White Shark Optimizer & EPANET – pierwsze uruchomienie

Igor Swat Rafał Piwowar

1 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


    def optimize(self, problem: Problem, no_sharks: int = 10, steps: 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)
        '''

        # 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

        # Step 2 - Evaluate initial population fitness
        fitness = np.array([problem.evaluate(pos) for pos in W])
        fitness_min = np.min(fitness)
        W_best = W.copy()
        W_gbest = W[np.argmin(fitness)]
```

```
36
37          # Main WSO loop
38          for k in range(1, steps + 1):
39              p1 = self.p_max + (self.p_max - self.p_min) * np.exp(-(4
                ↪  * k / steps)**2)
40              p2 = self.p_min + (self.p_max - self.p_min) * np.exp(-(4
                ↪  * k / steps)**2)
41              mv = 1 / (self.a0 + np.exp((steps / 2.0 - k) / self.a1))
42              s_s = abs(1 - np.exp(-self.a2 * k / steps))
43
44              # Step 3 - update shark velocities
45              nu = np.random.randint(0, no_sharks, no_sharks)
46              for i in range(no_sharks):
47                  c1, c2 = random.random(), random.random()
48                  v[i, :] = self.mu * (
49                      v[i, :] +
50                      p1 * c1 * (W_gbest - W[i, :]) +
51                      p2 * c2 * (W_best[nu[i], :] - W[i, :])
52                  )
53
54              # Step 4 - update positions
55              f = self.f_min + (self.f_max - self.f_min) / (self.f_max
                ↪  + self.f_min)
56              for i in range(no_sharks):
57                  out_high = W[i, :] > problem.ub
58                  out_low  = W[i, :] < problem.lb
59                  w0 = np.logical_xor(out_high, out_low)
60                  if random.random() < mv:
61                      W[i][w0] = problem.ub[w0] * out_high[w0] +
                        ↪  problem.lb[w0] * out_low[w0]
62                  else:
63                      W[i, :] += v[i, :] / f
64
65              # Step 5 - school movement update
66              for i in range(no_sharks):
67                  if random.random() <= s_s:
68                      D = np.abs(np.random.rand() * (W_gbest - W[i,
                        ↪  :]))
69                      sgn = np.sign(np.random.rand(problem.dim) - 0.5)
70                      tmp = W_gbest + np.random.rand(problem.dim) * D
                        ↪  * sgn
71                      W[i, :] = tmp if i == 0 else (W[i, :] + tmp) /
                        ↪  (2 * random.random())
```

```
72
73              # Step 6 - evaluate and update best positions
74              for i in range(no_sharks):
75                  if np.all((W[i, :] >= problem.lb) & (W[i, :] <=
                    ↪  problem.ub)):
76                      fit = problem.evaluate(W[i, :])
77                      if fit < fitness[i]:
78                          W_best[i, :] = W[i, :]
79                          fitness[i] = fit
80                      if fitness[i] < fitness_min:
81                          fitness_min = fitness[i]
82                          W_gbest = W_best[i].copy()
83
84          return W_gbest, fitness_min
```

1.1 Hiperparametry

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

- rozmiar populacji,
- liczbę iteracji.

1.2 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.

1.3 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.

1.4 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.

1.5 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 $[-5.12, 5.12]^2$).

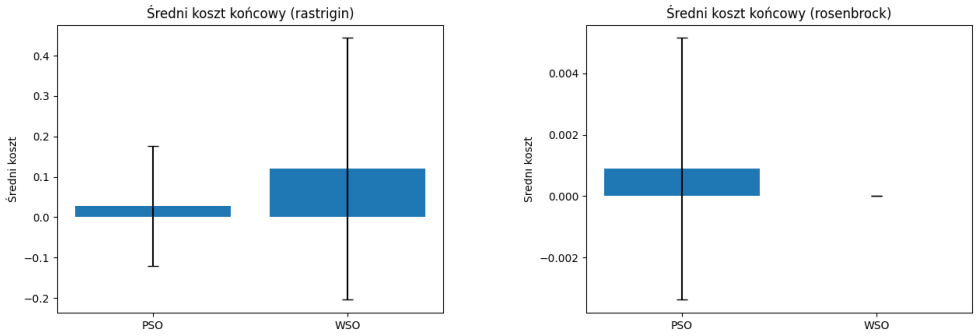
Dla każdego problemu uruchomiono $N = 200$ niezależnych prób. Parametry algorytmów dobrano następująco:

- 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 $T = 100$.

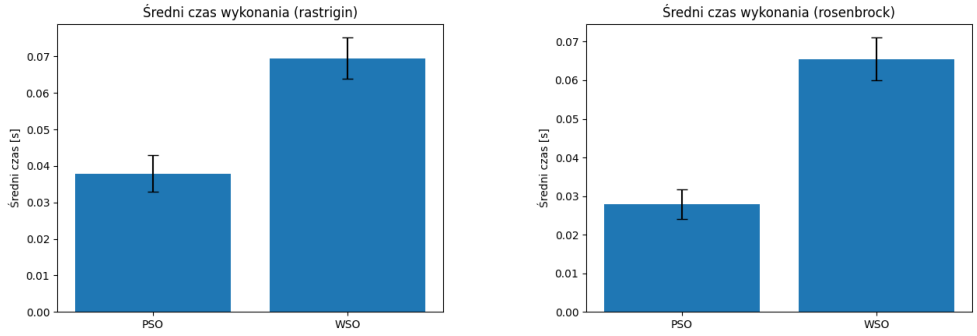
Jako miary jakości porównania przyjęto:

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 szyb-

szy 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 White Shark Optimizer – implementacja (Elixir)

2.1 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.

2.2 Implementacja

2.2.1 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
        }
```

```elixir

  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
```

2.2.2 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

```

```
44
45    @spec compute_ps(t()) :: t()
46    defp compute_ps(wso) do
47      p_min = wso.hyperparams.p_min
48      p_max = wso.hyperparams.p_max
49      p1 = p_max + (p_max - p_min) * :math.exp(-:math.pow( 4 * wso.k
        ↪  / wso.max_iterations, 2))
50      p2 = p_min + (p_max - p_min) * :math.exp(-:math.pow( 4 * wso.k
        ↪  / wso.max_iterations, 2))
51      %{wso | p1: p1, p2: p2}
52    end
53
54    @doc """
55    Initializes a new instance of the WhiteSharkOptimizer struct with
      ↪  the provided problem, hyperparameters, and optional
      ↪  configuration.
56
57    ### Parameters:
58    - `problem`: A struct defining the optimization problem to be
      ↪  solved. It must include the following fields:
59      - `d`: Dimensionality of the problem.
60      - `l`: Lower bounds for the search space as an Nx.Tensor of
        ↪  shape `{d}`.
61      - `u`: Upper bounds for the search space as an Nx.Tensor of
        ↪  shape `{d}`.
62      - `fun`: A fitness function of the form `(Nx.Tensor.t() ->
        ↪  float())`, used to evaluate the quality of solutions.
63      - The dimensionality (`d`) must match the bounds (`l` and `u`).
64    - `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.
65    - `opts`: A map of optional configuration values. These can
      ↪  include:
66      - `key`: Random key for generating initial population and
        ↪  randomness. Defaults to a key generated by
        ↪  `Nx.Random.key(0)` if not provided.
67      - `verbose`: If `false`, the best solution will be printed every
        ↪  iteration. Defaults to `true`.
68
69    ### Raises:
70    - `ArgumentError`: Raised in the following cases:
```

```
71      - `problem` is `nil`, as the optimizer cannot operate without a
        ↪  defined problem.
72      - `problem` is missing required fields (`d`, `l`, `u`, or
        ↪  `fun`).
73      - The dimensions of the bounds (`l` and `u`) do not match the
        ↪  dimensionality (`d`).
74      - The `fun` field is not a valid function.
75
76    ### Returns:
77    - A `WhiteSharkOptimizer` struct initialized with computed fields.
78    """
79    @spec new(Problem.t(), Hyperparameters.t(), map()) :: t()
80    def new(%{d: _, l: _, u: _, fun: _} = problem, hyperparams, opts
      ↪  \\ %{}) do
81      validate_problem(problem)
82
83      key = Map.get(opts, :key, Nx.Random.key(0))
84
85      {random_tensor, key} = Nx.Random.uniform(key, shape:
        ↪  {hyperparams.n, problem.d})
86      w_initial = random_tensor
87            |> Nx.multiply(Nx.subtract(problem.u, problem.l))
88            |> Nx.add(problem.l)
89      computed_fields = %{
90        w: w_initial,
91        v: Nx.broadcast(0, {hyperparams.n, problem.d}),
92        key: key,
93        problem: problem,
94        hyperparams: hyperparams,
95        w_best: w_initial,
96        best_fitness: Nx.broadcast(Nx.Constants.infinity(),
          ↪  {hyperparams.n})
97      }
98
99      %__MODULE__{}
100     |> struct(Map.merge(computed_fields, opts))
101     |> compute_ps()
102
103   end
104
105   defp validate_problem(%{d: d, l: l, u: u, fun: fun}) do
106     unless is_integer(d) and d > 0 do
107       raise ArgumentError, "`d` must be a positive integer"
```

```
108       end
109
110       unless is_function(fun, 1) do
111         raise ArgumentError, "`fun` must be a valid function of the
          ↪  form `Nx.Tensor.t() -> float()`"
112       end
113
114       unless Nx.axis_size(l, 0) == d and Nx.axis_size(u, 0) == d do
115         raise ArgumentError, "The dimensions of `l` and `u` must match
          ↪  `d` in the problem struct"
116       end
117     end
118
119     defp validate_problem(_) do
120       raise ArgumentError, "Problem struct must include fields `d`,
        ↪  `l`, `u`, and `fun`"
121     end
122
123     @spec fitness_function(t()) :: t()
124     defp fitness_function(wso) do
125
126       # Process each row and compute the fitness results
127       fitness_results =
128         Enum.map(0..(wso.hyperparams.n - 1), fn i ->
129           wso.w
130           |> Nx.slice([i, 0], [1, Nx.axis_size(wso.w, 1)])
131           |> Nx.squeeze()
132           |> wso.problem.fun.()
133         end)
134       |> Nx.tensor()
135
136       # Update the struct with computed fitness results
137       %{wso | fitness_results: fitness_results}
138     end
139
140     @spec find_wgbestk(t()) :: t()
141     defp find_wgbestk(wso) do
142       gbestk = Nx.argmin(wso.fitness_results)
143       gbestk_fitness_value = wso.fitness_results
144         |> Nx.slice([gbestk], [1])
145         |> Nx.reshape({})
146         |> Nx.to_number()
147       if gbestk_fitness_value < wso.best_g_fitness do
```

```
148         %{wso | wgbestk: Nx.slice(wso.w, [gbestk, 0], [1,
            ↪  Nx.axis_size(wso.w, 1)]),
149         best_g_fitness: gbestk_fitness_value}
150       else
151         wso
152       end
153     end
154
155     @spec find_wbest(t()) :: t()
156     defp find_wbest(wso) do
157       # Create a mask for rows where fitness_results < best_fitness
158       mask = Nx.less(wso.fitness_results, wso.best_fitness)
159
160       # Expand the mask to align dimensions (add an axis to match {n,
            ↪  d})
161       mask_expanded = Nx.new_axis(mask, -1)  # Shape: {n} -> {n, 1}
162
163       # Broadcast the mask to match the shape of w and w_best
164       mask_broadcasted = Nx.broadcast(mask_expanded, Nx.shape(wso.w))
            ↪  # Shape: {n, 1} -> {n, d}
165
166       # Perform conditional updates with Nx.select
167       updated_w_best = Nx.select(mask_broadcasted, wso.w, wso.w_best)
168       updated_best_fitness = Nx.select(mask, wso.fitness_results,
            ↪  wso.best_fitness)
169
170       # Return the updated struct
171       %{wso |
172         w_best: updated_w_best,
173         best_fitness: updated_best_fitness}
174     end
175
176     @spec movement_speed_towards_prey(t()) :: t()
177     defp movement_speed_towards_prey(wso) do
178
179        {c1, new_key} = Nx.Random.uniform(wso.key, shape:
            ↪  {wso.hyperparams.n, 1})
180        {c2, new_key} = Nx.Random.uniform(new_key, shape:
            ↪  {wso.hyperparams.n, 1})
181
182        {rand, new_key} = Nx.Random.uniform(new_key, 0.0,
            ↪  wso.hyperparams.n, shape: {wso.hyperparams.n})
183
```

```
184       nu = Nx.floor(rand) |> Nx.as_type(:s64)
185       selected_wbest = Nx.take(wso.w_best, nu, axis: 0)
186
187       new_v = Nx.multiply(wso.hyperparams.mu, (wso.v
188             |> Nx.add(wso.p1
189               |> Nx.multiply(c1)
190               |> Nx.multiply(Nx.subtract(wso.w_best, wso.w)))
191             |> Nx.add(wso.p2
192               |> Nx.multiply(c2)
193               |> Nx.multiply(Nx.subtract(selected_wbest, wso.w)))
194           ))
195
196       %{wso |
197        v: new_v,
198        key: new_key}
199
200     end
201
202     @spec movement_speed_towards_optimal_prey(t()) :: t()
203     defp movement_speed_towards_optimal_prey(wso) do
204       rand = wso.hyperparams.rand_fun.()
205       mv = 1 / (wso.hyperparams.a0 +
206         :math.exp( (wso.max_iterations/2.0 - wso.k)/
          ↪  wso.hyperparams.a1 ))
207
208         w_new = case rand < mv do
209         true ->
210           a = wso.w
211             |> Nx.subtract(Nx.broadcast(wso.problem.u,
              ↪  {wso.hyperparams.n, wso.problem.d}))
212             |> Nx.greater(0)
213             |> Nx.select(0, 1)
214
215           b = wso.w
216             |> Nx.subtract(Nx.broadcast(wso.problem.l,
              ↪  {wso.hyperparams.n, wso.problem.d}))
217             |> Nx.less(0)
218             |> Nx.select(0, 1)
219
220           w0 = Nx.logical_and(a, b)
221           # NOT XOR (a, b) = AND (NOT a, NOT a) note that a and b
              ↪  cannot both be 1
222
```

```
        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)))

    {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}
```

```
258     end
259
260     @spec indices_where_one(Nx.Tensor.t()) :: Nx.Tensor.t()
261     def indices_where_one(tensor) do
262       tensor
263       |> Nx.to_flat_list()
264       |> Enum.with_index()
265       |> Enum.filter(fn {value, _index} -> value == 1 end)
266       |> Enum.map(fn {_value, index} -> index end)
267       |> Nx.tensor()
268     end
269
270     @spec movement_towards_the_best_white_shark(t()) :: t()
271     defp movement_towards_the_best_white_shark(wso) do
272       ss = abs(1 - :math.exp( -wso.hyperparams.a2 * wso.k /
          ↪  wso.max_iterations))
273
274       {r3, new_key} = Nx.Random.uniform(wso.key, shape:
          ↪  {wso.hyperparams.n})
275       mask = Nx.less(r3, Nx.tensor(ss))
276
277       if Nx.to_number(Nx.all(Nx.logical_not(mask))) == 1 do
278           wso
279       else
280         indices = Nx.greater(mask, Nx.tensor([0]))
281           |> indices_where_one()
282         no_updates = elem(Nx.shape(indices), 0)
283         update_masked_indices_towards_the_best_white_shark(%{wso |
          ↪  key: new_key}, indices, no_updates)
284
285       end
286     end
287
288     @spec iteration(t()) :: t()
289     defp iteration(wso) do
290       if not wso.verbose do
291         IO.write("Iteration ")
292         IO.write(inspect(wso.k))
293         IO.write(" curr_best: ")
294         IO.write(wso.best_g_fitness)
295         IO.write(" at ")
296         IO.inspect(wso.wgbestk |> Nx.to_flat_list())
297       end
```

```
298
299       case wso.k < wso.max_iterations do
300         true ->
301           wso
302           |> compute_ps()
303           |> movement_speed_towards_prey()
304           |> movement_speed_towards_optimal_prey()
305           |> movement_towards_the_best_white_shark()
306           |> fitness_function()
307           |> find_wgbestk()
308           |> find_wbest()
309           |> (fn map -> Map.update!(map, :k, &(&1 + 1)) end).()
310           |> iteration()
311         false -> wso |> find_wgbestk() |> find_wbest()
312       end
313     end
314
315     @doc """
316     Executes the White Shark Optimization (WSO) algorithm on the given
        ↪ `WhiteSharkOptimizer` struct and returns the optimized
        ↪ results.
317
318     ### Parameters:
319     - `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
```

2.3 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

2.4 Przykład wywołania

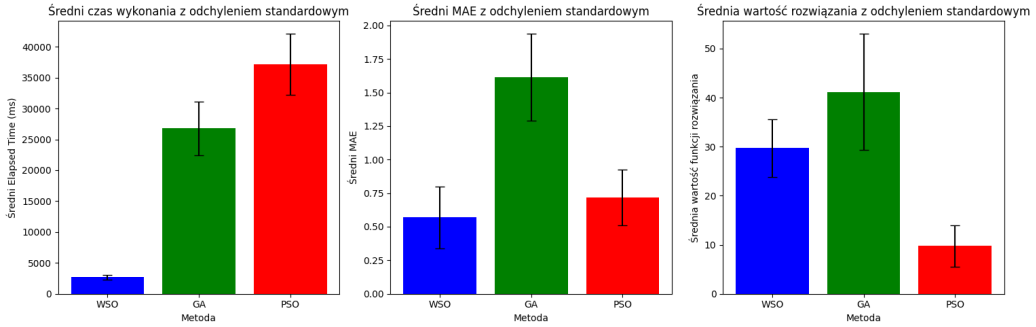
Poniżej przedstawiono kod do wywołania przykładowo funkcji Rosenbrock w 2 wymiarach

```
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(" ") }")


```

2.5 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.

3 Integracja z EPANET

W celu integracji algorytmu optymalizacyjnego ze środowiskiem EPANET, wykorzystujemy pakiet *Epyt*.

Kluczowym elementem procesu optymalizacji modelu sieci wodociągowej jest odpowiedni dobór funkcji straty (ang. *loss function*), która pełni rolę funkcji celu w algorytmie optymalizacyjnym. Funkcja ta musi jednoznacznie oceniać jakość każdego zaproponowanego rozwiązania, tj. konkretnego zestawu parametrów decyzyjnych (np. średnic rur), na podstawie wyników symulacji hydraulicznej.

W kontekście wykorzystania pakietu `Epyt`, każda ocena rozwiązania polega na:

1. Zastosowaniu danego zestawu parametrów do modelu za pomocą metod takich jak `setLinkValue()`.

2. Uruchomieniu symulacji hydraulicznej (`solveCompleteHydraulics()`), która oblicza wartości ciśnień, przepływów i innych wielkości w sieci dla zadanych warunków czasowych.

3. Pobranie wyników (`getComputedTimeSeries()`), a następnie wyliczenie wartości funkcji straty na ich podstawie.

Najczęściej stosowane funkcje straty mogą przyjmować różne formy, zależnie od celu projektu:

- **Suma kwadratów błędów ciśnienia**:

$$f(x) = \sum_{t \in T} \sum_{j \in \mathcal{N}} \left(p_j(t; x) - p_j^{\text{ref}}(t) \right)^2$$

  gdzie $p_j(t; x)$ to ciśnienie w węźle j w chwili t dla rozwiązań x, a $p_j^{\text{ref}}(t)$ to wartość referencyjna (np. minimalne wymagane ciśnienie).

- **Funkcje kosztowe** — mogą uwzględniać długość rur, koszty jednostkowe materiałów oraz ewentualne kary za niedotrzymanie ograniczeń hydraulicznych:

$$f(x) = \alpha \cdot Cost(x) + \beta \cdot Penalty(x)$$

- **Funkcje wielokryterialne** — łączące np. niezawodność, koszty i ciśnienie, stosując wagowanie lub metody Pareto.

W praktyce każda iteracja algorytmu optymalizacyjnego wiąże się z wykonaniem pełnej symulacji w Epyt. Oznacza to, że:

- Funkcja celu nie jest znana analitycznie ani różniczkowalna.
- Optymalizacja ma charakter czarnej skrzynki i wymaga algorytmów niegradientowych (np. metaheurystyk).
- Wydajność symulacji i równoległe przetwarzanie mogą znacząco wpłynąć na czas obliczeń.

Takie podejście — symulacja–ocena–aktualizacja — stanowi trzon podejścia optymalizacyjnego opartego na Epyt i może być łatwo dostosowane do różnych konfiguracji sieci i celów projektowych.

3.1 Przykład: integracja Epyt z algorytmem optymalizacyjnym

Poniżej przedstawiono przykładową implementację cyklu optymalizacji, w którym model sieci wodociągowej obsługiwany przez pakiet `Epyt` jest optymalizowany za pomocą metaheurystyki White Shark Optimizer (WSO). Kod pokazuje typową strukturę: inicjalizację modelu, definiowanie problemu optymalizacyjnego, wykonanie optymalizacji oraz analizę wyników.

```python
def epanet_wso_test(model_filepath: str) -> None:
    network = epanet(model_filepath)

    # Liczba zmiennych decyzyjnych (np. średnic rur)
    dim = network.getLinkPipeCount()
    lb = np.full(shape=(dim), fill_value=4)
    ub = np.full(shape=(dim), fill_value=20)

    # Definicja problemu optymalizacyjnego
    problem = EpanetProblem(dim, lb, ub, network, 24)

    # Konfiguracja algorytmu optymalizacyjnego
    optimizer = Optimizer()
    no_sharks = 50
    steps = 100

    # Uruchomienie optymalizacji
    diameters_best, loss_best = optimizer.optimize(
        problem, no_sharks=no_sharks, steps=steps
    )
    print("Optimal fitness:", loss_best)
    print("Optimal solution:", diameters_best, end="\n\n")

    # Symulacja hydrauliczna dla najlepszego rozwiązania
    network.solveCompleteHydraulics()
    results = network.getComputedTimeSeries()
    pd.DataFrame(results.Pressure).to_csv("pressure_24h.csv")

    # Zwolnienie zasobów
    network.unload()
```

Powyższa implementacja jest czysto ilustracyjna, a samo zdefiniowanie funkcji kosztu jak i uruchomienie symulacji na właściwym modelu odbędzie się w kolejnym etapie prac nad projektem.