ALGORYTMY METAHEURYSTYCZNE

Dokumentacja końcowa projektu

15-06-2025

Aleksander Bujnowski – nr albumu: 313595
Wojciech Kondracki – nr albumu: 310941

Spis treści

1 Opis problemu

Problemem badawczym jest analiza wpływu różnych generatorów liczb losowych, w tym quasi-losowych, na efektywność działania wybranego algorytmu ewolucyjnego. Algorytmy ewolucyjne opierają swoje działanie na losowości, która występuje w wielu etapach ich działania, takich jak:

- inicjalizacja populacji (losowy rozkład punktów początkowych),
- selekcja osobników (losowany jest podzbiór kandydatów),
- krzyżowanie (losowy wybór punktu krzyżowania),
- mutacja (losowa modyfikacja genotypu).

W każdej z tych faz jakość i charakterystyka generatora liczb losowych może wpłynąć na przebieg procesu optymalizacji. Przykładowo, można spodziewać się, że niektóre generatory będą prowadzić do szybszego zbiegania do lokalnego minimum, podczas gdy inne będą zapewniać bardziej równomierne pokrycie przestrzeni i zwiększać szanse na znalezienie optimum globalnego. W dalszej części pracy zostanie przeanalizowane, czy i jakie różnice w działaniu algorytmu ewolucyjnego wynikają z zastosowania różnych generatorów.

2 Opis rozwiązania

2.1 Algorytm ewolucyjny

Do badań zastosowany został **klasyczny algorytm ewolucyjny** o następujących właściwościach:

- selekcja *turniejowa*,
- krzyżowanie *jednopunktowe*,
- mutacja *uniform add perturb* (mutacja o wartość z zakresu $[-\delta, \delta]$, gdzie δ to zadany parametr, zależny od zakresu mutacji (lokalna lub daleka)),
- sukcesja *generacyjna*.

2.2 Funkcje testowe

Analiza zostanie przeprowadzona przy użyciu sześciu funkcji testowych: *f2, f8, f11, f17, f23, f29*, z zestawu *CEC 2017* [1], które umożliwiają ocenę skuteczności oraz stabilności działania algorytmu w różnych warunkach oraz dla problemów o zróżnicowanej złożoności.

2.3 Generatory liczb losowych

Wykorzystane w projekcie generatory liczb pseudolosowych (*PRNG*) oraz quasi-losowych (*QRNG*) są zestawione w poniższej tabeli:

Tabela 1: Generatory liczb losowych wykorzystanych w projekcie

GENERATOR	TYP	BIBLIOTEKA
Mersenne Twister [2]	PRNG	random [3]
PCG64 [4]	PRNG	numpy [5]
Xoshiro256 [6]	PRNG	randomgen [7]
Sobol [8]	QRNG	scipy [9]
Halton [10]	QRNG	scipy

Do generowania seedów użyty został kryptograficznie bezpieczny generator liczb losowych (*CSRNG*) z biblioteki secrets [11]. Dla każdego uruchomienia algorytmu losowano 32-bitowy seed przy użyciu funkcji *secrets.randbits(32)*, co zapewnia wysoką entropię i unikalność wartości.

2.4 Parametry eksperymentu

Każda seria badań (każdy przypadek funkcja testowa × generator) została wykonana z następującymi wartościami parametrów:

Tabela 2: Parametry eksperymentu zdefiniowane w projekcie

PARAMETR	OPIS	WARTOŚĆ
MAX_X	Granica przestrzeni poszukiwań	[-100, 100]
DIMENSIONALITY	Liczba wymiarów w problemie optymalizacji	30
RUNS	Liczba niezależnych uruchomień algorytmu w serii	30
U	Liczność populacji	50
PC	Prawdopodobieństwo krzyżowania	0.5
DELTA_S	Zakres mutacji lokalnej $[-\delta_s, \delta_s]$	0.1
DELTA_B	Zakres dużej (globalnej) mutacji $[-\delta_b, \delta_b]$	10
P_BIG_JUMP	Prawdopodobieństwo wykonania dużej mutacji	0.03
short_budget	Krótkoterminowy eksperyment	950
long_budget	Długoterminowy eksperyment	49950

Oznacza to, że każdy przypadek badawczy (funkcja × generator) był uruchamiany dwukrotnie — raz z ograniczonym budżetem (short_budget), a raz z rozszerzonym (long_budget). Dla stałej populacji liczącej 50 osobników przekładało się na 20 iteracji algorytmu z ograniczonym budżetem i 1000 iteracji algorytmu z rozszerzonym budżetem.

2.5 Zebrane wyniki i dalsza analiza

W ramach każdej serii badań rejestrowane były dwa rodzaje danych: przebieg konwergencji (wynik optymalizacji w każdej iteracji) oraz wyniki końcowe (wynik optymalizacji oraz czas trwania każdego uruchomienia).

Na podstawie zebranych danych wygenerowano zestawy plików wynikowych w formacie CSV, zawierające statystyki opisowe (takie jak średnia, odchylenie standardowe, wartości skrajne) dla każdej funkcji testowej, generatora liczb losowych oraz wariantu budżetowego. W celu oceny istotności różnic pomiędzy generatorami zastosowano nieparametryczny test Kruskala-Wallisa. Dla przypadków, w których uzyskano istotność statystyczną ($p < 0.05$), wykonano dodatkową analizę post hoc z użyciem testu Dunna z korektą Bonferroniego.

Wyniki numeryczne zilustrowano z wykorzystaniem:

- wykresów pudełkowych przedstawiających rozkład wyników końcowych,
- krzywych konwergencji prezentujących średni postęp optymalizacji w czasie,
- wykresów zależności czasu działania od jakości rozwiązania,
- (opcjonalnie) map cieplnych wyników testu Dunna.

3 Opis implementacji

W projekcie zaimplementowano algorytm ewolucyjny od podstaw w języku Python, bez użycia gotowych frameworków optymalizacyjnych. Taka własna implementacja zapewniła pełną kontrolę nad logiką działania oraz źródłami losowości, umożliwiając dokładną analizę wpływu różnych generatorów liczb losowych.

Wewnątrz algorytmu wykorzystywane są cztery podstawowe operacje losowe, zaimplementowane jako metody warstwy pośredniczącej, nazwanej *RNG*:

- *uniform(a, b, size)* – losowanie wartości z rozkładu jednostajnego $U(a, b)$.
  Zastosowanie: inicjalizacja populacji początkowej oraz mutacje.
- *choice(array)* – losowanie elementu z podanej tablicy.
  Zastosowanie: selekcja turniejowa, wybór rodziców do krzyżowania.
- *rand()* – losowanie wartości z przedziału $[0, 1)$.
  Zastosowanie: decyzje probabilistyczne - czy wykonać krzyżowanie lub mutację.
- *randrange(a, b)* – losowanie liczby całkowitej z zadanego zakresu.
  Zastosowanie: wybór punktu przecięcia w krzyżowaniu jednopunktowym.

Logika działania metod zależy od tego, który generator został wybrany, ale interfejs tychże funkcji pozostaje bez zmian. Dzięki zastosowaniu takiego wzorca projektowego możliwa jest łatwa podmiana źródła losowości bez modyfikacji pozostałej części kodu.

W celu zapewnienia poprawności działania i spójności interfejsu, każda z metod wrappera RNG została objęta testami jednostkowymi, używając biblioteki *pytest*. Testy te są automatycznie uruchamiane przy każdym pull requeście do głównej gałęzi projektu.

4 Wyniki

W ramach benchmarku *CEC 2017* rozwiązywane są problemy minimalizacyjne, co oznacza, że niższa wartość funkcji celu interpretowana jest jako wynik lepszy.

Dla zachowania przejrzystości analizy, wyniki prezentowane są w dwóch częściach: najpierw dotyczącej eksperymentów krótkoterminowych, a następnie długoterminowych. Każda z tych sekcji obejmuje analizę jakości rozwiązań, czasów działania oraz ocenę istotności statystycznej różnic pomiędzy zastosowanymi generatorami liczb losowych.

W następnych etapach raportu generator *Mersenne Twister* określany jest jako *Random*, a generator *PCG64* jako *Numpy*. Konwencja zaczerpnięta jest z nazw bibliotek, w których te generatory są wykorzystywane domyślnie.

4.1 Krótkoterminowy eksperyment (short_budget)

4.1.1 Wyniki optymalizacji

Tabela 3: Zestawienie statystyk z wyników optymalizacji funkcji w krótkim budżecie

Generator / Funkcja	f2				f8				f11			
	mean	*std*	*min*	*max*	*mean*	*std*	*min*	*max*	*mean*	*std*	*min*	*max*
Random	1.4e+49	7.8e+49	5.4e+37	4.2e+50	1209.15	41.98	1140.48	1279.22	21497.68	9656.54	7284.12	44327.68
Numpy	2.2e+45	7.4e+45	1.0e+37	3.9e+46	1194.88	38.97	1102.03	1282.36	22189.28	8668.93	9895.36	44101.05
Xoshiro	9.4e+45	3.4e+46	1.6e+37	1.8e+47	1212.31	47.32	1108.07	1315.47	23938.24	8975.15	6542.09	42807.39
Sobol	1.6e+46	5.1e+46	6.3e+37	2.4e+47	1198.97	36.73	1134.18	1288.56	22697.74	10000.02	5667.61	51354.79
Halton	3.2e+46	1.4e+47	2.5e+37	7.7e+47	1174.64	30.24	1109.24	1237.86	20907.63	7558.71	8807.25	36343.18

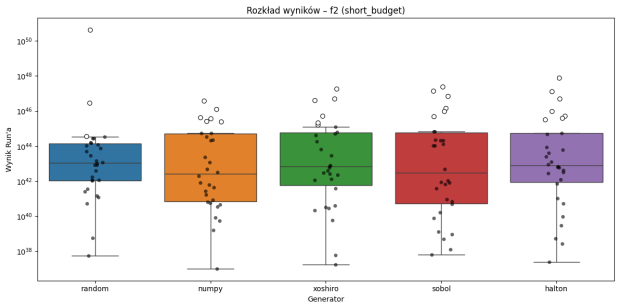
Generator / Funkcja	f17				f23				f29			
	mean	*std*	*min*	*max*	*mean*	*std*	*min*	*max*	*mean*	*std*	*min*	*max*
Random	3862.41	974.38	2851.91	7926.73	3265.24	80.93	3126.22	3435.78	7439.95	1617.43	5219.60	12124.87
Numpy	3767.38	1218.00	2681.63	8861.32	3227.24	108.82	3035.23	3579.13	6643.30	1015.15	4964.84	9239.60
Xoshiro	3457.64	382.03	2531.68	4325.71	3267.53	93.23	3116.74	3462.21	7169.70	1751.91	5367.37	12414.79
Sobol	3489.63	332.69	2959.79	4223.05	3245.84	90.69	3082.80	3468.41	7071.53	1696.61	5671.51	14302.25
Halton	3422.01	397.07	2867.96	4890.23	3242.17	102.26	3070.27	3472.49	7119.03	2804.02	5018.59	20781.25

Analizując rezultaty optymalizacji w krótkoterminowym eksperymencie, nie można zauważyć wyraźnej przewagi konkretnego generatora. Dla większości funkcji wyniki uzyskane

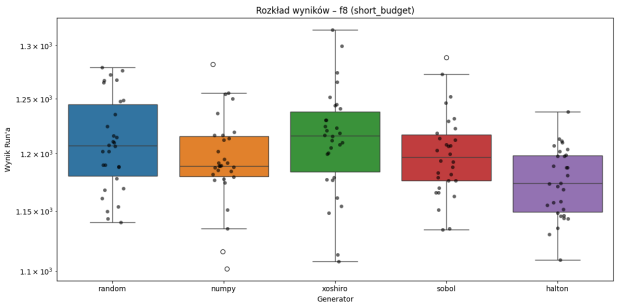
przez wszystkie generatory są zbliżone. Przykładowo dla funkcji *f23* wszystkie generatory osiągnęły średnie wyniki w przedziale [3327.24, 3367.53], przy podobnych wartościach skrajnych i odchyleniach standardowych – różnice są marginalne.

Pojedyncze przypadki różnic nie mają charakteru globalnego. Przykładowo, w funkcji *f2* generator Random osiągnął wynik średni rzędu $1.4e+49$, podczas gdy Numpy uzyskał $2.2e+45$, co wydaje się dużą różnicą – jednak ta tendencja nie powtarza się w kolejnych funkcjach testowych.

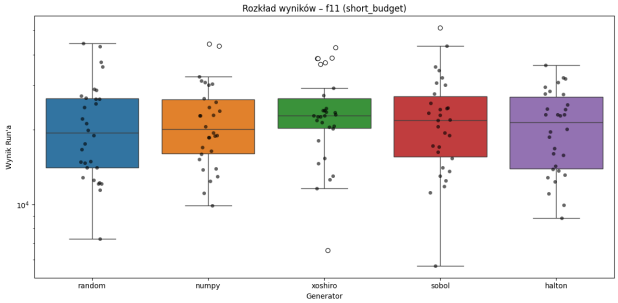
Wyniki z każdej iteracji przedstawione na wykresach pudełkowych, są następujące.



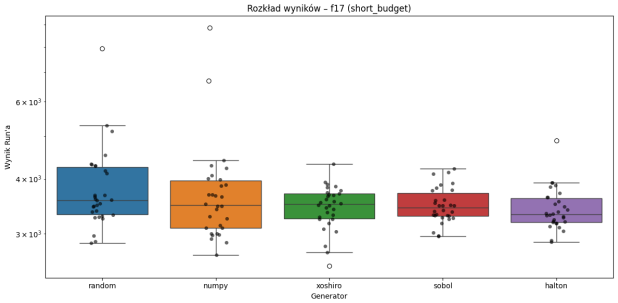
(a) f2



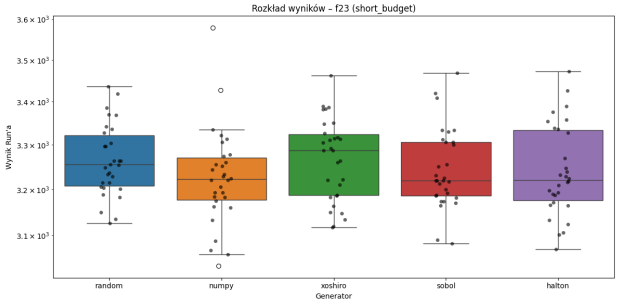
(b) f8



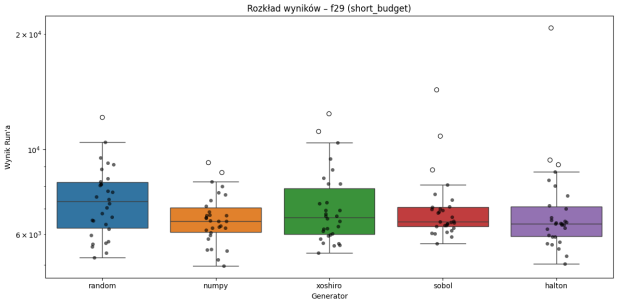
(c) f11



(d) f17



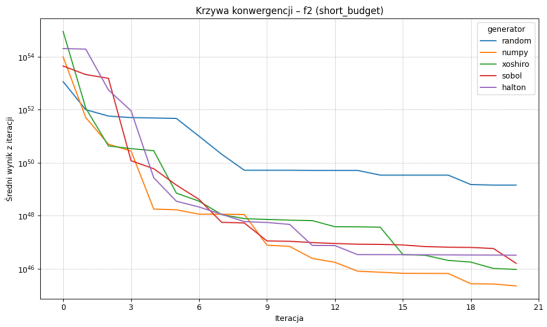
(e) f23



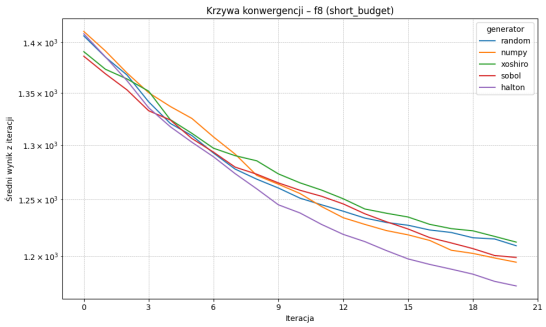
(f) f29

Rysunek 1: Wykresy pudełkowe dla funkcji testowych w krótkim budżecie

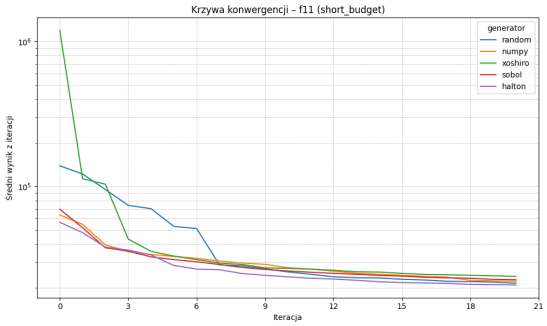
Przebieg optymalizacji każdej funkcji testowej dostępny jest na poniższych wykresach zestawiających krzywe konwergencji.



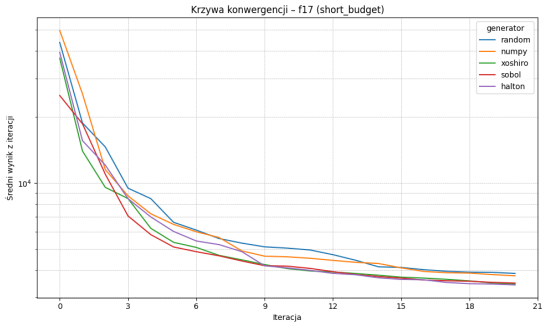
(a) f2



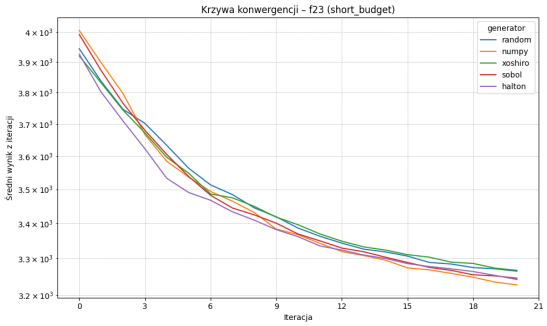
(b) f8



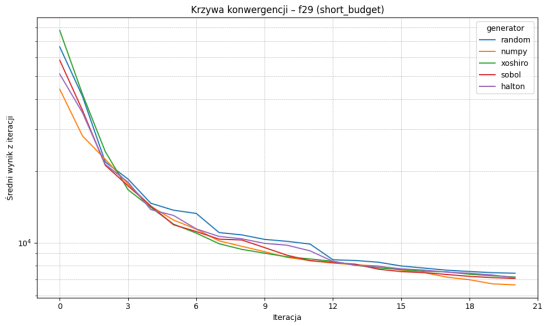
(c) f11



(d) f17



(e) f23



(f) f29

Rysunek 2: Krzywe konwergencji dla funkcji testowych w krótkim budżecie

Analiza wykresów pudełkowych i krzywych konwergencji dla krótkiego budżetu obliczeniowego nie wykazuje jednoznacznej i trwałej przewagi któregokolwiek z badanych generatorów liczb losowych.

Choć w niektórych przypadkach zauważalne są lokalne różnice - np. bardzo szeroki rozkład wyników generatora Xoshiro w funkcji *f8* lub odstający od reszty przebieg optymalizacji

f2 z wykorzystaniem Random, to nie wykazują one charakteru globalnego i nie powtarzają się systematycznie w innych zadaniach testowych.

Wyniki większości generatorów mieściły się w bardzo zbliżonych zakresach, co wskazuje na podobną skuteczność niezależnie od przyjętej metody generowania liczb losowych. Również porównanie rozrzutów oraz przebiegów optymalizacji nie pozwala na jednoznaczną klasyfikację generatorów jako bardziej lub mniej stabilnych.

4.1.2 Wyniki czasowe

Tabela 4: Zestawienie statystyk z wyników czasowych (w sekundach) funkcji w krótkim budżecie

Generator / Funkcja	**f2**		**f8**		**f11**	
	mean	*std*	*mean*	*std*	*mean*	*std*
Random	0.090	0.008	0.177	0.018	0.110	0.003
Numpy	0.107	0.024	0.164	0.007	0.110	0.002
Xoshiro	0.095	0.010	0.163	0.005	0.112	0.006
Sobol	0.164	0.032	0.202	0.005	0.149	0.004
Halton	0.227	0.017	0.289	0.003	0.233	0.005

Generator / Funkcja	**f17**		**f23**		**f29**	
	mean	*std*	*mean*	*std*	*mean*	*std*
Random	0.190	0.002	0.260	0.004	0.371	0.003
Numpy	0.192	0.005	0.261	0.003	0.376	0.006
Xoshiro	0.191	0.003	0.263	0.005	0.377	0.006
Sobol	0.228	0.004	0.300	0.002	0.410	0.004
Halton	0.313	0.005	0.387	0.005	0.501	0.008

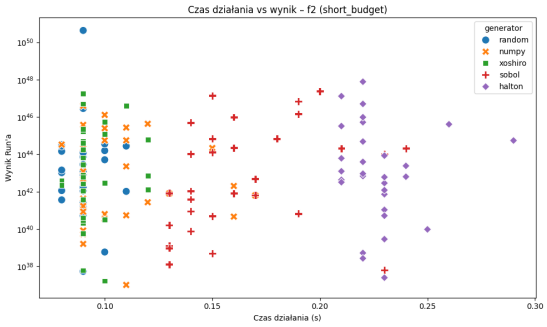
Na podstawie przedstawionych wyników czasowych można zauważyć, że quasi-losowe generatory cechują się wyraźnie dłuższym czasem działania, w eksperymencie krótkoterminowym. Zarówno Sobol, jak i Halton charakteryzują się konsekwentnie wyższym średnim czasem wykonania w porównaniu do klasycznych generatorów pseudolosowych.

Halton jest najwolniejszym generatorem w całym zestawieniu. Dla każdej z testowanych funkcji Halton zajmuje najwięcej czasu, niezależnie od charakterystyki funkcji celu. Może to wynikać z kosztów obliczeniowych generowania wartości w sekwencji Haltona, szczególnie w przestrzeniach o większej liczbie wymiarów.

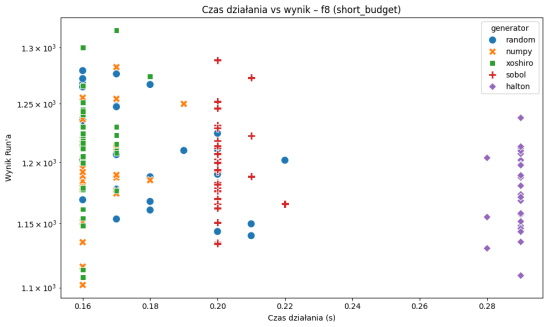
Wszystkie badane generatory PRNG osiągają bardzo podobne do siebie wyniki. Najczęściej najlepszą średnią uzyskuje Random, jednak są to różnice jedynie w tysięcznych częściach sekundy.

Wszystkie generatory wykazują bardzo małe odchylenia standardowe, co oznacza wysoką powtarzalność wyników czasowych.

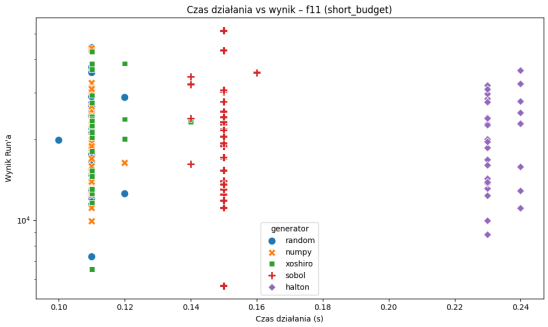
Wyniki czasowe w odniesieniu do wyników optymalizacji prezentują się następująco.



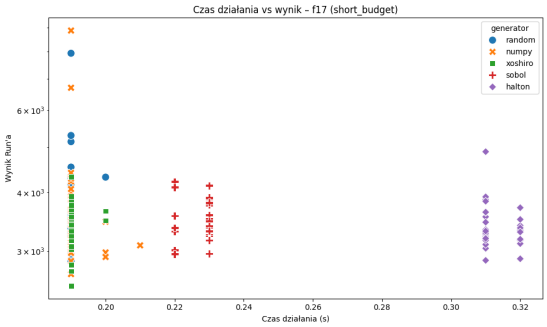
(a) f2



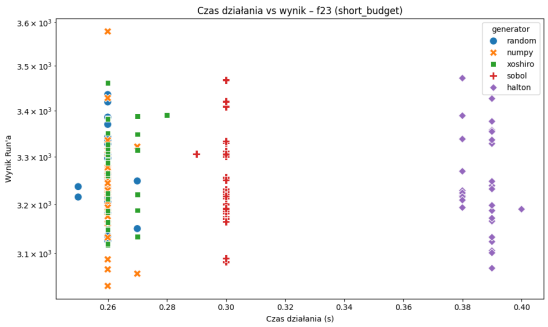
(b) f8



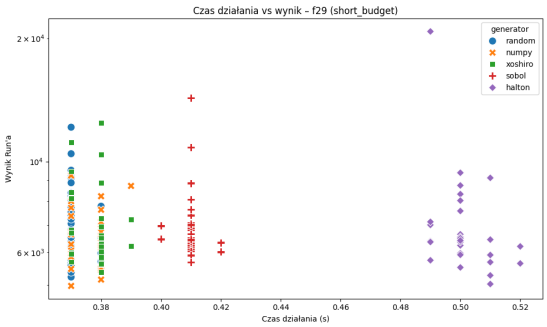
(c) f11



(d) f17



(e) f23



(f) f29

Rysunek 3: Zestawienie wyników czasowych i wyników optymalizacji funkcji testowych w krótkim budżecie

Powyższe wykresy dobrze obrazują zwiększony koszt obliczeniowy generatorów QRNG, w szczególności generatora Halton. Dłuższy czas działania nie wydaję się jednak przekładać na jednoznacznie lepsze rezultaty optymalizacji.

4.1.3 Wyniki istotności statystycznej

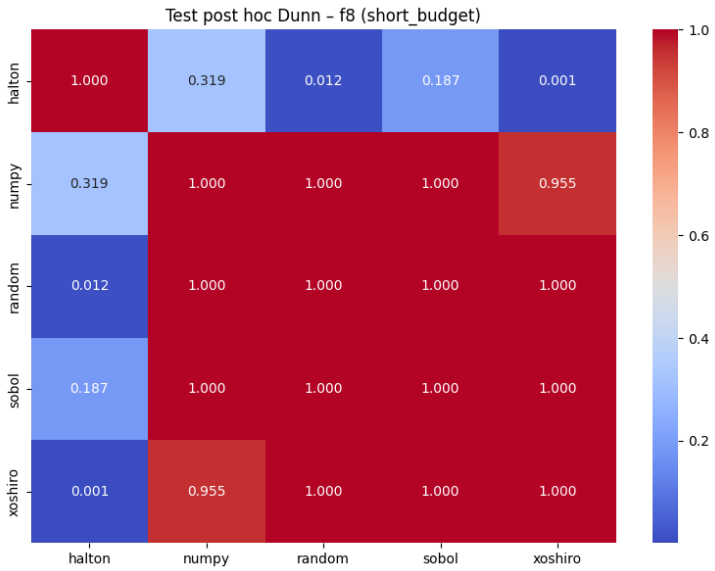
Tabela 5: Wyniki testu Kruskala-Wallisa dla funkcji testowych w krótkim budżecie

Funkcja / Parametr	**H_statistic**	**p_value**
f2	0.644	0.958
f8	17.015	0.002
f11	1.805	0.772
f17	5.668	0.225
f23	4.285	0.369
f29	4.77	0.312

Test Kruskala-Wallisa wskazuje, że istotność statystyczna została stwierdzona tylko dla funkcji *f8* ($p < 0.05$), co oznacza, że dla tej funkcji przynajmniej jeden z generatorów uzyskał wyniki różniące się znacząco od pozostałych.

Dla pozostałych funkcji uzyskane wartości p_value są znacznie większe od przyjętego poziomu istotności co oznacza, że nie stwierdzono istotnych różnic pomiędzy wynikami generatorów.

Wynik testu post hoc dla funkcji *f8* jest następujący.



Rysunek 4: Mapa cieplna wyniku testu post hoc dla funkcji testowej *f8* w krótkim budżecie

Mapa cieplna przedstawia macierz wartości *p* dla par generatorów, wyznaczonych testem post hoc Dunn (z korektą Bonferroniego), który został przeprowadzony po stwierdzeniu istotnych różnic przez test Kruskala-Wallisa.

Statystycznie istotne różnice ($p < 0.05$) występują dla par Halton × Random oraz Halton × Xoshiro.

W obu przypadkach przewagę można przypisać generatorowi Haltona, który – odwołując się do wcześniej zaprezentowanych wyników – uzyskiwał niższe wartości funkcji celu.

Chociaż ogólna analiza wskazywała na podobieństwo wyników między generatorami, Halton wyróżnił się istotnie pozytywnie względem dwóch innych generatorów w kontekście funkcji *f8* przy krótkim budżecie. To może sugerować jego lepsze zachowanie na tym typie funkcji, przynajmniej przy ograniczonej liczbie ewaluacji.

4.2 Długoterminowy eksperyment (long_budget)

4.2.1 Wyniki optymalizacji

Tabela 6: Zestawienie statystyk z wyników optymalizacji funkcji w długim budżecie

Generator / Funkcja	f2				f8				f11			
	mean	*std*	*min*	*max*	*mean*	*std*	*min*	*max*	*mean*	*std*	*min*	*max*
Random	1.4e+28	7.8e+28	1.0e+20	4.3e+29	1114.82	53.52	1007.83	1214.34	1674.33	285.72	1302.02	2322.15
Numpy	6.9e+26	1.0e+27	9.3e+19	3.0e+27	1129.34	45.79	1057.43	1220.94	1719.20	346.10	1363.89	3206.63
Xoshiro	3.6e+31	1.9e+32	2.4e+20	1.0e+33	1129.88	52.59	981.48	1206.41	1644.43	248.42	1322.62	2340.58
Sobol	5.2e+27	1.3e+28	9.1e+20	5.7e+28	1118.05	49.44	1045.38	1245.11	1589.49	237.35	1343.46	2465.32
Halton	1.0e+28	4.0e+28	1.5e+20	2.1e+29	1111.94	42.25	1029.12	1215.52	1672.00	206.71	1313.20	2071.21

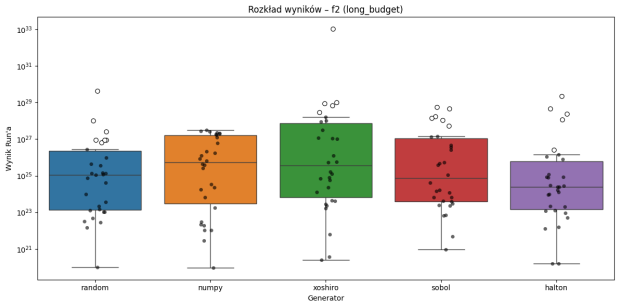
Generator / Funkcja	f17				f23				f29			
	mean	*std*	*min*	*max*	*mean*	*std*	*min*	*max*	*mean*	*std*	*min*	*max*
Random	2611.14	287.42	1927.69	3138.14	2997.09	96.19	2834.26	3278.69	4992.15	455.34	4219.57	5898.74
Numpy	2478.50	268.07	1830.93	2951.59	2995.35	94.01	2838.85	3178.08	4851.49	382.17	3915.09	5777.25
Xoshiro	2590.81	277.84	1931.18	3057.81	2978.38	80.19	2803.63	3179.35	5071.32	476.70	4187.23	6208.89
Sobol	2682.02	227.42	2287.80	3283.15	2988.50	84.92	2848.80	3237.22	5110.36	462.96	4386.95	6394.85
Halton	2695.84	279.98	2054.69	3142.88	3020.88	82.14	2891.07	3174.99	4869.84	476.21	4027.18	5975.66

W przypadku wydłużonego budżetu ewaluacyjnego ponownie nie zaobserwowano jednoznacznej dominacji żadnego z testowanych generatorów. Wyniki wskazują na dużą stabilność algorytmu niezależnie od źródła losowości, a różnice między generatorami są często niewielkie i niespójne między funkcjami.

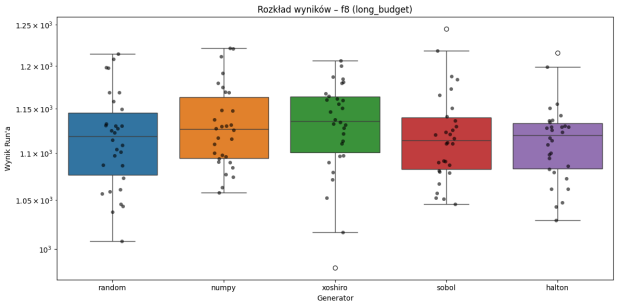
Zauważyć można pewne różnice w wynikach (np. dla funkcji *f2*, gdzie najlepiej wypada generator Numpy, natomiast Xoshiro znacząco odstaje lub dla funkcji *f11* gdzie dobre wyniki osiąga Sobol), jednak są to jednostkowe przypadki, które nie mają odzwierciedlenia w rezultatach dla pozostałych funkcji.

Warto zaznaczyć, że wszystkie uzyskane wyniki są lepsze od rezultatów z krótkiego budżetu co wskazuje na poprawne działanie optymalizatora.

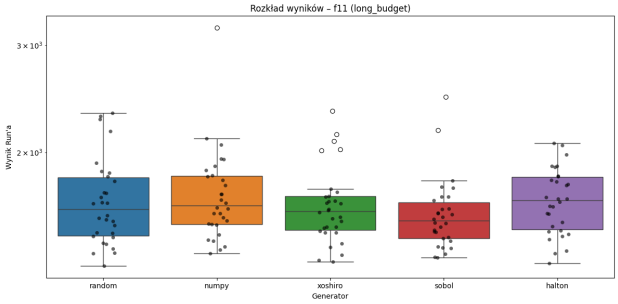
Wyniki z każdej iteracji przedstawione na wykresach pudełkowych, są następujące.



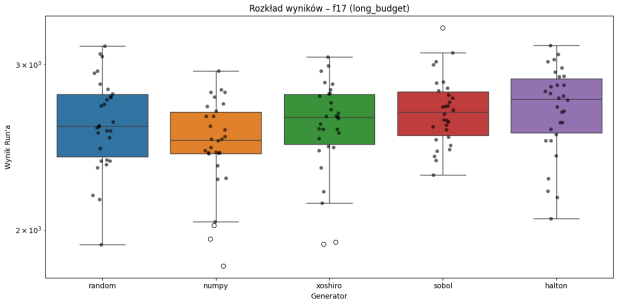
(a) f2



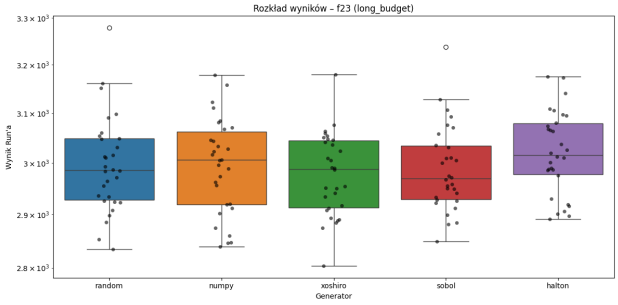
(b) f8



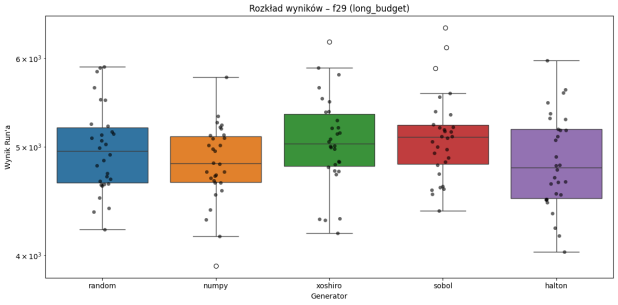
(c) f11



(d) f17



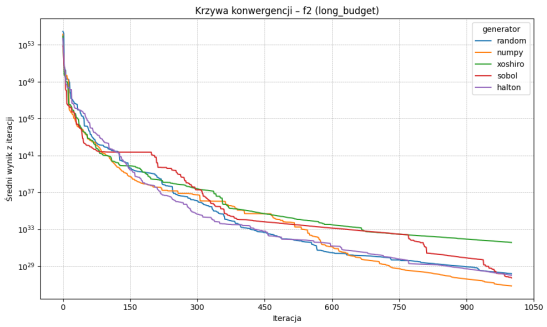
(e) f23



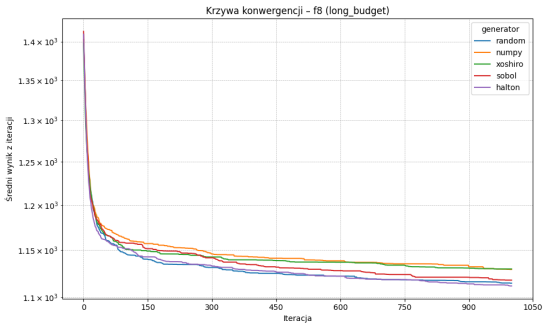
(f) f29

Rysunek 5: Wykresy pudełkowe dla funkcji testowych w długim budżecie

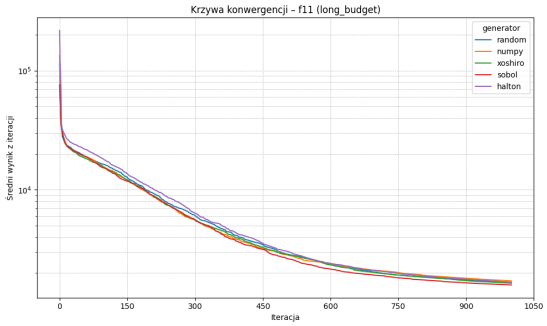
Przebieg optymalizacji każdej funkcji testowej, dostępny jest na poniższych wykresach, zestawiających krzywe konwergencji.



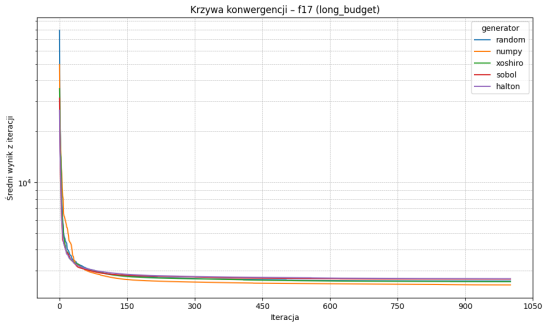
(a) f2



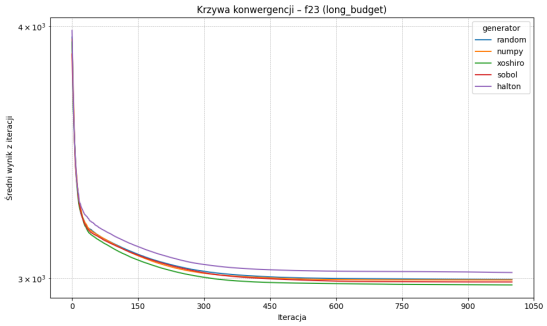
(b) f8



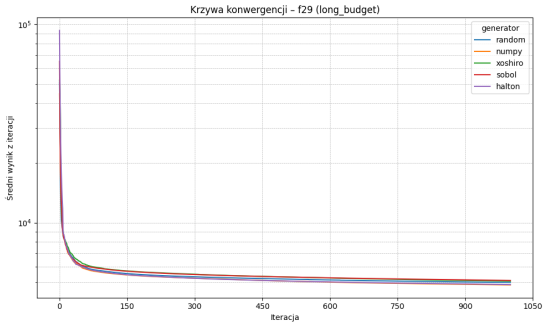
(c) f11



(d) f17



(e) f23



(f) f29

Rysunek 6: Krzywe konwergencji dla funkcji testowych w długim budżecie

Po przejściu na dłuższy budżet ewaluacji zauważalna jest stabilizacja wyników między generatorami. Różnice obserwowane w krótkim budżecie ulegają spłaszczeniu – mediany wyników poszczególnych generatorów dla większości funkcji testowych znajdują się blisko siebie, a rozrzut danych zmniejsza się, szczególnie w funkcjach o niższej złożoności.

Niektóre funkcje (np. *f2*, *f8*) wykazują nadal pewne zróżnicowanie, niemniej, ogólny obraz

pozostaje nie wskazuje, że któraś z metod dominuje wyraźnie we wszystkich przypadkach.

W dłuższym horyzoncie czasowym różnice w skuteczności generatorów ulegają wyraźnemu zatarciu. Wszystkie generatory wykazują zbliżoną tendencję do stabilnej konwergencji – ich średnie wartości błędu maleją regularnie, a różnice między nimi stopniowo się zmniejszają. Najbardziej zróżnicowane rezultaty występują w funkcji *f2* – ale nawet tam przewagi jednych generatorów są tylko okresowe.

4.2.2 Wyniki czasowe

Tabela 7: Zestawienie statystyk z wyników czasowych (w sekundach) funkcji w długim budżecie

Generator / Funkcja	**f2**		**f8**		**f11**	
	mean	*std*	*mean*	*std*	*mean*	*std*
Random	4.042	0.013	7.870	0.061	5.370	0.067
Numpy	4.158	0.024	7.936	0.047	6.130	0.413
Xoshiro	4.183	0.040	8.197	0.127	5.812	0.187
Sobol	5.933	0.035	10.149	0.496	7.843	0.523
Halton	10.333	0.056	14.393	0.305	12.466	0.428

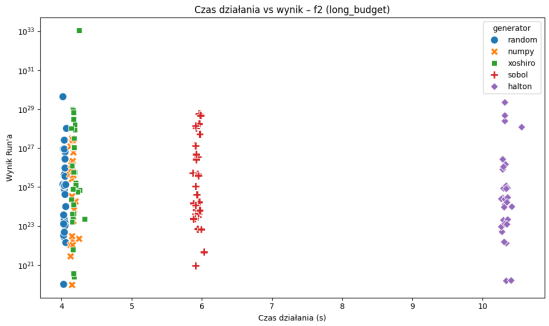
Generator / Funkcja	**f17**		**f23**		**f29**	
	mean	*std*	*mean*	*std*	*mean*	*std*
Random	9.230	0.090	12.751	0.269	18.233	0.401
Numpy	9.543	0.276	13.225	0.251	18.893	0.576
Xoshiro	9.810	0.405	13.341	0.654	19.151	0.875
Sobol	11.693	0.286	14.717	0.404	20.640	0.518
Halton	16.103	0.442	19.620	1.034	24.547	0.225

Po raz kolejny, tendencje uzyskane przy krótkim budżecie powtarzają się. Najkrótszy czas działania osiągają wszystkie trzy generatory pseudolosowe – przy czym Random jest ponownie najszybszy dla większości funkcji.

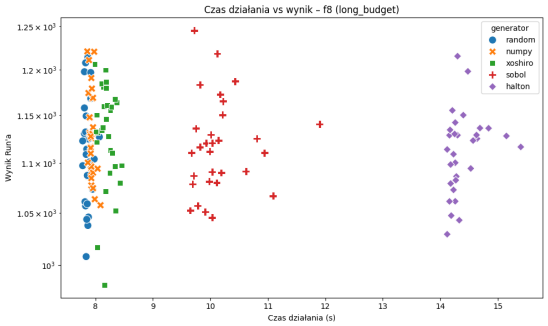
Quasi-losowe generatory, takie jak Sobol i Halton, są znacząco wolniejsze. Halton jest bezsprzecznie najwolniejszy we wszystkich przypadkach, osiągając czasy ponad dwukrotnie większe niż PRNG – np. dla funkcji *f11* średni czas działania generatora Halton to aż 12.466 s, w porównaniu do 6 s dla generatorów PRNG.

Najniższe odchylenia standardowe, oznaczające stabilność czasową generatora, w większości przypadków osiąga Random. Dobrze wypada również generator Numpy. Dla pozostałych wartości odchyleń standardowych ciężko doszukać się wyraźnych trendów, np. wyniki Haltona są najwyższe dla *f2, f17* i *f23*, za to najniższe dla *f29*.

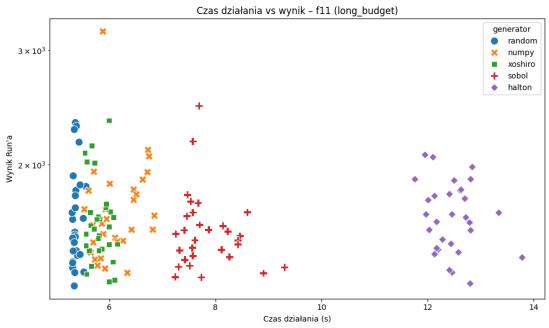
Wyniki czasowe w odniesieniu do wyników optymalizacji prezentują się następująco.



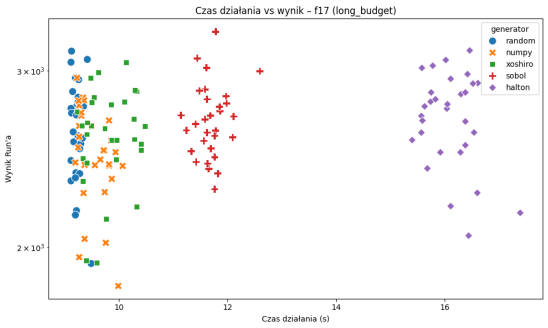
(a) f2



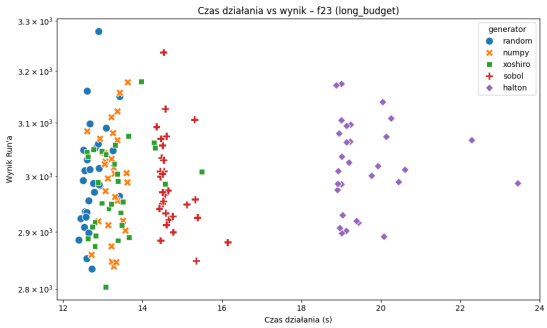
(b) f8



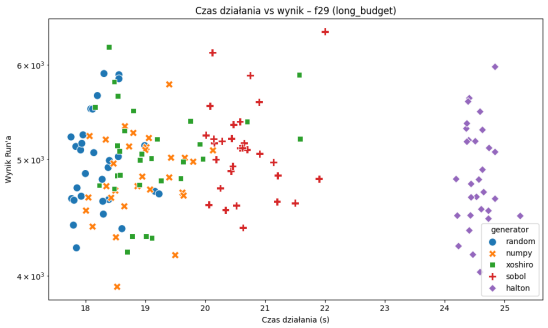
(c) f11



(d) f17



(e) f23



(f) f29

Rysunek 7: Zestawienie wyników czasowych i wyników optymalizacji funkcji testowych w długim budżecie

Zależność między czasem działania a jakością uzyskiwanego rozwiązania pozostaje spójna z obserwacjami z krótkiego budżetu. Pomimo istotnie wyższego czasu, generatory QRNG nie przynoszą systematycznej poprawy wyników.

Z punktu widzenia efektywności czasowej, generatory PRNG wydają się być bardziej opłacalne — generują rozwiązania równie dobre, przy mniejszym koszcie czasowym.

4.2.3 Wyniki istotności statystycznej

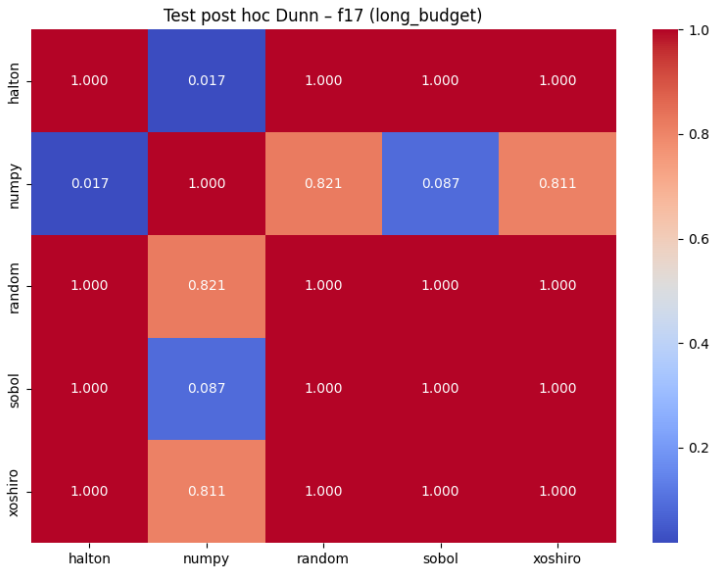
Tabela 8: Wyniki testu Kruskala-Wallisa dla funkcji testowych w długim budżecie

Funkcja / Parametr	**H_statistic**	**p_value**
f2	4.609	0.33
f8	4.898	0.298
f11	5.089	0.278
f17	11.392	0.022
f23	4.103	0.392
f29	6.722	0.151

Wynik testu w długoterminowym eksperymencie nie pokrywa się z wynikiem z eksperymentu krótkoterminowego. Poprzednio uzyskana istotność statystyczna dla funkcji *f8* nie jest teraz wykazana. Oznaczać to może, że taka istotność jest zatracana przy większej liczbie iteracji lub była wynikiem przypadku.

Tym razem tylko funkcja *f17* wykazała istotne statystycznie różnice pomiędzy generatorami.

Wynik testu post hoc dla funkcji *f17* jest następujący.



Rysunek 8: Mapa cieplna wyniku testu post hoc dla funkcji testowej *f17* w długim budżecie

Dla tego przypadku występuje tylko jedna istotna różnica między parą Halton × Numpy. Biorąc pod uwagę zaprezentowane wyniki optymalizacji, można stwierdzić, że dla funkcji *f17* Numpy osiągnął nieco lepsze wyniki. Dominacja ta nie rozciąga się jednak na inne

porównania, co wzmacnia ogólny wniosek o braku wyraźnego i konsekwentnego lidera wśród badanych generatorów w długim budżecie.

Ten typ wyniku dobrze pokazuje granice wpływu wyboru generatora – lokalnie może on robić różnicę, ale globalnie nie przekłada się to na jednoznaczną przewagę.

5 Podsumowanie

Przeprowadzone eksperymenty nie pozwalają na jednoznaczne wskazanie generatora liczb, który systematycznie zapewniałby lepsze wyniki optymalizacji w kontekście wykorzystanego algorytmu ewolucyjnego. Zarówno analiza wyników tabelarycznych, jak i wizualizacje w postaci wykresów pudełkowych, czy krzywych konwergencji nie wykazały istotnych przewag któregoś z generatorów.

W przypadkach, w których dany generator osiągał lepsze rezultaty dla konkretnej funkcji testowej, efekt ten nie był widoczny w ujęciu globalnym – nie przenosił się na inne funkcje. Sugeruje to, że to charakterystyka konkretnego problemu optymalizacyjnego, a nie zastosowany mechanizm losowości, może odgrywać kluczową rolę w skuteczności poszukiwań.

Z perspektywy czasowej najwolniejszym generatorem okazał się konsekwentnie Halton, a następnie Sobol – oba należące do klasy quasi-losowych. Najszybszym rozwiązaniem był zazwyczaj klasyczny pseudolosowy generator Random, co może mieć znaczenie przy dużych budżetach obliczeniowych lub zastosowaniach w czasie rzeczywistym.

Z punktu widzenia analizy statystycznej, interesującym rezultatem było pojawienie się istotnych statystycznie różnic w parach Halton × Random/Xoshiro (dla funkcji f8 i krótkoterminowego eksperymentu) oraz Halton × Numpy (dla funkcji f17 i długoterminowego eksperymentu) – a więc pomiędzy generatorami quasi-losowymi i pseudo-losowymi. Choć trudno jednoznacznie rozstrzygnąć, czy nie są to przypadki losowe, fakt ten może sugerować istnienie subtelnych, ale zauważalnych różnic między tymi dwiema klasami generatorów.

Podobne obserwacje zostały odnotowane również w literaturze naukowej. Przykładowo, w artykule „Population initialization techniques for evolutionary algorithms for single-objective constrained optimization problems: Deterministic vs. stochastic techniques” [12] przeprowadzono badania porównawcze różnych metod inicjalizacji populacji w algorytmach: PSO, DE, GWO i ABC, z wykorzystaniem funkcji testowych z benchmarku CEC 2017. Autorzy stwierdzili, że przy wystarczającej liczbie iteracji, algorytmy ewolucyjne nie są wrażliwe na metody inicjalizacji i nie ma znaczących różnic między wybranymi metodami inicjalizacji populacji. Zbliżone wnioski pojawiły się również w badaniu nad algorytmem genetycznym autorstwa A. Reese [13], gdzie również nie odnaleziono jednego optymalnego dla badanych problemów

oraz podkreślono, że wybór najlepszego generatora liczb losowych zależy od wielu czynników, takich jak liczba zmiennych, typ funkcji, przestrzeń poszukiwań czy pożądana dokładność. Chociaż wspomniane prace nie odwzorowują dokładnie tego samego przypadku testowego, co niniejsze badania, ich wyniki potwierdzają ogólną tendencję do braku wyraźnie lepszego generatora liczb losowych w opracowywanych problemach optymalizacji. Uzyskane w tym badaniu rezultaty wydają się więc być zgodne z ustaleniami z literatury i dostarczają dodatkowego potwierdzenia, że wpływ generatora liczb losowych na końcową jakość rozwiązań algorytmu ewolucyjnego nie musi być znaczący.

Bibliografia

[1] N. H. Awad, M. Z. Ali, P. N. Suganthan, J. J. Liang i B. Y. Qu. *Problem Definitions and Evaluation Criteria for the CEC 2017 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization*. Technical Report. Nanyang Technological University, Jordan University of Science, Technology, i Zhengzhou University, 2016.

[2] Makoto Matsumoto i Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". W: *ACM Trans. Model. Comput. Simul.* 8.1 (sty. 1998), 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: https://doi.org/10.1145/272991.272995.

[3] Python Software Foundation. *The Python Standard Library – `random` module*. Python 3.12 documentation. 2023. URL: https://docs.python.org/3/library/random.html.

[4] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Spraw. tech. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, wrz. 2014.

[5] Charles R. Harris i in. "Array programming with NumPy". W: *Nature* 585.7825 (wrz. 2020), s. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[6] David Blackman i Sebastiano Vigna. *Scrambled Linear Pseudorandom Number Generators*. 2022. arXiv: 1805.01407 [cs.DS]. URL: https://arxiv.org/abs/1805.01407.

[7] Kevin Sheppard. *Randomgen: A generator library compatible with NumPy and Numba*. https://bashtage.github.io/randomgen/. Accessed: 2025-05-31. 2023.

[8] I.M Sobol'. "On the distribution of points in a cube and the approximate evaluation of integrals". W: *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967), s. 86–112. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(67)90144-9. URL: https://www.sciencedirect.com/science/article/pii/0041555367901449.

[9] Pauli Virtanen i in. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". W: *Nature Methods* 17 (2020), s. 261–272. DOI: 10.1038/s41592-019-0686-2.

[10] J. H. Halton. “On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals”. W: *Numerische Mathematik* 2.1 (1960), s. 84–90. ISSN: 0945-3245. DOI: 10.1007/BF01386213. URL: https://doi.org/10.1007/BF01386213.

[11] Python Software Foundation. *The Python Standard Library – `secrets` module*. Python 3.12 documentation. 2023. URL: https://docs.python.org/3/library/secrets.html.

[12] Alaa Tharwat i Wolfram Schenck. “Population initialization techniques for evolutionary algorithms for single-objective constrained optimization problems: Deterministic vs. stochastic techniques”. W: *Swarm and Evolutionary Computation* 67 (2021), s. 100952. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2021.100952. URL: https://www.sciencedirect.com/science/article/pii/S2210650221001140.

[13] Andrea Reese. “Random number generators in genetic algorithms for unconstrained and constrained optimization”. W: *Nonlinear Analysis: Theory, Methods Applications* 71.12 (2009), e679–e692. ISSN: 0362-546X. DOI: https://doi.org/10.1016/j.na.2008.11.084. URL: https://www.sciencedirect.com/science/article/pii/S0362546X08007360.