | w: w_new}
end


@spec update_masked_indices_towards_the_best_white_shark(t(), Nx.Tensor.t(), integer()) :: t()
defp update_masked_indices_towards_the_best_white_shark(wso, indices, no_updates) do
    {r1_masked, new_key} = Nx.Random.uniform(wso.key, shape: {no_updates, 1})
    {r2_masked, new_key} = Nx.Random.uniform(new_key, shape: {no_updates, 1})

    w_bestk_masked = Nx.take(wso.w_best, indices, axis: 0)
    w_masked       = Nx.take(wso.w, indices, axis: 0)

    {rand_masked, new_key} = Nx.Random.uniform(new_key, shape: {no_updates, wso.problem.d})

    d_masked = Nx.abs(Nx.multiply(rand_masked, Nx.subtract(w_bestk_masked, w_masked)))
```

```elixir

      {rand, new_key} = Nx.Random.uniform(new_key, 0.0, 2.0, shape: {no_updates, wso.problem.d})

      update_masked = w_bestk_masked
        |> Nx.add(Nx.multiply(Nx.multiply(r1_masked, d_masked),
                              Nx.sign(Nx.subtract(r2_masked, 0.5))))
        |> Nx.add(w_masked)
        |> Nx.divide(rand)
        |> Nx.subtract(w_masked)

      w_new = Nx.indexed_add(wso.w, Nx.new_axis(indices, -1), update_masked, axes: [0])

      %{wso | key: new_key, w: w_new}
  end

  @spec indices_where_one(Nx.Tensor.t()) :: Nx.Tensor.t()
  def indices_where_one(tensor) do
    tensor
    |> Nx.to_flat_list()
    |> Enum.with_index()
    |> Enum.filter(fn {value, _index} -> value == 1 end)
    |> Enum.map(fn {_value, index} -> index end)
    |> Nx.tensor()
  end

  @spec movement_towards_the_best_white_shark(t()) :: t()
  defp movement_towards_the_best_white_shark(wso) do
    ss = abs(1 - :math.exp( -wso.hyperparams.a2 * wso.k / wso.max_iterations))

    {r3, new_key} = Nx.Random.uniform(wso.key, shape: {wso.hyperparams.n})
    mask = Nx.less(r3, Nx.tensor(ss))

    if Nx.to_number(Nx.all(Nx.logical_not(mask))) == 1 do
        wso
    else
      indices = Nx.greater(mask, Nx.tensor([0]))
        |> indices_where_one()
      no_updates = elem(Nx.shape(indices), 0)
```

```
      update_masked_indices_towards_the_best_white_shark(%{wso |
       ↪  key: new_key}, indices, no_updates)

    end
  end

  @spec iteration(t()) :: t()
  defp iteration(wso) do
    if not wso.verbose do
      IO.write("Iteration ")
      IO.write(inspect(wso.k))
      IO.write(" curr_best: ")
      IO.write(wso.best_g_fitness)
      IO.write(" at ")
      IO.inspect(wso.wgbestk |> Nx.to_flat_list())
    end

    case wso.k < wso.max_iterations do
      true ->
        wso
        |> compute_ps()
        |> movement_speed_towards_prey()
        |> movement_speed_towards_optimal_prey()
        |> movement_towards_the_best_white_shark()
        |> fitness_function()
        |> find_wgbestk()
        |> find_wbest()
        |> (fn map -> Map.update!(map, :k, &(&1 + 1)) end).()
        |> iteration()
      false -> wso |> find_wgbestk() |> find_wbest()
    end
  end

  @doc """
  Executes the White Shark Optimization (WSO) algorithm on the given
   ↪  `WhiteSharkOptimizer` struct and returns the optimized
   ↪  results.

  ### Parameters:
  - `wso`: A `WhiteSharkOptimizer` struct, already initialized with
   ↪  the problem, hyperparameters, and optional configuration
   ↪  values. The struct should also have initial positions (`w`),
   ↪  velocities (`v`), and other necessary fields set.
```

```
320
321   ### Returns:
322   - An updated `WhiteSharkOptimizer` struct with:
323     - `wgbestk`: The global best position found by the algorithm.
324     - `best_g_fitness`: The fitness value of the global best solution.
325     - `w_best`: The personal best positions for each individual solution in the population.
326     - `best_fitness`: The fitness values corresponding to the personal best positions.
327
328   """
329   @spec run(t()) :: t()
330   def run(wso) do
331     wso
332     |> fitness_function()
333     |> find_wgbestk()
334     |> find_wbest()
335     |> iteration()
336
337   end
338
339 end
```

3.3 Dyskusja

Implementacja pozwala na sprecyzowanie wymiaru d problemu. Wektorów u oraz l o wymiarach d które kodują dolną i górną granice przestrzeni poszukiwań oraz funkcji

Ruch do najlepszego żarłacza Zdecydowanie najciekawsza i najbardziej nietrywialna część algorytmu pod względem implementacji w Nx. W () podano której linii implementacji dotyczy się ten komentarz. Ruch do najlepszego żarłacza składa się z

- Wybieramy losowo indeksy które będziemy aktualizować (W kolejnych iteracjach większa jest szansa na działanie tego mechanizmu) (274 - 281)

- Wybieramy rekiny o wylosowanych indeksach używając funkcji Nx.take(). Otrzymujemy macierz liczba-rekininów-do-aktualizacji X wymiar-problemu (239-240)

- Obliczamy aktualizacje tylko dla macierzy stworzonej powyżej (248-253)
- Zmodyfikowane osobliki dodajemy do oryginalnej macierzy i zwracamy wynik (255-257)

Plus takiej implementacji: Aktualizacji rekinów może być bardzo mało. Nie jest łatwo zapisać aktualizacje w sposób wektorowy Minus: Musimy wybierać indeksy z macierzy używając Nx.take

Alternatywa Zamiast wybierać indeksy z macierzy i konstruować nową macierz można rozważyć czy nie aktualizować macierzy używając tylko obliczeń równoległych (tzn. bez wybierania indeksów używając maski). Ponieważ przez większość iteracji algorytmu wykonywana jest wersja gdzie bardzo mała (lub zerowa) populacja osobników jest w ten sposób aktualizowana wybraliśmy wersje z wybieraniem indeksów ale testy które porównywałyby obie implementacje nie były przeprowadzane

3.4 Przykład wywołania

Poniżej przedstawiono kod do wywołania przykładowo funkcji Rosenbrock w 2 wymiarach

```
defmodule Rosenbrock2D do

  def evaluate(x, y, a \\ 1, b \\ 100) do
    (a - x) * (a - x) +
      b * (y - x * x) * (y - x * x)
  end

  def evaluate_nx(u \\ Nx.tensor([1.0, 1.0], type: {:f, 32}), a \\ 1.0, b \\ 100.0) do
    # Split u into x, y components
    [x, y] = Nx.to_flat_list(u)

    # Compute the Rosenbrock function for 2D
    evaluate(x + :math.exp(0.5), y + :math.pi() + 1, a, b)
  end

end


hyperparams = Hyperparameters.new(%{n: 100})
```

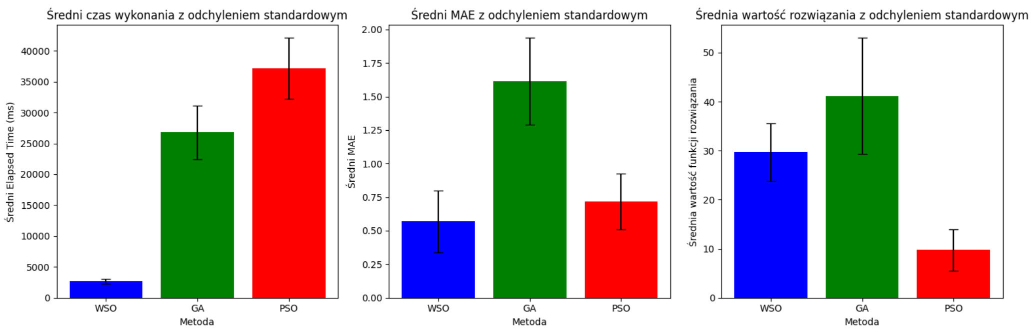
```
problem = Problem.new(%{d: 2, name: "Rosenbrock2D", fun: &Rosenbrock2D.evaluate_nx/1, minimize: true})
wso = WhiteSharkOptimizer.new(problem, hyperparams, %{verbose: false, key: Nx.Random.key(12), max_iterations: max_iterations})
wso = WhiteSharkOptimizer.run(wso)

IO.write("WSO SOLUTION: #{wso.best_g_fitness} at #{wso.wgbestk |> Nx.to_list() |> List.flatten() |> Enum.map(&Float.to_string/1) |> Enum.join(" ") }")


```

3.5 Porównanie algorytmów dla funkcji Rastrigin w 10 wymiarach

Populacja wynosiła 100 osobników. Liczba iteracji 100. Testowana funkcja to minimalizacja funkcji Rastrigin w 10 wymiarach. Zastosowano przesunięcie aby minimum nie było w punkcie x = 0 tylko w innnym potencjalnie trudniejszym do znalezenia. Porównane algorytmy to WSO (White Shark Optimizer), GA (Genetic Algorithm - Algorytm genetyczny) i PSO (Particle Swarm Optimization). Dla WSO zastosowano domyślne parametry sugerowane przez autorów. Dla GA: stopień selekcji: 0.7, stopień krzyżowania: 0.8, stopień mutacji: 0.02. Dla PSO: współczynnik bezwładności: 0.5, c1 (współczynnik poznawczy) = 1.5, c2 (współczynnik społeczny) = 1.5 .



Rysunek 3: Porównanie algorytmów WSO, GA i PSO dla funkcji Rastrigin w 10 wymiarach.

4 Optymalizacja modelu sieci wodociągowej

Optymalizacja została przeprowadzona na udostępnionym przez prowadzącego

W celu poprawy zgodności modelu hydraulicznego z rzeczywistym zachowaniem systemu, przeprowadza się optymalizację parametrów modelu. W niniejszym przypadku optymalizacji poddaje się współczynniki chropowatości (*roughness coefficient*) poszczególnych rur w sieci.

4.1 Parametry optymalizacji

Parametrami podlegającymi optymalizacji są współczynniki chropowatości wszystkich rur w modelu EPANET. Ich odpowiedni dobór wpływa bezpośrednio na obliczenia strat ciśnienia i przepływu, a zatem również na ciśnienie w poszczególnych węzłach sieci.

4.2 Algorytm optymalizacyjny

Do optymalizacji zastosowano algorytm ewolucyjny White Shark Optimizer (WSO) – heurystyczną metodę inspirowaną strategią polowania rekina białego. Algorytm ten iteracyjnie przeszukuje przestrzeń parametrów w celu zminimalizowania zadanej funkcji celu, dostosowując wartości chropowatości dla uzyskania najlepszego dopasowania modelu do danych pomiarowych.

4.3 Funkcja celu

Funkcja celu f ma postać błędu średniokwadratowego (MSE) pomiędzy ciśnieniami wygenerowanymi przez symulację a ciśnieniami zmierzonymi w rzeczywistej sieci wodociągowej. Pomiar ciśnienia pochodzi z trzech punktów pomiarowych, których dane są zapisane w pliku tekstowym `P.txt` w postaci:

$$\texttt{ID, p}_{true}$$

gdzie `ID` to identyfikator węzła, a p_{true} to zmierzone ciśnienie. Wartość funkcji celu obliczana jest jako:

$$f = \frac{1}{n}\sum_{i=1}^{n}(p_i - p_i^{true})^2$$

gdzie p_i to ciśnienie w węźle i uzyskane z symulacji EPANET.

4.4 Symulacja EPANET jako źródło danych

Dla każdej iteracji algorytmu WSO generowany jest zestaw parametrów chropowatości, które następnie wprowadzane są do modelu EPANET. Model jest uruchamiany, a z wyników symulacji odczytywane są ciśnienia w wybranych węzłach. Dane te służą jako wejście do obliczenia błędu średniokwadratowego.

4.5 Schemat optymalizacji

Proces optymalizacji przebiega zgodnie z następującym schematem:

1. Inicjalizacja populacji współczynników chropowatości dla wszystkich rur.
2. Dla każdego osobnika w populacji:
   (a) Przypisanie wartości chropowatości do odpowiednich rur w modelu EPANET.
   (b) Uruchomienie symulacji i odczyt ciśnień w punktach pomiarowych.
   (c) Obliczenie wartości funkcji celu (MSE) względem danych z `P.txt`.
3. Aktualizacja populacji zgodnie z zasadami WSO (lokalne i globalne przeszukiwanie).
4. Powtórzenie kroków do momentu osiągnięcia kryterium stopu (np. liczba iteracji lub minimalny błąd).

Dzięki zastosowaniu algorytmu WSO możliwe jest uzyskanie zoptymalizowanych wartości chropowatości rur, które prowadzą do lepszej zgodności modelu EPANET z rzeczywistym działaniem systemu wodociągowego.

4.6 Środowisko wykonawcze

Symulacje optymalizacyjne zostały przeprowadzone na superkomputerze **Helios**, zainstalowanym w Akademickim Centrum Komputerowym Cyfronet AGH w Krakowie. System ten został uruchomiony w ramach projektu *EuroHPC PL* i oparty jest na architekturze **HPE Cray EX4000**.

Komponent	**Parametry**
Procesory (CPU)	75 264 rdzeni `AMD EPYC`, architektura Zen 4, pamięć RAM DDR5 o pojemności 200 TB
Akceleratory (GPU)	440 akceleratorów `NVIDIA Grace Hopper GH200`, HBM3, interfejs PCIe 5.0
Partycja interaktywna (INT)	24 akceleratory `NVIDIA H100`, lokalne dyski NVMe
Moc obliczeniowa	Maksymalna wydajność do 35 PFLOPS, do 1,8 EFLOPS w obliczeniach AI
System plików (magazyn danych)	`Lustre`: – scratch: 1,5 PB, przepustowość 1,8 TB/s – project: 16 PB, przepustowość 200 GB/s
Interkonekt	`Slingshot`, przepustowość 200 Gb/s
Chłodzenie	Chłodzenie cieczą CPU i GPU, odzysk ciepła do systemów zewnętrznych

Tabela 1: Parametry techniczne superkomputera Helios

4.6.1 Zastosowanie w symulacjach

Wykorzystano CPU do uruchamiania równoległych procesów optymalizacyjnych oraz do oceny funkcji celu poprzez uruchamianie symulacji EPANET. Wydzielona przestrzeń dyskowa scratch umożliwiała szybki dostęp do danych tymczasowych generowanych w trakcie symulacji hydraulicznych. Interkonekt Slingshot zapewniał niskie opóźnienia i wysoką przepustowość wymiany danych między węzłami.

5 Wyniki

Poniżej zaprezentowany zostały szczegółowe wyniki optymalizacji przeprowadzonej na omawianym wcześniej modelu, wraz z porównaniem do analogicznej optymalizacji przeprowadzonej algorytmem PSO (Particle Swarm Optimization).

5.1 Analiza wydajności algorytmu White Shark Optimizer (Python)

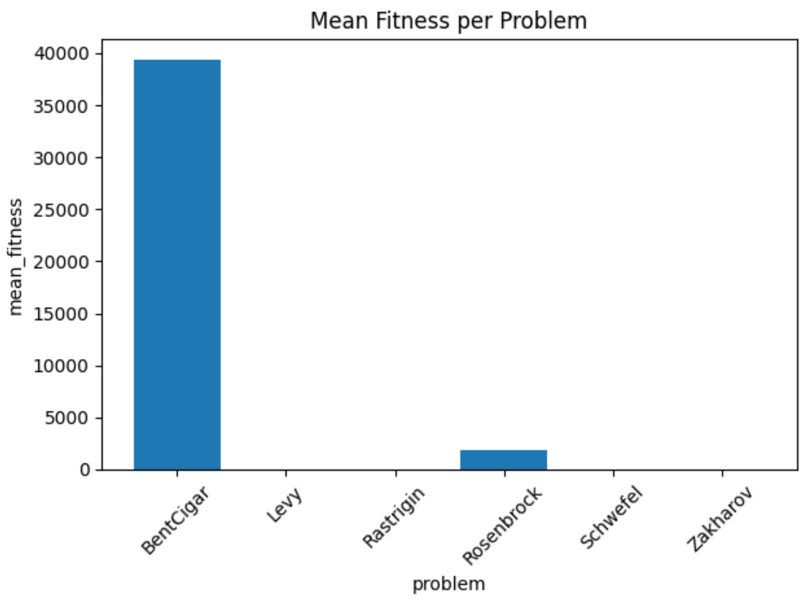
W celu oceny jakości algorytmu optymalizacyjnego *White Shark Optimizer* (WSO), przeprowadzono testy porównawcze na zestawie klasycznych funkcji benchmarkowych, takich jak: Rosenbrock, Rastrigin, Lévy, Zakharov, Schwefel oraz Bent Cigar. Każdy test był uruchamiany z tą samą konfiguracją parametrów: 100 iteracji, populacja 100 rekinów oraz wymiar przestrzeni poszukiwań 10. Poniższe testy dotyczą implementacji algorytmu WSO w Pythonie.

Dla każdej funkcji obliczano następujące statystyki:

- Średnią wartość funkcji celu osiągniętą przez WSO,
- Odchylenie standardowe wartości funkcji celu,
- Minimalną oraz maksymalną wartość funkcji celu,
- Średni czas obliczeń dla pojedynczego uruchomienia (w milisekundach).

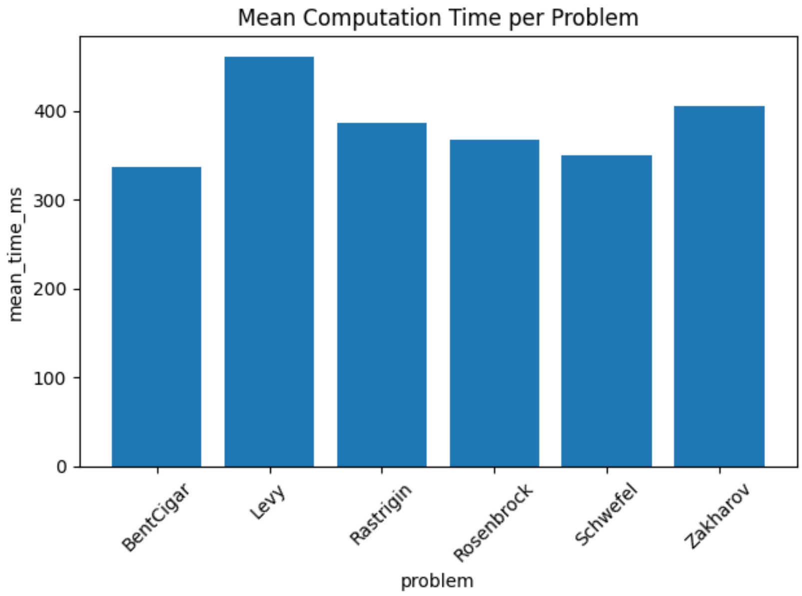
Dane zostały następnie zebrane i przeanalizowane za pomocą skryptu napisanego w języku Python z użyciem bibliotek `pandas` oraz `matplotlib`. Poniżej przedstawiono wybrane wykresy wizualizujące wyniki.

Średnia wartość funkcji celu w zależności od problemu:



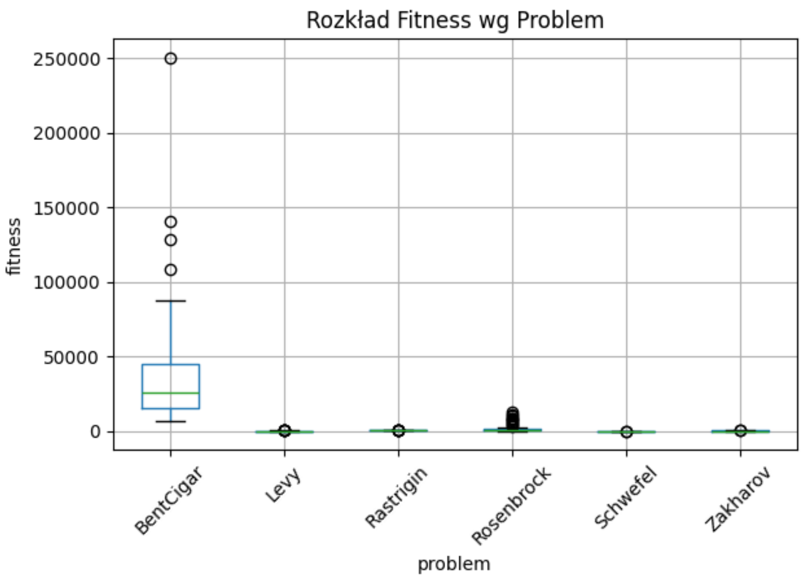
Rysunek 4: Średnia wartość funkcji celu osiągana przez WSO dla różnych problemów.

Średni czas obliczeń w zależności od problemu:



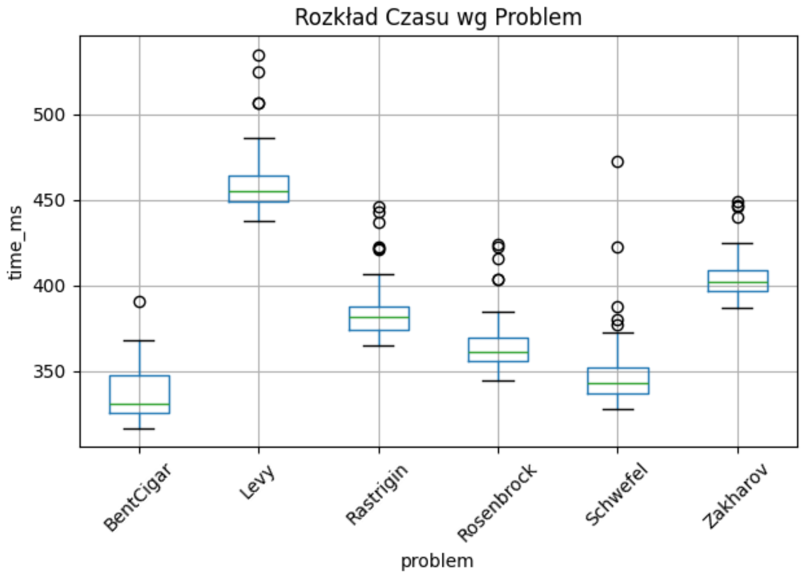
Rysunek 5: Średni czas wykonania pojedynczego uruchomienia WSO w milisekundach.

Rozkład wartości funkcji celu:



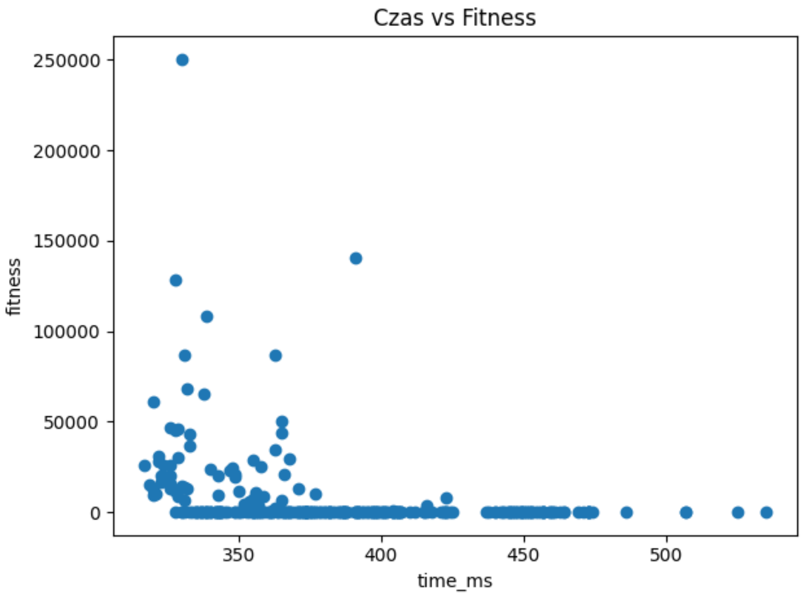
Rysunek 6: Wykres pudełkowy wartości funkcji celu osiąganych przez WSO.

Rozkład czasów wykonania:



Rysunek 7: Wykres pudełkowy czasów obliczeń dla różnych funkcji.

Zależność między czasem a jakością rozwiązania:

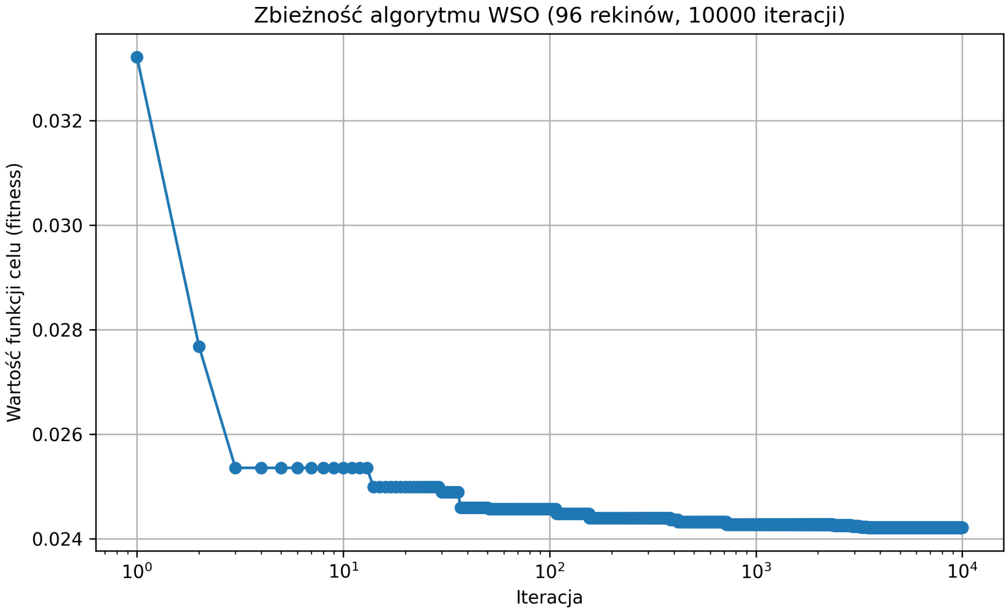


Rysunek 8: Zależność między czasem wykonania a jakością (fitness) rozwiązania.

Analiza wykazała, że WSO osiąga wysoką skuteczność optymalizacji przy relatywnie niskim czasie obliczeń, szczególnie w przypadku funkcji Schwefel oraz Zakharov, co potwierdzają bardzo niskie wartości funkcji celu oraz mała rozrzutność wyników. W funkcji Bent Cigar zauważono większy rozrzut wyników, co może świadczyć o trudności algorytmu w optymalizacji tej konkretnej funkcji.

5.2 Najlepszy model

Najlepszą wartość fitness uzyskano po optymalizacji algorytmem White Shark Optimizer z parametrami: populacja $N = 96$ rekinów oraz $T = 10000$ iteracji. Poniżej przedstawiono fragment przebiegu wartości funkcji celu (fitness) w kolejnych iteracjach. Początkowa wartość (dla niezmienionego modelu startowego) wynosiła 0,0332, a pierwsze istotne zmiany obserwowano już po kilkunastu iteracjach. Z czasem algorytm stopniowo zbiegał do wartości końcowej na poziomie około 0,0242, co świadczy o skuteczności mechanizmów eksploracji i eksploatacji w algorytmie WSO.



Rysunek 9: Zbieżność algorytmu WSO dla populacji 96 rekinów i 10000 iteracji

5.2.1 Porównanie parametrów po optymalizacji

Poniżej zaprezentowano porównanie wartości współczynnika chropowatości (roughness) dla pierwszych 10 rur w modelu EPANET, przed i po optymalizacji. Zmiany te obrazują wpływ działania algorytmu na kalibrację parametrów hydraulicznych sieci wodociągowej.

ID Rury	Chropowatość - przed	Chropowatość - po
p1	0.0500	0.3537
p2	0.0500	0.0252
p4	0.0500	4.7768
p5	5.0000	6.2855
p6	0.0500	9.9362
p7	0.0500	6.6795
p8	5.0000	4.0833
p9	5.0000	8.4314
p10	0.0500	5.6473

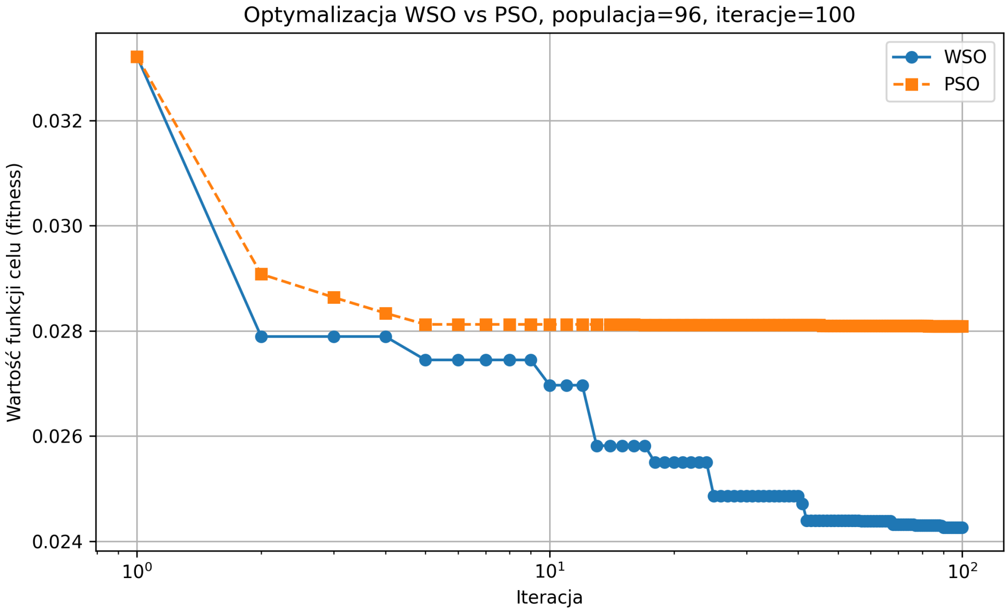
Tabela 2: Zmiana współczynników chropowatości dla pierwszych 10 rur przed i po optymalizacji algorytmem WSO

Zauważyć można, że algorytm znacząco dostosował parametry chropowatości, w tym również w kierunku zwiększenia ich wartości w rurach o niskim przepływie. Świadczy to o możliwości dostosowywania się algorytmu do lokalnych warunków hydraulicznych modelu w celu minimalizacji funkcji celu.

5.3 Porównanie zbieżności algorytmów WSO i PSO

W niniejszej sekcji przedstawiono porównanie efektywności algorytmu Whale Swarm Optimization (WSO) z Particle Swarm Optimization (PSO) w kontekście rozważanego problemu kalibracji sieci wodociągowej. Oba algorytmy zostały uruchomione w kilku konfiguracjach: populacja 46 lub 96 osobników (rekini / cząstki) oraz maksymalna liczba iteracji wynosząca 100.

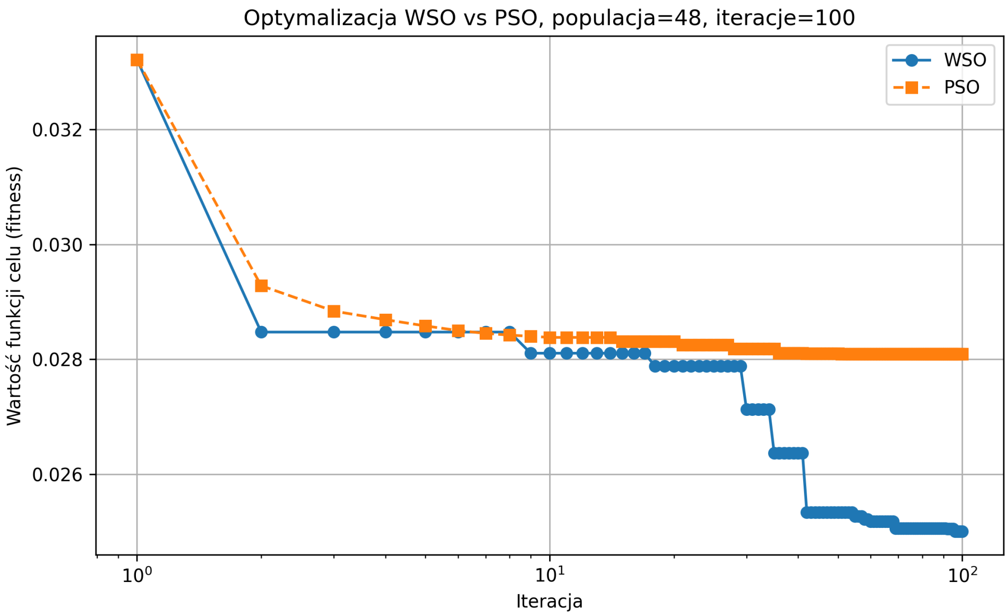
Celem porównania jest ocena szybkości zbieżności oraz jakości osiąganych wartości funkcji celu przez każdy z algorytmów. Poniższy wykres prezentuje średnie wartości funkcji celu w kolejnych iteracjach:



Rysunek 10: Porównanie zbieżności algorytmów WSO i PSO (populacja = 96, iteracje = 100)

Z analizy wykresu można będzie wyciągnąć wnioski dotyczące skuteczności poszczególnych strategii optymalizacyjnych. Widzimy, iż algorytm WSO stopniowo poprawia wartość błędu, znajdując rozwiązania o coraz lepszej (mniejszej) wartości fitness. Z kolei algorytm PSO, po postępach w początkowych iteracjach, zdaje się utykać w minimum lokalnym i nie poprawia w znaczący sposób wyniku na przestrzeni kolejnych iteracji.

Analogiczne porównanie, ale dla 48 rekinów / cząstek i 100 iteracji:



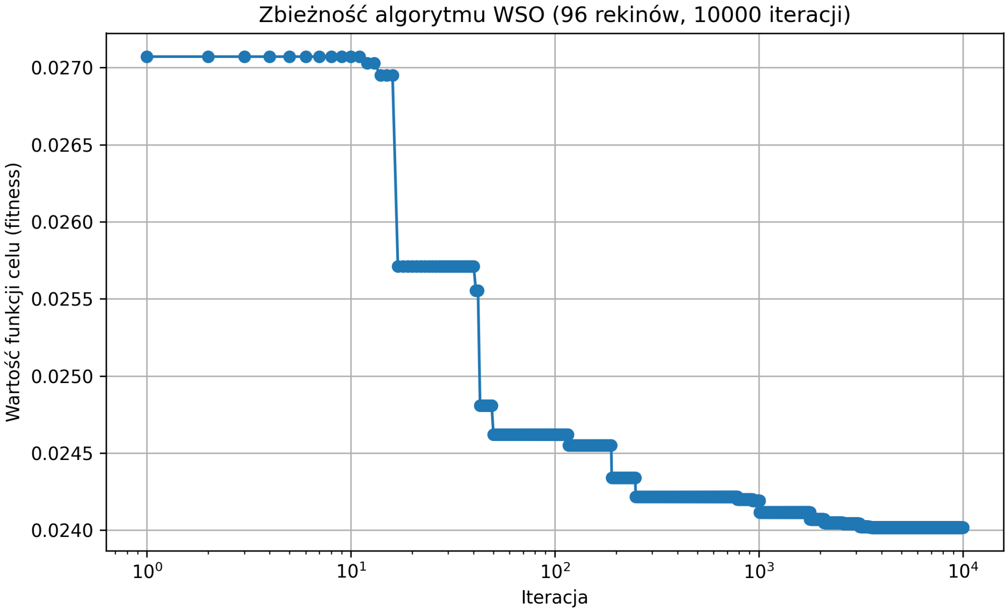
Rysunek 11: Porównanie zbieżności algorytmów WSO i PSO (populacja = 48, iteracje = 100)

Widzimy podobną sytuację jak przy większej liczebności populacji. Oba algorytmy wykazują szybki spadek wartości funkcji celu w początkowych iteracjach (do około 5. iteracji). Następnie PSO bardzo szybko osiąga stagnację – od około 15. iteracji wartość funkcji celu przestaje się poprawiać i pozostaje na poziomie około 0,0281. Z kolei WSO nadal stopniowo poprawia wynik, osiągając ostatecznie wartość niższą niż PSO. Ostatecznie WSO znajduje lepsze rozwiązanie globalne, podczas gdy PSO zatrzymuje się wcześniej w minimum lokalnym. Wskazuje to na wyższą zdolność eksploracyjną algorytmu WSO w porównaniu do PSO.

5.4 Test zbieżności algorytmu WSO po zaburzeniu modelu

W celu przetestowania stabilności i zdolności algorytmu *White Shark Optimizer* (WSO) do powrotu do jakościowych rozwiązań, dokonano celowego zaburzenia współczynników szorstkości (roughness) w uprzednio zoptymalizowanym modelu. Do wartości tych dodano losowy szum o rozkładzie normalnym $N(0, \sigma^2)$, a następnie przeprowadzono ponowną optymalizację modelu przy populacji 96 osobników i liczbie iteracji równej 10000.

Na rysunku 12 przedstawiono przebieg zbieżności wartości funkcji celu w kolejnych iteracjach algorytmu.



Rysunek 12: Zbieżność funkcji celu dla algorytmu WSO po zaburzeniu modelu

Tabela 3 zawiera porównanie wartości współczynnika roughness dla każdego odcinka (oznaczonego jako p) po pierwszej oraz drugiej optymalizacji. Dodatkowo uwzględniono różnicę Δ pomiędzy tymi wartościami. W wielu przypadkach widoczna jest zdolność algorytmu do korekcji zaburzeń i powrotu do podobnych wartości parametrów.

Tabela 3: Porównanie wartości roughness przed i po ponownej optymalizacji

Odcinek	Roughness (1. optymalizacja)	Roughness (2. optymalizacja)	Δ
p1	0.3537	0.3541	$+0.0004$
p2	0.0252	0.0206	-0.0046
p4	4.7768	9.8303	$+5.0535$
p5	6.2855	0.8711	-5.4144
p6	9.9362	1.9884	-7.9478
p7	6.6795	9.2934	$+2.6139$
p8	4.0833	11.0315	$+6.9482$
p9	8.4314	2.6184	-5.8130
p10	5.6473	7.9528	$+2.3055$

Widzimi, iż współczynniki dla pierwszych dwóch rur (p1 i p2) praktycznie nie zmieniły się, jednak dla innych zaszły znaczne zmiany. Może to sugerować, iż pewne rury są ważniejsze dla końcowego wyniku pochodzącego z symulacji od innych, których chropowatość nie wpływa znacząco na błąd obliczany w wyniku symulacji.