Studia i Materiały Informatyki Stosowanej, Tom 13, Nr 1, 2021
str. 17-32

Porównanie wybranych algorytmów wilczego stada stosowanych w rozwiązaniach problemów optymalizacji

Belco Sangho[1]

[1] *Uniwersytet Kazimierza Wielkiego, Instytut Informatyki, Kopernika 1, 85-074 Bydgoszcz*

Streszczenie: *Algorytmy optymalizacyjne zyskały uznanie jako szybki i konsekwentny sposób rozwiązywania problemów optymalizacyjnych. W ostatnim czasie wilki są coraz częściej wykorzystywane jako inspiracja do tworzenia algorytmów, jak i w projektach używających tych algorytmów. W niniejszej pracy opisano sześć wybranych algorytmów. Następnie zaimplementowano je w języku R i porównano z pomocą sześciu funkcji porównujących, tzw. benchmarków. Wyniki trzydziestu testów na każdej z funkcji zaprezentowano za pomocą średniego wyniku, odchylenia standardowego wyniku, średniego czasu oraz odchylenia standardowego czasu. Dodatkowo zaprezentowano wykres zbieżności na dwóch z funkcji porównujących. Uzyskane wyniki algorytmów często różniły się od tych zaprezentowanych w publikacjach, jednak skuteczność części z nich była lepsza bądź porównywalna z PSO[1], DE[2] i GA[3]. Najlepszym wilczym algorytmem okazał się Grey Wolf Optimizer[4].*

Słowa kluczowe: *Optymalizacja, Algorytmy rojowe, Algorytmy wilcze, Wilki, Funkcje porównujące*

Comparison of selected wolf pack algorithms used in solving optimization problems

Abstract: *Optimization algorithms have gained recognition as a fast and consistent way to solve optimization problems. Recently, wolves have been increasingly used as inspiration for algorithms as well as in projects using these algorithms. In this paper, six selected algorithms are described. They were then implemented in R and compared using six comparison functions, called benchmarks. The results of thirty tests on each function were presented by mean score, standard deviation of the score, mean time and standard deviation of the time. Additionally, a convergence plot on two of the benchmark functions was presented. The algorithm results obtained often differed from those presented in the publications, but the performance of some of the algorithms was better or comparable to PSO[1], DE[2], and GA[3]. The best wolf algorithm was found to be Grey Wolf Optimizer[4].*

Keywords: *Optimization, Swarm algorithms, Wolf herd algorithm, Wolfs, Benchmarks*

1. Wprowadzenie

Problemy optymalizacyjne są kwestią, którą prawie wszyscy z nas znają szkoły podstawowej. Rozwiązanie funkcji X,Y zazwyczaj znajdowało się w miejscu zerowym, na przecięciu się wykresu wartości z osią X. Pozwalało nam to zazwyczaj określić ile owoców ma kupić sprzedawca aby jak najlepiej wykorzystać wzmożony popyt, oraz nie zamrozić funduszy w przypadku zbyt dużego zakupu. Dokładnie takimi problemami zajmują się algorytmy optymalizacyjne, które szukają jak najmniejszego kosztu w dostępnym nam obszarze poszukiwania. Teren ten jest zazwyczaj stworzony za pomocą pewnej liczby zmiennych bądź wyborów, których wartości musimy odpowiednio dostosować aby uzyskać najlepszy wynik.

Jednak skoro została tutaj przywołana szkoła, to czemu nie skorzystać np. z pochodnych, bądź liniowo obliczyć wszystkie wartości? Cóż, algorytmy oparte na pochodnych istnieją (takie jak), ale okazało się, że są one zbyt czasochłonne/wadliwe aby dało się z nich korzystać w coraz to bardziej złożonych funkcjach. Dokładnie z tego samego powodu nie używa się liniowych obliczeń. Mimo iż moc komputerowa rośnie w zastraszającym tempie (zgodnie z prawem Moora głoszącym o podwajaniu się liczby tranzystorów co około dwa lata), jeszcze szybciej rosła złożoność problemów, które należało rozwiązać. Dodatkowo niektóre problemy wymagają

szybkiej reakcji, jak na przykład obliczenie długości kroku motoru w robocie.

Dziedziną zajmującą się szybkim rozwiązywaniem problemów optymalizacyjnych, jednak bez gwarancji znalezienia najlepszego rozwiązania, jest heurystyka. Nauka ta nie bierze pod uwagę jedynie poprawności rozwiązania, ale też prostotę samego algorytmu i prędkość działania. Aby zobrazować sens tej nauki, weźmy na przykład problem komiwojażera. Występuje w nim pewna liczba miast, oraz trasy je łączące o określonej długości. Celem jest takie pokonanie wszystkich dróg, aby uzyskać najmniejszy wynik. Z uwagi na ilość kombinacji dróg (których liczba wynosi silnię z $(n-1)/2$, co dla dziesięciu miast daje nam liczbę 181440) obliczenie tego jedynego najlepszego rozwiązania jest niezwykle czasochłonne. Jeśli jednak zamiast znalezienia najlepszego wyniku zadowolimy się tylko wartością zbliżoną do niego, to algorytm heurystyczny znajdzie rozwiązanie w przeciągu sekund.

Algorytmy optymalizacyjne zazwyczaj działają w sposób losowy, co z jednej strony pozwala na lepsze przeszukiwanie obszaru (niedeterministyczne kolejne kroki pozwalają na mniejszą szansę zakopania się w rozwiązaniach lokalnych). Jest to szczególnie ważne kiedy obszar funkcji ma bardzo nieprzewidywalną strukturę. Z drugiej strony jednak sprawia, że każde rozwiązanie zazwyczaj różni się od poprzedniego, co w przypadku dużych różnic między kolejnymi wynikami sprawia, że skuteczność algorytmu staje pod znakiem zapytania.

Funkcjonowanie algorytmów jest oparte na iteracjach, to jest takich samych krokach, które są podejmowane tak długo, aż nie osiągniemy pewnego celu. Sprawia to, że algorytmy te są zazwyczaj stosunkowo proste. Pozwala to na łatwą implementację i modyfikację działania. Jest to ważne, gdyż bardzo często algorytmy są pisane z myślą o wszystkich rodzajach funkcji. Zazwyczaj jednak zależnie od projektu w którym mają one być użyte, różne są kryteria odnośnie dokładności, bądź dostosowania do konkretnego typu problemów. Prosta wypływa też na prędkość działania.

Przeważająca większość algorytmów korzysta z populacji (wielu agentów posiadających własne rozwiązanie), chociaż możliwe jest oparcie działania na tylko jednym rozwiązaniu. Generalnie rzecz biorąc, najczęściej wyróżnia się trzy główne podgatunki algorytmów optymalizacyjnych:

- Genetyczne - inspirujące się procesami ewolucji zachodzącymi w naturze. Funkcjonuje w nich populacja, która w każdej iteracji jest sortowana za pomocą funkcji dopasowania, aby następnie lepsza część członków (zazwyczaj połowa) została wykorzystana aby stworzyć potomków. Gorsza część populacji zostaje usunięta. Przykładowe algorytmy to *Genetic Algorithm* [3] oraz *Differential evolution* [2].

- Fizyczne - oparte na zjawiskach fizycznych takich jak grawitacja bądź wyżarzanie. Ich działanie z uwagi całkowicie odmienne działanie różnych procesów może bardzo mocno się od siebie różnić. *Big Bang Big Crunch* [5] oparto na teorii początku i końca wszechświata, to jest Wielkiego Wybuchu, oraz Wielkiego Krachu. Inicjalne jest generowane wiele rozwiązań rozprzestrzenionych po obszarze poszukiwania, aby następnie je zgromadzić do pojedynczego wyniku. Innymi algorytmami fizycznymi są np. *Gravitational Search Algorithm* [6] bądź *Simulated Annealing* [7].

- Rojowe - symulowane jest w nich najczęściej zachowanie stad, ławic bądź grup zwierząt jednego gatunku, które poszukują pożywienia. Z uwagi na mnogość zwierząt i ich stosunkowo proste zachowania, są to najczęściej tworzone algorytmy. Przykładowe rozwiązania to kolonia mrówek [8] oraz rój cząstek [1].

Wilcze algorytmy należą do ostatniej wymienionej grupy algorytmów. Używanie tych zwierząt jako inspiracji zaczęło się stosunkowo niedawno, ponieważ wszystkie opisane tu prace są zostały stworzone po roku 2010. Najstarsze rozwiązanie używające tych ssaków znalezione przez autora tej pracy wystąpiło w 2007 roku [9]. Wilki polują w małych grupach, najczęściej rodzinach. Ich taktyka na obalanie dużych zwierząt jest stosunkowo prosta i opiera się na kilku krokach:

1. Wykrycie ofiary z pomocą ich wyczulonego zmysłu węchu. Wilki regularnie przeszukują swój obszar łowny, który jest zazwyczaj stały. Pozwala im to na znalezienie zarówno kopytnych, jak i też innych źródeł pożywienia.

2. Zmuszenie pojedynczego osobnika do ucieczki. Jako że wilki nie podejmą walki z całym stadem, to spróbują one odizolować ich ofiarę od reszty. W przypadku spotkania samotnej ofiary też podejmą tę próbę, aby ją zmęczyć.

3. Okrążenia i próba obalenia zwierzęcia przez stado wilków. Proces ten może trwać nawet kilka godzin.

Zachowanie to jest przedstawione na zdjęciu 1. Zasady te są na tyle proste, że z powodzeniem można przenieść je na algorytm. Z uwagi jednak na istnienie kilkunastu algorytmów wilczych, rodzi się pytanie który z nich tak naprawdę najlepiej się nadaje do zastosowań komercyjnych bądź naukowych.

Rysunek 1: Zachowanie wilków podczas polowania na dużego kopytnego. Na części A przedstawiono pogoń, na B do D widoczne jest okrążanie i nękanie. Zdjęcie E pokazuje ostateczną walkę.

Niniejsza praca postara się odpowiedzieć na to pytanie. W następnym rozdziale przedstawiono 6 wilczych rozwiązań, które następnie przetestowano na 6 wykresach.

2. Wilcze algorytmy

2..1 Grey Wolf Optimizer (GWO)

Grey Wolf Optimizer [4] ukazał się na łamach czasopisma *Advances in Engineering Software* w 2014 roku. W publikacji przedstawiono inspirację autorów, schemat działania algorytmu, testy na 29 benchmarkach i 3 problemach technicznych. Pojawiło się w niej również zastosowanie algorytmu do rozwiązania problemu w inżynierii optycznej.

Algorytm oparto na zachowaniu stadnym wilków, w którym dzielą się one na poszczególne grupy:

- Alfa - para dowodząca. Podejmuje najważniejsze decyzje, takie jak wybór czasu snu bądź polowania.
- Beta - część pomagająca alfom w podejmowaniu i egzekwowaniu decyzji.
- Omega - najniższa ranga w hierarchii, najczęściej zajmująca się doglądaniem młodych. Omegi często są kozłami ofiarnymi dla silniejszych wilków.
- Delta - pozostałe wilki, zazwyczaj są to starsi, opiekunowie najmłodszych, strażnicy bądź zwiadowcy.

Co ciekawe autorzy GWO nie zaprezentowali w swojej pracy dowodów pochodzących z literatury naukowej, które potwierdzałyby takie twarde podziały żywych wilków. Badania na stadach w dziczy [10] [11] wydają się tym twierdzeniom twardo zaprzeczać. Przedstawiają one wilki raczej jako rodzinę, w której alfa są partnerami mającymi młode.

Alfę(α), betę(β) oraz deltę(δ) przeniesiono do algorytmu jako trzy najlepsze wilki. Pozostałe są nazywane omegami(ω) To na ich podstawie oblicza się kroki wszystkich wilków w trakcie okrążania ofiary, za pomocą następnych równań:

$$\overrightarrow{D} = |\overrightarrow{C} \cdot \overrightarrow{X}_p(t) - \overrightarrow{X}(t)| \quad (1)$$

$$\overrightarrow{X}_p(t+1) = \overrightarrow{X}_p(t) - \overrightarrow{A} \cdot \overrightarrow{D} \quad (2)$$

$$\overrightarrow{A} = 2\overrightarrow{a} \cdot \overrightarrow{r}_1 - \overrightarrow{a} \quad (3)$$

$$\overrightarrow{C} = 2 \cdot \overrightarrow{r}_2 \quad (4)$$

gdzie

$\overrightarrow{X}_p(t)$	: Oznacza pozycję ofiary, a w przełożeniu na działanie algorytmu średnia z trzech najlepszych wilków.
$\overrightarrow{X}$	: Pozycja wilka w przestrzeni poszukiwań.
$\overrightarrow{A}, \overrightarrow{D}, \overrightarrow{C}$	: Wektory współczynników
t	: Konkretna iteracja pętli.
$\overrightarrow{a}$	: Współczynnik który zmniejsza się z 2 do 0 na w trakcie iteracji, co ma zwiększać znaczenie eksploatacji podczas działania algorytmu.
$\overrightarrow{r}_1, \overrightarrow{r}_2$	: Liczby generowane losowo z przestrzeni [0,1].

Wzór 2 pozwala nam na obliczenie ruchu wilka, korzystając z pozycji ofiary oraz wprowadzając pewną losowość z pomocą $\overrightarrow{A}, \overrightarrow{C}$. Zmniejszanie wartości $\overrightarrow{a}$ pozwala z czasem zmniejszyć przedział wartości "skoku" wilka. Z uwagi na to, aby zapewnić że losowość będzie występować nawet w końcowych krokach algorytmu, wprowadzono dodatkowo $\overrightarrow{C}$, który zawsze daje wartości z przedziału $[2r, 0]$. Jako że ofiara nie jest obecna w algorytmie, jej pozycję symulujemy przy pomocy alfy, omegi i delty. Dzieje się tak na mocy następujących równań:

$$\begin{aligned} \overrightarrow{D}_\alpha &= |\overrightarrow{C}_1 \cdot \overrightarrow{X}_\alpha - \overrightarrow{X}| \\ \overrightarrow{D}_\beta &= |\overrightarrow{C}_2 \cdot \overrightarrow{X}_\beta - \overrightarrow{X}| \\ \overrightarrow{D}_\delta &= |\overrightarrow{C}_3 \cdot \overrightarrow{X}_\delta - \overrightarrow{X}| \end{aligned} \quad (5)$$

$$\begin{aligned} \overrightarrow{X}_1 &= \overrightarrow{X}_\alpha - \overrightarrow{A}_1 \cdot \overrightarrow{D}_\alpha \\ \overrightarrow{X}_2 &= \overrightarrow{X}_\beta - \overrightarrow{A}_2 \cdot \overrightarrow{D}_\beta \\ \overrightarrow{X}_3 &= \overrightarrow{X}_\delta - \overrightarrow{A}_3 \cdot \overrightarrow{D}_\delta \end{aligned} \quad (6)$$

$$\overrightarrow{X}(t+1) = \frac{\overrightarrow{X}_1 + \overrightarrow{X}_2 + \overrightarrow{X}_3}{1} \quad (7)$$

Działanie tych wzorów można zobaczyć na rysunku 2. Średnia z pozycji trzech najlepszych wilków pozwala nam uzyskać teoretyczną pozycję ofiary. Jak widać więc działanie algorytmu nie jest zbyt złożone. W algorytmie 1 przedstawiono pseudokod GWO, który odzwierciedla kod wykorzystany w testach.

Algorithm 1 GWO

1: Stwórz członków $NumPopulation$ wilczego stada X_i, generując ich w losowych miejscach przestrzeni
2: $a = 2$
3: Oblicz i posortuj na podstawie wartości funkcji dopasowania dla każdego wilka
4: $X_\alpha, X_\beta, X_\delta$ są równe odpowiednio pierwszemu, drugiemu i trzeciemu wilkowi
5: **for** $mIter = 1, 2, \ldots, maxIteration$ **do**
6: **for** $i = 1, 2, \ldots, NumPopulation$ **do**
7: $A \leftarrow 2 * a * rand(0,1) - a$
8: $C \leftarrow 2 * rand(0,1)$
9: Oblicz nową pozycję wilka korzystając z równań 5, 6 i 7
10: **if** $X_i < X_\alpha$ **then**
11: $X_\alpha \leftarrow X_i$
12: **end if**
13: **if** $X_i > X_\alpha$ & $X_i < X_\beta$ **then**
14: $X_\beta \leftarrow X_i$
15: **end if**
16: **if** $X_i > X_\beta$ & $X_i < X_\delta$ **then**
17: $X_\delta \leftarrow X_i$
18: **end if**
19: **end for**
20: $a \leftarrow 2 - t * ((2)/mIter)$
21: **end for**
22: return X_α

2..2 Grey Wolf Optimizer with Evolutionary population (GWO-EPD)

Modyfikacja *Grey Wolf Optimizer* stworzona przez tych samych autorów została opublikowana rok po oryginalnym algorytmie w 2015 roku w czasopiśmie *Neural Computing and Applications*. Zmiana w nim zaprezentowana jest stosunkowo niewielka. Opiera się ona na dodaniu jeszcze jednego kroku, który eliminuje gorszą połowę populacji. Następnie tworzy ją znów, opierając się na pozycji jednego z trzech przewodzących wilków, bądź losowo w całej przestrzeni. Działanie przetestowano w publikacji na kilkunastu funkcjach testujących, ale co ciekawe porównano go tylko z oryginalnym algorytmem.

Działanie to opiera się na wzorach:

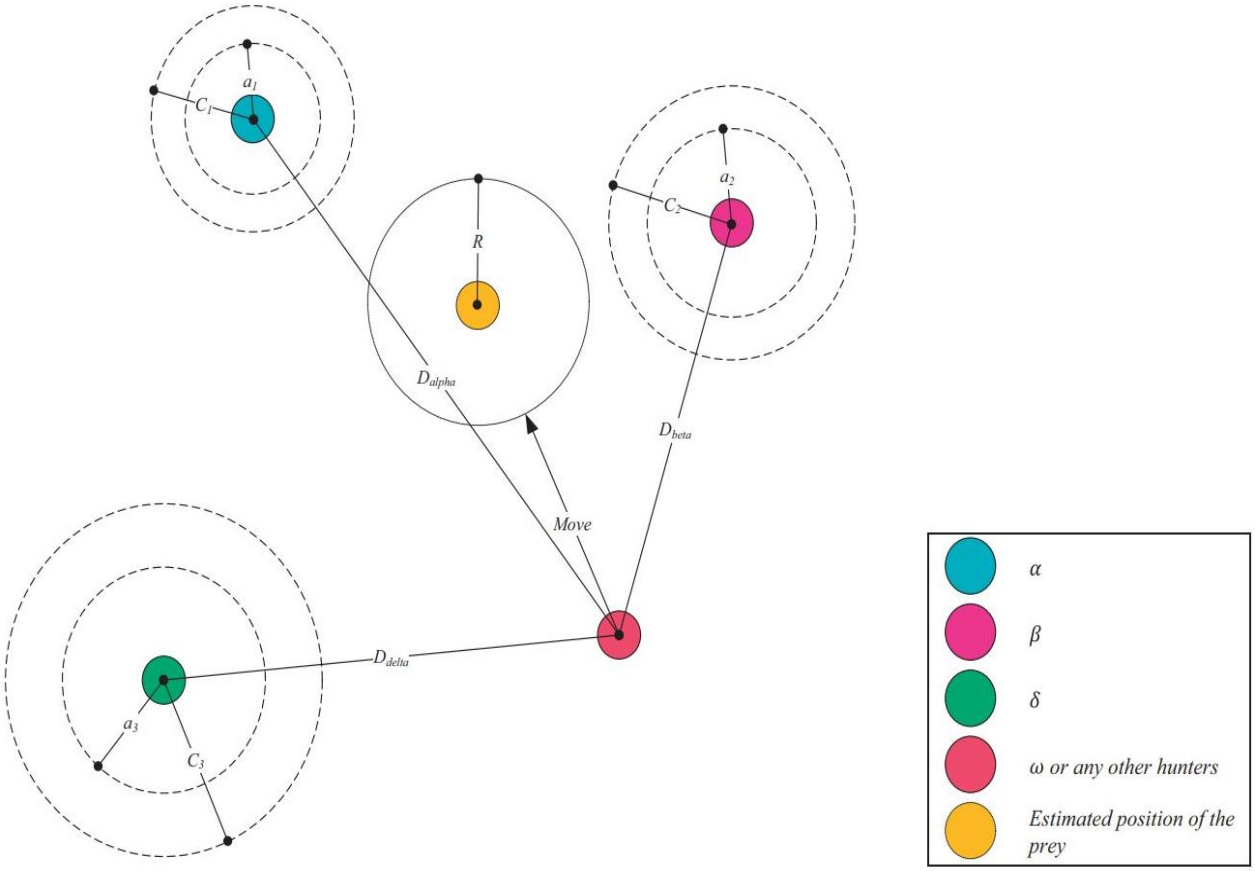
$$\overrightarrow{X}_p(t+1) = \overrightarrow{X}_\alpha(t) \pm (ub - lb \cdot r + lb) \quad (8)$$

$$\overrightarrow{X}_p(t+1) = \overrightarrow{X}_\beta(t) \pm (ub - lb \cdot r + lb) \quad (9)$$

$$\overrightarrow{X}_p(t+1) = \overrightarrow{X}_\delta(t) \pm (ub - lb \cdot r + lb) \quad (10)$$

$$\overrightarrow{X}_p(t+1) = (ub - lb \cdot r + lb) \quad (11)$$

Gdzie:



Rysunek 2: Działanie wzorów 5, 6, 7, rysunek z pracy *Grey Wolf Optimizer* [4]

$\overrightarrow{X}_p(t)$	: Oznacza pozycję w następnej iteracji.
$\overrightarrow{X}_\alpha, \overrightarrow{X}_\beta, \overrightarrow{X}_\delta$	: Pozycja wilka alfa, beta i delta.
ub, lb	: Respektywnie górna oraz dolna granica przestrzeni poszukiwania rozwiązania.
r	: Liczba generowane losowo z z przestrzeni [0,1].

Można zauważyć, że z uwagi na użycie górnej i dolnej wartości przedziału, oraz wykorzystanie losowej zmiennej z przedziału $[0, 1]$, umiejscowienie wilka może być dalekie od jednego z przewodzących wilków. Kod modyfikacji przedstawiono w algorytmie 2. Należy go umieścić na końcu każdej iteracji w GWO.

Algorithm 2 GWO-EDP

1: **for** $i = 1, 2, \ldots, NumPopulation/2$ **do**
2: $ub \leftarrow$ Górna granica przedziału
3: $lb \leftarrow$ Górna granica przedziału
4: $r \leftarrow rand(0, 1)$
5: Z równym prawdopodobieństwem, użyj jednego z wzorów 8, 9, 10 bądź 11
6: **end for**

2..3 Wolf Colony (WC)

Algorytm *The Wolf Colony Algorithm and Its Application* [12] zaprezentowano w czasopiśmie *HCTL Open International Journal of Technology Innovations and Research* w roku 2011. W publikacji pojawiły się testy na 4 funkcjach. Dodatkowo zaimplementowano algorytm na jednym problemie wyznaczania trasy.

Działanie algorytmu opiera się na zachowaniach zainspirowanych tymi z natury: zwiadach, oblężeniu i śmierci z powodu braku pożywienia. W algorytmie wykorzystano dość dużo parametrów, których objaśnienie wraz z wartościami użytymi w publikacji znajduje się poniżej:

$n = 200$	: Liczba stada.
$mark$	: Maksymalna liczba iteracji.
d	: Liczba wymiarów funkcji do obliczenia.
$\overrightarrow{X}_i$	: Wektor pozycji wilka i.
$q = 5$	: Liczba wilków przeprowadzająca zwiady.

$h = 4$: Ilość propozycji ruchów wygenerowana podczas zwiadów.
$maxdh = 15$: Maksymalna liczba wykonanych kroków podczas zwiadów.
$stepa = 1.5$: Wartość wykorzystywana do kontroli maksymalnej długości kroku podczas zwiadów.
$stepb = 0.9$: Wartość wykorzystywana do kontroli maksymalnej długości kroku podczas oblężenia.
$m = 5$: Ilość wilków usuwana podczas śmierci na koniec każdej iteracji.

Zwiady są przeprowadzane jedynie przez część stada, i są oparte na równaniu:

$$Y_j = \overrightarrow{X}_i + randn \cdot stepa \tag{12}$$

W którym Y_j równa się wartości funkcji dopasowania, a *randn* jest liczbą losową z przedziału [-1,1]. Naraz jest generowanych h takich rozwiązań, a najlepsze z nich jest porównywane do najlepszego wyniku w ogóle znalezionego przez algorytm. Jeśli jest lepsze, to zwiady dla tego konkretnego wilka kończą się. W przeciwnym wypadku jeśli ta pozycja jest lepsza od aktualnej pozycji, to następuje ruch wilka, a następnie zachowanie to jest powtarzane maksymalnie *maxdh* razy.

Podczas oblegania całe stado angażuje się w eksploatację potencjalnego rozwiązania globalnego, wyznaczanego przez pozycję najlepszego wilka. Jeśli w wyniku ruchu nowy wilk stanie się najlepszym, dalszy atak jest prowadzony w jego stronę. Dzieje się tak na mocy równania:

$$\overrightarrow{X}_i^{k+1} = \overrightarrow{X}_i^k + rand \cdot stepb \cdot (\overrightarrow{X}_{best} - \overrightarrow{X}_i^k) \tag{13}$$

Gdzie:

$\overrightarrow{X}_i^{k+1}$: Pozycja i wilka po wykonaniu ruchu.
$\overrightarrow{X}_i^k$: Pozycja i wilka przed wykonaniem ruchu.
$\overrightarrow{X}_{best}$: Pozycja najlepszego wilka.
$stepb$: Wartość wykorzystywana do kontroli maksymalnej długości kroku.
$rand$: Losowa liczba z przedziału [0,1].

Ostatnim krokiem jest śmierć głodnych wilków. Zdecydowano się tutaj ustawić tą wartość na $m = 5$, która jest stosunkowo niska biorąc pod uwagę wielkość stada użytego w publikacji. Na miejsce martwych wilków generowane są nowe losowo w przestrzeni. Pseudokod przedstawiony w algorytmie 3 obrazuje zachowanie kodu użytego w testach.

Algorithm 3 WC

```
Stwórz członków NumPopulation wilczego stada X_i, generując ich w losowych miejscach przestrzeni
s ← 0.12
q ← 5
h ← 4
maxdh ← 15
stepA ← 1.5
stepB ← 0.9
Oblicz i posortuj na podstawie wartości funkcji dopasowania dla każdego wilka
X_best ← pierwszy wilk
for mIter = 1, 2, ..., maxIteration do
    for i = 1, 2, ..., q do
        for j = 1, 2, ..., maxdh do
            Wygeneruj h X_scout nowych pozycji wilka na podstawie równania 12
            X_best−scout ← X_scout
            if X_best−scout < X_best then
                X_best ← X_best−scout
                X_i ← X_best−scout break
            else
                if X_best−scout < X_i then
                    X_i ← X_best−scout
                end if
            end if
        end for
    end for
    for i = 2, ..., NumPopulation do
        X_i ← wartość obliczona zgodnie z równaniem 13
    end for
    Wygenerowanie m wilków losowo w przestrzeni i zastąpienie nimi m najgorszych wilków
end for
```

2..4 Wolf Pack Algorithm (WPA)

Algorytm WPA [13] ukazał się na łamach czasopisma *Mathematical Problems* w 2014 roku. W publikacji opisano testy na 8 funkcjach porównawczych. Mimo iż nie jest to tam wprost opisane, algorytm ten jest mocno inspirowany, żeby nie powiedzieć oparty, na poprzednim algorytmie, *Wolf Colony*. W porównaniu do niego trochę zmieniono równanie kroku zwiadu, jak i oblegania i śmierci, oraz dodano dodatkowy krok, zwoływanie. Różnice dotknęły także zmiennych sterujących, które opisano w sposób następujący:

N : Liczba wilków.
$\overrightarrow{X}_1$: Pozycja wilka w pierwszej iteracji generowana losowo.
Y_{lead} : Najlepsza/najwyższa wartość wyznaczona przez funkcję optymalizowaną dla najlepszego wilka.
k_{max} : Maksymalna liczba iteracji.

$S = 0.12$: Parametr odpowiadający za długość kroku wilka.
$L_{near} = 0.08$: Minimalna odległość od wilka prowadzącego podczas zwoływania.
$T_{max} = 8$: Maksymalna liczba powtórzeń zachowania zwiadowczego.
$\beta = 2$: Liczba wpływająca na ilość przeżywających wilków.

Zwiady są wykonywane przez całe stado, poza najsilniejszym (najlepszym) członkiem. Równanie wykorzystane do symulacji tego stanowiska to:

$$\overrightarrow{X}_i^p = \overrightarrow{X}_i + sin(2\pi \cdot \frac{p}{h}) \cdot S, \quad p = [1, 2, ..., h] \quad (14)$$

Gdzie:

$\overrightarrow{X}_i^p$: Pozycja wilka po podjęciu kroku w stronę p.
h : wartości z przedziału $[h_{min}, h_{max}]$ będące liczbą całkowitą, niestety proponowany przedział nie został podany przez twórców.

Zasada na aktualizację bądź kontynuowanie tego procesu jest taka sama jak w WC, zmieniła się jedynie nazwa zmiennej sterującej maksymalną ilością kroków na T_{max}. Z uwagi na podjęcie zwiadów przez całe stado, oraz teoretycznej górnej wartości tych kroków, znacząco wzrósł też czas działania algorytmu.

Zwoływanie ma za zadanie przeprowadzić konwersję wszystkich wilków w pobliże ofiary. Wilk prowadzący gra rolę ofiary. Pozycja ta jest oznaczana jako $\overrightarrow{X}_{lead}$. Równanie przedstawiające to zachowanie to:

$$\overrightarrow{X}_{i+1} = \overrightarrow{X}_i + \frac{S}{2} \cdot \frac{(\overrightarrow{X}_{lead} - \overrightarrow{X}_i)}{|\overrightarrow{X}_{lead} - \overrightarrow{X}_i|} \quad (15)$$

W przypadku uzyskania lepszej wartości funkcji dopasowania przez aktualnego wilka to on staje się ofiarą. W przeciwnym wypadku ruch jest powtarzany do osiągnięcia wartości granicznej, tj. $L(\overrightarrow{X}_{lead}, \overrightarrow{X}_i) < L_{near}$. Z równaniem tym jest jednak jeden problem. W przypadku uzyskania na dole równania zera (kiedy wilk jak i ofiara stoją w tym samym miejscu), mamy do czynienia z wartością nieskończoną. Z uwagi na brak opisu tego problemu przez autorów algorytmu, podczas implementacji usunięto dolną część ułamka aby temu zapobiec. Nie jest jasne jak wpłynęło to na skuteczność algorytmu, ale było to wymagane do testów.

Obleganie jest prawie identyczne jak w przypadku poprzedniego algorytmu, i jest dane na mocy poniższego równania:

$$\overrightarrow{X}_{i+1} = \overrightarrow{X}_i + \lambda \cdot 2S \cdot |\overrightarrow{X}_{lead} - \overrightarrow{X}_i| \quad (16)$$

λ oznacza wartość losową z przedziału [-1,1]. Jeśli wartość funkcji dopasowania wilka X_{i+1} jest lepsza po kroku, jest ona aktualizowana.

Zmiany dotknęły także śmierci stada, w której to bierze udział większa liczba członków. R wilków ginie, a wartość ta jest generowana losowo z przedziału $[n/(2 \cdot \beta), n/\beta]$. Pozycje nowych wilków wyznacza równanie:

$$\overrightarrow{X} = \overrightarrow{X}_{lead} \cdot rand \quad (17)$$

Gdzie *rand* jest liczbą z przedziału $[-0.1, 0, 1]$. Pseudokod został podany w algorytmie 4

2..5 Wolf search algorithm

The Wolf search algorithm with ephemeral memory opublikowano w 2012 roku podczas konferencji *Seventh International Conference on Digital Information Management*. Jego działanie sprawdzono na ośmiu funkcjach w dwóch różnych wariantach. Niestety w pracy nie podano przyjętych parametrów sterujących algorytmem, co należy zaliczyć na duży minus. Mechanizm działania jest tutaj inny od pozostałych wilczych stad. Zamiast polować na ofiarę symulowaną przez lepszego członka, zwierzęta wypatrują lepszego miejsca/ofiary w sposób przypadkowy. Ruch ten jest podany następującym równaniem:

$$\overrightarrow{X}_i = \overrightarrow{X}_i + \alpha \cdot v \cdot rand() \quad (18)$$

Gdzie:

α : Parametr sterujący prędkością wilka.
v : Parametr sterujący zasięgiem wzroku
$rand()$: Liczba generowana losowo z przedziału [-1,1]

Krok zostanie wykonany tylko, jeśli wygenerowane miejsce jest lepsze od aktualnie zajmowanego. Jako że ssaki te mają dobrą pamięć, autorzy zasymulowali ten fakt poprzez dodanie sprawdzenia czy dane miejsce już zostało zwiedzone w poprzednich krokach. Ilość miejsc wstecz zapamiętanych przez wilka może zostać zmieniona. Jednakże gdy poruszamy się w przestrzeni ciągłej, możliwość wylosowania dwa razy tego samego miejsca, tj. powrotu do uprzednio zwiedzonego miejsca, jest co najmniej znikoma. Z tego powodu tego typu rozwiązania są zazwyczaj stosowane w problemach dyskretnych, jak problem komiwojażera. Z uwagi na to, że testy zaprezentowane w tej publikacji są oparte na przestrzeniach ciągłych, ten fakt pominięto. Generując liczbę losową *rand*() tysiąc razy w środowisku R, ani razu nie powtórzyła się ta sama wartość, co symuluje możliwość powrócenia w to samo miejsce. Oczywiście jeśli miejsc w okolicy wilka byłoby więcej (zależnie od długości pamięci wilka oraz ich

Algorithm 4 WPA

Stwórz członków $NumPopulation$ wilczego stada X_i, generując ich w losowych miejscach przestrzeni
$s \leftarrow 0.12$
$L \leftarrow 0.08$
$T \leftarrow 8$
$beta \leftarrow 2$
Oblicz i posortuj na podstawie wartości funkcji dopasowania dla każdego wilka
$X_{best} \leftarrow$ pierwszy wilk
for $mIter = 1, 2, \ldots, maxIteration$ **do**
  for $i = 1, 2, \ldots, NumPopulation$ **do**
    $h \leftarrow rand(4, 12)$
    for $i = 1, 2, \ldots, h$ **do**
      Wygeneruj X_{scout} nowych pozycji wilka na podstawie równania 14
      if $X_{scout} < X_{best}$ **then** $X_{best} \leftarrow X_{scout}$
        $X_i \leftarrow X_{scout}$ break
      else
        if $X_{scout} < X_i$ **then**
          $X_i \leftarrow X_{scout}$
        end if
      end if
    end for
  end for
  for $i = 1, 2, \ldots, NumPopulation$ **do**
    while $X_{best} - X_i > L$ **do**
      Zmień pozycję wilka zgodnie z równaniem 15
      if $X_i < X_{best}$ **then** $X_{best} \leftarrow X_i$ break
      end if
    end while
  end for
  for $i = 1, 2, \ldots, NumPopulation$ **do** $X_{siege} \leftarrow$ wartość oblicznona zgodnie z równaniem 16
    if $X_{siege} < X_{best}$ **then** $X_{best} \leftarrow X_{siege}$
      $X_i \leftarrow X_{siege}$ break
    else
      if $X_{siege} < X_i$ **then** $X_i \leftarrow X_{siege}$
      end if
    end if
  end for
  $ilMarWil \leftarrow$ liczba losowa z przedziału $(1, (numPopulation/(2 * beta), numPopulation/beta))$
  for $i = 1, 2, \ldots, ilMarWil$ **do**
    $X_i \leftarrow$ wartość oblicznona zgodnie z równaniem 17
    if $X_i < X_{best}$ **then**
      $X_{best} \leftarrow X_i$
    end if
  end for
end for

bliskości) szansa ta by wzrastała, ale dalej byłaby ona pomijalna.

Krokiem, który ma zapewnić konwergencję, jest wzajemne wypatrywanie się wilków. Mają one określony zasięg wzroku, z którego wybierają najlepszego widocznego członka stada. Jeśli jego pozycja jest lepsza, podejmą one próbę ruchu w jego kierunku. Dzieje się tak na mocy równania:

$$\overrightarrow{X}_i = \overrightarrow{X}_i + \beta e^{-r^2}(\overrightarrow{X}_j - \overrightarrow{X}_i) \tag{19}$$

Gdzie:

β : Oznacza różnicę pomiędzy wartością funkcji dopasowania dla wilka $\overrightarrow{X}_i i \overrightarrow{X}_j$
r^2 : odległość pomiędzy wilkiem a jego towarzyszem podniesiona do kwadratu.
e : Stała Eulera.

Im dalej jeden wilk jest od drugiego, tym mniejszy będzie ruch. Zachowanie to ma promować eksplorację. Ostatnim krokiem układanki jest dodanie szansy na wyrwanie się z optimum lokalnego poprzez nagły i daleki skok wilka. Z pewną szansą, przyjętą w testach przedstawionych w tej pracy na 25%, zwierzęta uciekną z zajmowanej aktualnie pozycji zgodnie z równaniem:

$$\overrightarrow{X}_i = \overrightarrow{X}_i + \alpha \cdot s \cdot escape() \tag{20}$$

Gdzie funkcja escape zwróci wartość większą od zasięgu wzroku wilka, a mniejszą od połowy obszaru przeszukiwanego. s oznacza długość kroku. Pseudokod jest widoczny na algorytmie 5. Zawarto w nim też użyte wartości parametrów sterujących.

2..6 Gaussian Guided Self-Adaptive Wolf Search Algorithm (GSAW-SA)

Modyfikacja poprzedniego algorytmu, *Gaussian Guided Self-Adaptive Wolf Search Algorithm Based on Information EntropyTheory* ukazała się na łamach czasopisma *Entropy* w 2018 roku. Dotyczy wprowadzenia zmiany parametrów sterujących ruchami wilków podczas działania algorytmu. Prędkość kroku, długość kroku, zasięg wzroku oraz szansa na ucieczkę zostały podporządkowane tej logice. Aby to uzyskać wykorzystano mapę chaosu *Gaussian* (pol. Guassa). Nowe zmienne są generowane na koniec każdej iteracji przy pomocy równania:

$$x_{n+1} = exp(-\alpha \cdot x_n^2) + \beta \quad = e^{-\alpha x_n^2} + \beta \tag{21}$$

α ustawiono na 5.4, a β na -0,52. X oznaczono zmienną której wartość jest zmieniana. Można zauważyć, że wartości te są bardzo podatne na początkowo przyjętą liczbę. Niestety autorzy nie podali w jaki sposób zainicjować zmienne. Aby wykonać testy, zdecydowano się na ustawienie wszystkich zmiennych na 0.25. Parametry sterujące nie są

Algorithm 5 WSA

Stwórz członków $NumPopulation$ wilczego stada X_i, generując ich w losowych miejscach przestrzeni
$\alpha \leftarrow 0.25$
$v \leftarrow 1$
$s \leftarrow 0.8$
$p_a \leftarrow 0.75$
Oblicz i posortuj na podstawie wartości funkcji dopasowania dla każdego wilka
$X_{best} \leftarrow$ pierwszy wilk
for $mIter = 1, 2, \ldots, maxIteration$ **do**
    for $i = 1, 2, \ldots, NumPopulation$ **do**
        $X_i \leftarrow$ wartość obliczona zgodnie z równaniem 18
        Wartość ta zostaje przypisana tylko jeśli $X_i < X_{i+1}$
        $BliskieWilki \leftarrow$ Wilki w zasięgu wzroku v
        $NajWilk \leftarrow$ Najlepszy z bliskich wilków
        if $NajWilk < X_i$ **then**
            Wykonaj równanie 19 dla wilka i
        else
            Powtórz krok pierwszy, tj równanie 18 i ewentualną podmianę pozycji
        end if
        if $LosowaLiczba[0, 1] > p_a$ **then**
            Wykonaj ucieczkę zgodnie z równaniem 20
        end if
    end for
end for

jednak zmieniane za każdą iteracją. Wprowadzono test, w którym najpierw wybierane są 4 różne losowe wilki. Następnie przy pomocy jednego z czterech mechanizmów genetycznych widocznych w tabeli 1, sprawdzamy najlepsze rozwiązanie z tym wygenerowanym. Jeśli jest lepsze, zmieniamy parametry. Cały mechanizm można zobaczyć w pseudokodzie widocznym na algorytmie 6. W testach użyto równania genetycznego oznaczonego jako RDE2.

3. Testy

3..1 Narzędzia

Do testów wykorzystano popularny język skryptowy R, w wersji 4.0.3. Jest on głównie używany do statystyki oraz pracy z dużymi ilościami danych, takimi jak *data mining*. Z uwagi na wykorzystaną przy nim licencję GNU, można z niego korzystać darmowo. Został on użyty we wszystkich zaprezentowanych tutaj publikacjach wilczych algorytmów.

Z języka R korzysta się z konsoli. Aby ułatwić pi-

Algorithm 6 GSAWSA

Stwórz członków $NumPopulation$ wilczego stada X_i, generując ich w losowych miejscach
$accelerate, p_a, s, v \leftarrow 0.25$
$P_{update} \leftarrow 0.75$
$\alpha \leftarrow -5.4$
$\beta \leftarrow -0.52$
Oblicz i posortuj wilki
$X_{best} \leftarrow$ pierwszy wilk
for $mIter = 1, 2, \ldots, maxIteration$ **do**
    for $i = 1, 2, \ldots, NumPopulation$ **do**
        $X_i \leftarrow$ wartość obliczona zgodnie z równaniem 18
        Wartość ta zostaje przypisana tylko jeśli $X_i < X_{i+1}$
        $BliskieWilki \leftarrow$ Wilki w zasięgu wzroku v
        $NajWilk \leftarrow$ Najlepszy z bliskich wilków
        if $NajWilk < X_i$ **then**
            Wykonaj równanie 19 dla wilka i
        else
            Powtórz krok pierwszy, tj równanie 18 i ewentualną podmianę pozycji
        end if
        if $LosowaLiczba[0, 1] > p_a$ **then**
            Wykonaj ucieczkę zgodnie z równaniem 20
        end if
    end for
    if $losowa(0, 1) > P_{update}$ **then**
        Wylosuj 4 pozycje różnych wilków
    end if
    for $mIter = 1, 2, \ldots, params$ **do**
        Oblicz wartość RDE2 1.
        if $X_{evolve} < X_{best}$ **then**
            for $j = 1, 2, \ldots, 4$ **do**
                if $losowa(0, 1) > P_{update}$ **then**
                    $param_j \leftarrow$ wartość z równania 21
                end if
            end for
        end if
    end for
end for

Tabela 1: Funkcje krzyżujące użyte w SAWSA

Nazwa	Wzór
DE1	$y = best_{gloable} + F \cdot (x_{r1} + x_{r2} - x_{r3} - x_{r4})$
DE2	$y = x_{r1} + F \cdot (best_{gloable} - x_{r2}) - F \cdot (x_{r3} - x_{r4})$
DE3	$y = best_{gloable} + F \cdot (x_{r1} - x_{r2})$
RDE1	$y = best_{selected} + F \cdot (x_{r1} + x_{r2} - x_{r3} - x_{r4})$
RDE2	$y = x_{r1} + F \cdot (best_{selected} - x_{r2}) - F \cdot (x_{r3} - x_{r4})$
RDE3	$y = best_{selected} + F \cdot (x_{r1} - x_{r2})$
RDE4	$y = x_{r1} + F \cdot (x_{r2} - x_{r3})$

sanie skryptów, użyto narzędzia R studio. Pozwala ono w prosty sposób uzyskać dostęp do konsoli, kodu, plików. Widoczne są w nim również załadowane zmienne środowiskowe, funkcje czy inne obiekty.

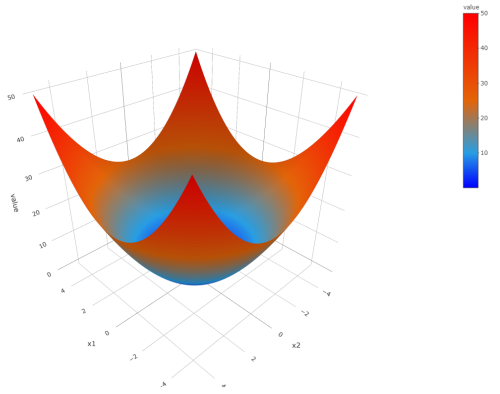
Jako że publikacja GWO wskazywała pakiet R, do którego został dodany kod źródłowy algorytmu, postanowiono z niego skorzystać. W paczce tej zawarto też inne klasyczne rozwiązania problemów optymalizacyjnych (np. PSO czy GA). Korzystanie z niej jest darmowe. Nazywa się ona *metaheuristicOpt*. Wyniki GWO oraz pozostałych klasycznych algorytmów zostały uzyskane przy pomocy tego pakietu.

Resztę wilczych algorytmów autor zaimplementował samodzielnie. W porównaniu do oryginalnych kodów, dokonano pewnych zmian w WSA, GSAWSA oraz WPA. Powody te zostały opisane w poprzednim rozdziale, i wskazują na potrzebę dokładnego opisania użytych parametrów sterujących, czy niepoprawnych wartości wynikających z niektórych równań. Pozostałe algorytmy zostały zaimplementowane zgodnie z myślą oryginalnych autorów.

Funkcje testowe należą do klasycznych problemów optymalizacyjnych. Aby uniknąć potrzeby pisania ich kodu, zostały one pobrane z strony *Optimization Test Functions and Datasets* [14], dostępnej pod adresem https://www.sfu.ca/ ssurjano/optimization.html. Z pośród kilkudziesięciu dostępnych tam funkcji ciągłych wybrano 6.

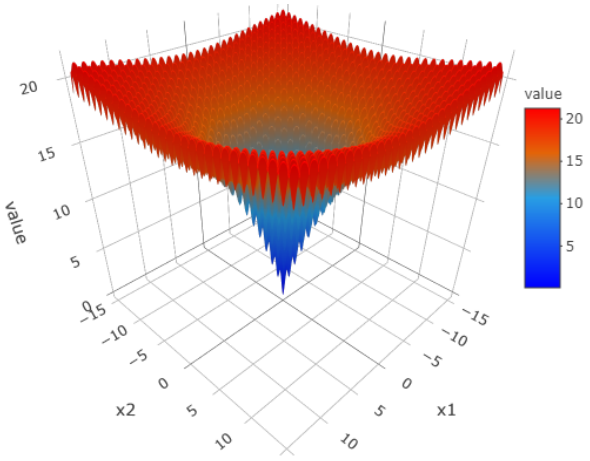
3..2 Funkcje testowe

Wzory, miejsca zerowe, wymiary oraz przestrzeń przeszukiwana zostały zawarte w tabeli 2. Wykresy funkcji zostały wygenerowane przy pomocy funkcji *plot_ly* z pakietu plotly. Pozwala on na tworzenie ładnych i wyraźnych wykresów. Pierwszą funkcją wykorzystaną do testów jest *Sphere*. Jest ona stosunkowo prosta, i bada głownie eksploatację algorytmów. Stanowi ona niejako test, którego niezaliczenie równa się z pewną porażką stworzonego rozwiązania. Wersja w trzech wymiarach jest widoczna na rysunku 3.



Rysunek 3: Wykres 3D funkcji Sphere

Drugą wykorzystaną funkcją jest Ackley. Zawiera ona wiele lokalnych optimum. Ma kształt gąbczastego leja. Parametry z których ona korzysta ustawiono na $a = 20, b - 0.2ic = 2\pi$. Jej wykres widać na rysunku 4.



Rysunek 4: Wykres Ackley

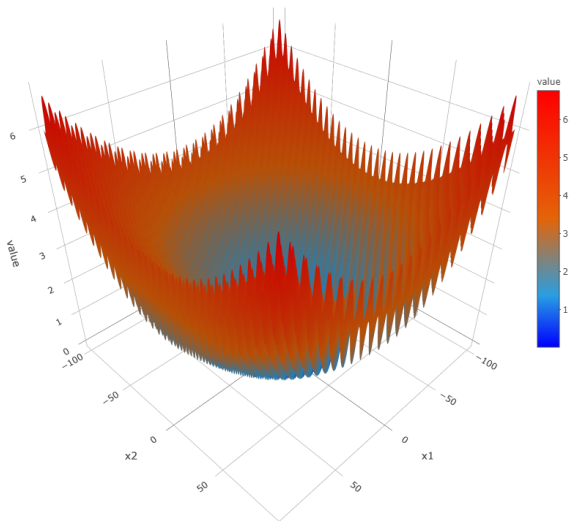
Kolejny, podobny wykres posiada Griewank. Różni się ona od Ackley wyglądem ogólnym, ale tak samo jest najeżona lokalnymi rozwiązaniami funkcji. Widać ją na rysunku 5.

Czwartym benchmarkiem jest Levy. Ma ona o wiele mniej miejsc zerowych, ale za to nie jest tak regularna. Jej wykres można zobaczyć na rysunku 6.

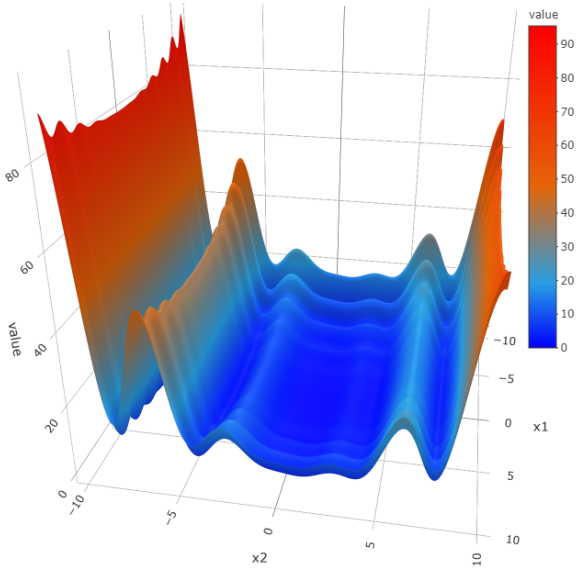
Rastrigin ponownie posiada wiele lokalnych rozwiązań, i podobny wygląd do Griewanka, ale ma mniejszą połać terenu. Wykres widać na rysunku 7.

Tabela 2: Funkcje wykorzystane do porównania skuteczności algorytmów

Funkcja	Formuła	f_{min}	Przestrzeń	Wymiar
Sphere	$F(\overrightarrow{x}) = \sum_{i=1}^{n} x_i^2$	0	(-5.12,5.12)	10
Ackley	$F(\overrightarrow{x}) = -aexp(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2} - exp(\frac{1}{d}\sum_{i=1}^{d} cos2\pi x_1) + a + exp(1))$	0	(-32.768, 32.768)	10
Griewank	$F(\overrightarrow{x}) = \frac{1}{4000}\sum_{i=1}^{n} x^2 - \Pi_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) + 1$	0	(-600,600)	10
Rastrigrin	$F(\overrightarrow{x}) = 10d + \sum_{i=1}^{d}[x_i^2 - 10cos(2\pi x_i)]$	0	(-5.12,5.12)	10
Levy	$F(\overrightarrow{x}) = sin^2(\pi\omega_1) + \sum_{i=1}^{d}[1 + 10sin(\pi\omega_i + 1)] + (\omega_d - 1)[1 + sin^2(2\pi\omega_d)]$, $gdzie\ \omega_i = 1 + \frac{x_i - 1}{4}$, $dla\ \ i = 1, \cdot, d$	0	(-10,10)	10
Schwefel	$F(\overrightarrow{x}) = 418.9829d - \sum_{i=1}^{d}[x_i sin(\sqrt{\lvert x_i \rvert})]$	0	(-500,500)	10



Rysunek 5: Wykres Griewank



Rysunek 6: Wykres Levy

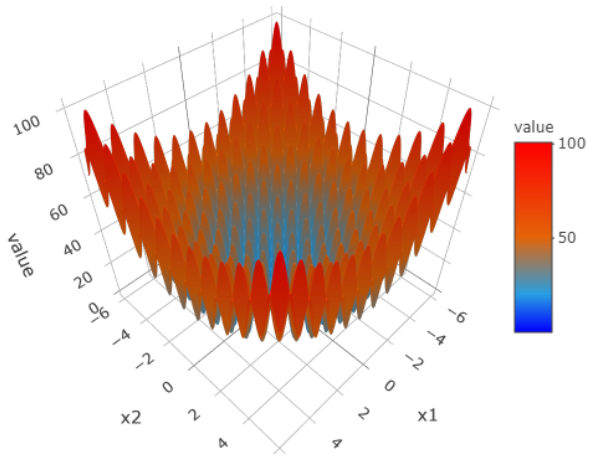
Ostatnim elementem układanki, oraz najtrudniejszym problem optymalizacyjnym jest Schwefel. Ma ona głębokie i rozległe optimum lokalne. Widać to wyraźnie na rysunku 8. Algorytmy genetyczne poradziły sobie o wiele lepiej, co jest po części związane z ich działaniem. Eksploracja jest w nich zazwyczaj o wiele lepsza, jednak często nie radzą sobie one tak dobrze z eksploatacją.

3..3 Wyniki

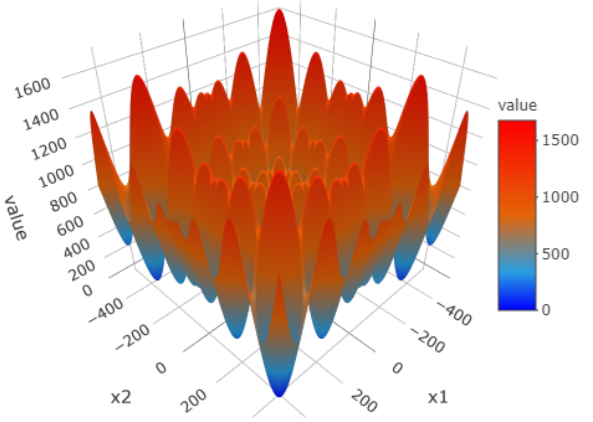
W celu porównania wyników wilczych algorytmów z innymi rozwiązaniami, postanowiono skorzystać z trzech innych algorytmów. Użyto *Genetic Algorithm* (GA) [3], *Particle swarm optimization* (PSO) [1] oraz *Differential evolution* (DE) [2]. Parametry w GA zostały zainicjalizowane w następujący sposób:

0.8 : Współczynnik krzyżowania
0.1 : Szansa na mutację

W *Particle swarm optimization* występują 4 parametry. Ich wartość ustawiono na:



Rysunek 7: Wykres Rastrigin



Rysunek 8: Wykres Schwefel

$Vmax = 2$: Prędkość maksymalna
$ci = 1.49445$: Współczynnik dążenia do najlepszego lokalnego rozwiązania
$cg = 1.49445$: Współczynnik dążenia do najlepszego globalnego rozwiązania

$w = 0.729$: Bezwładność, określająca wpływ rozwiązania znalezionego w poprzednim kroku na następny krok

Strategia i współczynniki DE wybrano następująco:

Best/1 : Strategia generowania nowych członków
0.5 : Współczynnik mutacji
0.8 : Współczynnik amplifikacji (ang. *scaling vector*)

Wyniki wszystkich testów są widoczne w tabeli 3 oraz 4. Każdy z algorytmów został przetestowany 30 razy na każdej z funkcji. Następnie wyliczono średni wynik oraz odchylenie standardowe wyników. Dodatkowo obliczono średni czas w sekundach oraz odchylenie standardowe od tych wartości. Jedynie WPA uzyskał wyniki czasu w sekundach. Każdy z algorytmów działał na ustawionym tysiącu iteracji.

Aby dalej poszerzyć wiedzę o jakości wyników, przygotowano wykresy konwergencji na funkcji Griewank. Wynik GWO, GWO-EPD, WC, WPA oraz GA widnieje na rysunku 9, a pozostałych na rysunku 10. Na pierwszym wykresie widnieje jedynie 100 iteracji, z uwagi na szybką zbieżność. Rysunki wykonano na podstawie jednego testu.

Algorytm *Grey Wolf Optimizer* jest najlepszy z obecnych tu rozwiązań wilczych. Nie poradził sobie jedynie z ostatnią funkcją. Mimo różnicy w czasie na korzyść niektórych z innych wilków, wykres zbieżności pokazuje że jedynie WPA jest od niego szybszy i to jedynie pod względem iteracji. Poza tym, oferuje lepsze rozwiązania w funkcji Levy i Rastrigin. Działa lepiej od *Particle swarm optimization*, który jest uznanym algorytmem. Jego jedynymi rywalami są oba algorytmy genetyczne, od których odróżnia go precyzja. Lepiej eksploatuje on rozwiązania globalne. W funkcji Schwefel poradziły sobie one lepiej od niego, choć jedynie GA w sposób znaczny (o dwie wielkości). Ogólnie jednak rzecz biorąc, widać dlaczego jest to najbardziej cytowany algorytm wilczy z wybranych w tej pracy.

Grey Wolf Optimizer with Evolutionary population jest modyfikacją niewiele różniącą się od oryginału. Choć autorzy mogą się w teorii powołać na *No free lunch theorems for optimization* [15], który traktuje o tym, że każdy algorytm aby zyskać coś w jakiejś klasie problemów musi coś poświęcić w innej, to nie widać aby w tym przypadku się to udało. Wyniki obu algorytmów są prawie nierozróżnialne, z czego największa różnica jest w ostatniej funkcji. Niestety na niekorzyść modyfikacji względem oryginalnego algorytmu. Należałoby potraktować to raczej jako porażkę. W końcu celem zmian ma zazwyczaj być poprawa jakości.

Wolf colony sprawił się jako trzeci z wilczych rozwiązań. Zaoferował on bardzo podobne wyniki jak PSO, jednak często od niego gorsze. Mimo tego znalazł się na tym miejscu z uwagi na jego w miarę normalny czas, mimo faktu iż był wolniejszy od wszystkich oprócz WPA. Dodatkowo zaoferował podobną zbieżność jak GWO, nieznacznie mu ustępując. Był też do tego o wiele szybszy w konwergencji niż *Particle swarm optimization*. Poradził sobie też lepiej w ostatniej funkcji od pozostałych wilków i roju cząstek. Ogólnie więc mówiąc, jest to dobry algorytm, jednak nie można mówić tu o znacznej

różnicy względem klasycznego roju.

Biorąc pod uwagę dobre wyniki swojego oryginału, nie można się dziwić, że ktoś spróbował polepszyć algorytm wilczej kolonii. Jednak droga podjęta przez twórców *Wolf pack* jest definitywnie problematyczna. Na pierwszy rzut oka widać ogromny wzrost czasu względem innych algorytmów. Mimo iż teoretycznie zbieżność jest osiągana już w dosłownie kilku iteracjach, jest to okupione potwornym czasem wykonania w ustalonej ilości pętli. Dodatkowo, algorytm ten najbardziej poległ w ostatniej funkcji. Ciężko więc mówić o dużym sukcesie. Dodatkowo, różnica między zerem a 0.1 nie jest tak duża aby pozwolić sobie na taką różnice w czasie. W Levy algorytm też nie uzyskał najlepszego wyniku. Mimo tego, został on uznany za lepszego od pozostałych dwóch algorytmów wilczych.

Wolf search algorithm boryka się z dużą ilością problemów. Brak zawartych domyślnych wartości parametrów sterujących mógł wpłynąć na wynik, i nie sposób stwierdzić jak duże miało to znaczenie. Dodatkowo, z uwagi na mechanizm raczej należący do domen algorytmów optymalizujących funkcje dyskretne, można wyżej przeczytać o braku jego implementacji. Sam styl napisania publikacji też pozostawia wiele do życzenia w kwestii wytłumaczenia różnych operacji tego rozwiązania. Finalnie, wyniki tu uzyskane są najgorsze z obecnych, i zawsze słabsze niż klasyczny algorytm, PSO. Nie ma aż tak ogromnej różnicy żeby mówić o kompletnej porażce, ale nie ma tez powodu żeby z tego rozwiązania korzystać. Zbieżność widoczna na wykresie konwergencji jest bardzo skokowa, co dalej podkopuje pozycję WSA.

Modyfikacja WSA, *Gaussian Guided Self-Adaptive Wolf Search Algorithm*, nie różni się wiele od swojego oryginału. Popełniła ona podobne błędy jeśli chodzi o zawarcie wartości parametrów sterujących i tak samo ominęła dokładne wytłumaczenie pewnych kluczowych operacji. Zastosowanie ciekawego podejścia, jakim jest zmiana parametrów, nie przyniosło większych sukcesów. Wyniki względem oryginału są raz lepsze a raz gorsze. Dodatkowo, zapewne przez wprowadzenie dodatkowych obliczeń, algorytm jest wolniejszy. Ocena więc należy się podobna, jeśli nie wręcz gorsza od oryginału - w obecnym stanie brak powodów aby z tego rozwiązania korzystać.

4. Wnioski

Jak widać, wilcze algorytmy są w stanie zaoferować lepsze bądź podobne wyniki w porównaniu z innymi klasycznymi rozwiązaniami. Niewątpliwe najlepszy okazał się *Grey Wolf Optimizer*. Jedynie algorytmy ewolucyjne w jednym przypadku podważyły jego umiejętności. Jednak mimo tego, podobnie jak PSO, dalej nadaję się on do wykorzystania w realnych problemach optymalizacyjnych. Biorąc pod uwagę jego wysokie wskaźniki cytowań, inni badacze bez wątpienia zgadzają się z tym faktem. Modyfikacja GWO, dodająca mechanizm ewolucyjny nie wniosła za wiele do tematu, nieznacznie pogarszając wyniki.

Kolejne dwa algorytmy, *Wolf Colony* oraz *Wolf Pack* uzyskały gorszą ocenę, głównie ze względu na ich czas wykonania, i gorsze osiągi w części funkcji testowych. Mimo tego są porównywalne bądź lepsze od ich konkurenta rojowego, PSO.

Największą porażką na pewno odznaczyły się Wilk szukający oraz jego modyfikacja Gaussa. Wyraźne nagłe skoki w znajdywaniu lepszych rozwiązań są negatywnym zjawiskiem w konwergencji. Nie pozwala to dobrze określić optymalnego czasu w którym algorytm znajdzie rozwiązanie. Ogólnie rzecz biorąc wyniki tych rozwiązań nie są takie złe, ale po prostu istnieją lepsze alternatywy. Na przykład te wskazane w niniejszej pracy.

Podsumowując, algorytmy wilczego stada nadają się do rozwiązywania problemów optymalizacyjnych. Widać wyraźnie czemu te drapieżniki są często podejmowane jako inspiracja do tworzenia nowych rozwiązań bądź modyfikowania już istniejących. Algorytmy te pokazują nowe i ciekawe podejścia do od dawna znanego problemu.

Bibliografia

[1] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[2] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. 11(4), 1997.

[3] John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[4] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46 – 61, 2014.

[5] Osman K. Erol and Ibrahim Eksin. A new optimization method: Big bang–big crunch. *Advances in Engineering Software*, 37(2):106 – 111, 2006.

[6] Esmat Rashedi, Hossein Nezamabadi-pour, and Saeid Saryazdi. Gsa: A gravitational search algorithm. *Information Sciences*, 179(13):2232 – 2248, 2009. Special Section on High Order Fuzzy Sets.

Tabela 3: Wyniki algorytmów optymalizacyjnych.

Funkcja	Algorytm	Średnia	Odchylenie	Średni Czas[s]	Odchylenie[s]
Sphere $f_{min}(\overrightarrow{x}) = 0$	GWO	2.0553e-177	0	10.5816	2.5157
	GWO-EPD	7.7542e-176	0	11.8804	2.4582
	WC	0.2376	0.1363	15.7010	0.1884
	WPA	0	0	1.1865min	0.1166min
	WSA	3.7366	1.2116	4.7903	0.1740
	GSAWSA	8.8820	2.4832	6.5562	0.7882
	PSO	5.5897e-56	1.1653e-55	6.9985	1.2662
	DE	1.7554e-42	2.4270e-42	0.4618	0.0437
	GA	0.0015	0.0011	1.1592	0.1334
Ackley $f_{min}(\overrightarrow{x}) = 0$	GWO	4.9441e-15	1.5979e-15	15.6035	1.3637
	GWO-EPD	4.8257e-15	1.5283e-15	15.0345	1.4891
	WC	1.5461	0.5477	20.6302	0.3285
	WPA	4.4408e-16	0	1.7590min	0.2256min
	WSA	17.0793	1.1482	5.6244	0.3050
	GSAWSA	16.3092	1.1823	7.2237	0.2136
	PSO	3.9968e-15	0	7.3	0.4276
	DE	3.9968e-15	0	0.7036	0.0693
	GA	0.4647	0.1954	1.4450	0.1222
Griewank $f_{min}(\overrightarrow{x}) = 0$	GWO	0.0141	0.0160	13.8200	1.8674
	GWO-EPD	0.0225	0.0323	12.3534	1.8143
	WC	2.3631	1.3632	19.1972	1.6045
	WPA	0	0	1.4091min	0.1103min
	WSA	53.9747	16.0915	6.0211	0.2803
	GSAWSA	41.0102	9.9708	7.9068	0.3944
	PSO	0.5386	0.3019	6.7454	0.109
	DE	0.0441	0.0192	1.0485	0.2405
	GA	0.5803	0.2304	1.2187	0.07
Rastrigin $f_{min}(\overrightarrow{x}) = 0$	GWO	0.3323	1.2917	11.3405	1.0982
	GWO-EPD	0.4318	1.3261	11.4035	1.1253
	WC	20.3965	7.1699	17.8619	0.2876
	WPA	0	0	1.2134min	0.0215min
	WSA	60.0253	4.8182	5.6589	0.3012
	GSAWSA	65.6314	5.9479	8.2285	1.5928
	PSO	5.5054	2.5831	6.2371	0.2680
	DE	2.1557	1.3344	0.8224	0.1119
	GA	0.3053	0.2346	1.0348	0.1174

Tabela 4: Wyniki algorytmów optymalizacyjnych cz. 2.

Funkcja	Algorytm	Średnia	Odchylenie	Średni Czas[s]	Odchylenie[s]
Levy $f_{min}(\overrightarrow{x}) = 0$	GWO	0.0376	0.05	12.8763	0.9715
	GWO-EPD	0.0478	0.0512	11.7281	1.1759
	WC	0.1735	0.1340	23.8095	0.3820
	WPA	0.5393	0.2329	1.7894min	0.04061min
	WSA	8.2483	2.0257	6.001	0.2625
	GSAWSA	8.5194	2.2857	7.4072	0.2403
	PSO	1.4996e-32	0	8.4826	0.3885
	DE	1.4996e-32	0	1.0607	0.1201
	GA	0.0032	0.0021	1.1665	0.0920
Schwefel $f_{min}(\overrightarrow{x}) = 0$	GWO	969.3719	284.6110	12.2941	0.3728
	GWO-EPD	1096.2342	252.9833	11.3166	1.9551
	WC	1426.7238	320.7987	16.6173	0.3092
	WPA	2358.2336	246.4415	1.2656min	0.0299min
	WSA	1905.5797	147.8068	5.5269	0.3443
	GSAWSA	1731.03105	145.8400	6.7911	0.1325
	PSO	2053.4427	426.3038	6.0361	0.1060
	DE	687.7272	355.6033	0.7139	0.0712
	GA	1.9594	1.0994	1.0676	0.1832

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[8] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[9] C. Yang, X. Tu, and J. Chen. Algorithm of marriage in honey bees optimization based on the wolf pack search. In *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, pages 462–467, 2007.

[10] Rolf Peterson, Amy Jacobs, Thomas Drummer, L. Mech, and Douglas Smith. Leadership behavior in relation to dominance and reproductive status in gray wolves, canis lupus. *Canadian Journal of Zoology-revue Canadienne De Zoologie - CAN J ZOOL*, 80:1405–1412, 08 2002.

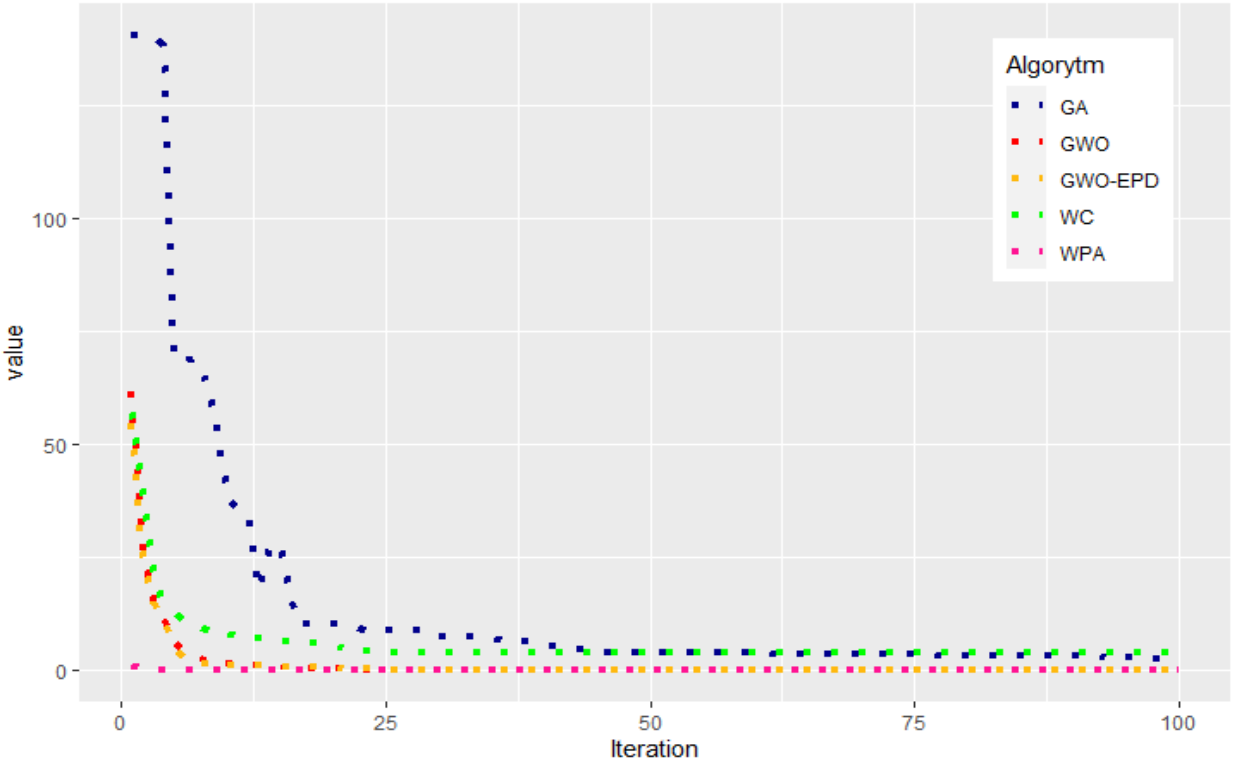
[11] L. Mech. Alpha status, dominance, and division of labor in wolf packs. *Canadian Journal of Zoology*, 77:1196–1203, 1999.

[12] C.-Y Liu, X.-H Yan, and H. Wu. The wolf colony algorithm and its application. *Chinese Journal of Electronics*, 20:212–216, 04 2011.

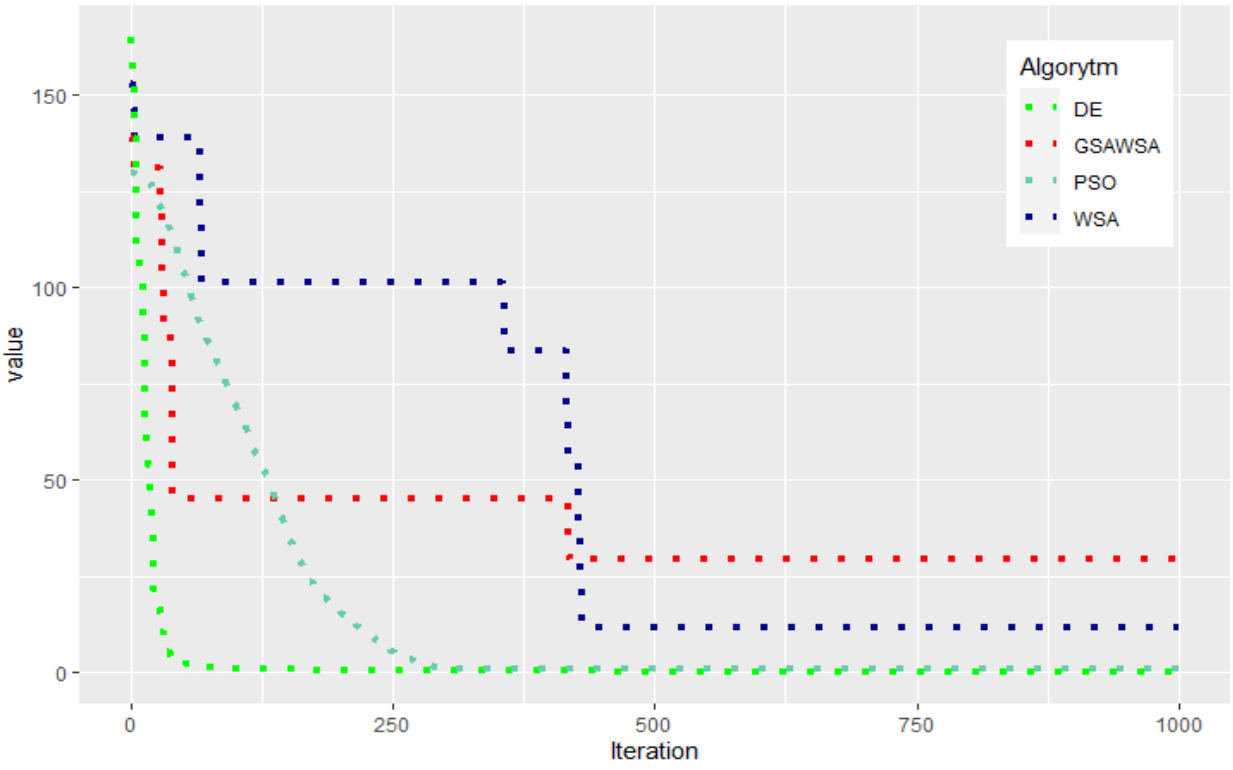
[13] Hu-Sheng Wu and Feng-Ming Zhang. Wolf pack algorithm for unconstrained global optimization. *Mathematical Problems in Engineering*, 2014:1–17, 03 2014.

[14] Derek Bingham Sonja Surjanovic. *Optimization Test Functions*, 2013.

[15] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.



Rysunek 9: Wyniki GWO, GWO-EPD, WC, WPA, GA w funkcji Griewank.



Rysunek 10: Wyniki WSA, GSAWSA, PSO oraz DE w funkcji Griewank.