Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka techniczna (ITE)**
Specjalność: **Grafika i systemy multimedialne (IGM)**

PRACA DYPLOMOWA INŻYNIERSKA

Porównanie efektywności implementacji w językach Julia i Java algorytmu genetycznego dla problemu komiwojażera

A comparison of the efficiency of implementing a genetic algorithm for the travelling salesman problem in Julia and Java

Krzysztof Fokczyński

Opiekun pracy
Dr inż. Marcin Łopuszyński

Słowa kluczowe: Julia, Java, Algorytm genetyczny, Problem komiwojażera

WROCŁAW, 2025

Streszczenie

Praca dotyczy porównania wydajności implementacji algorytmu genetycznego w Julii oraz Javie dla problemu komiwojażera pod względem wydajności czasowej oraz szczytowej zajętości pamięci. W obu językach zostały zaimplementowane równoważne wersje algorytmu. Badania zostały przeprowadzone na wybranych instancjach problemu komiwojażera. W pierwszym etapie badań wyłonione zostały konfiguracje algorytmu pozwalające na osiąganie najlepszych wyników pod względem kosztu ścieżki. Wybrane konfiguracje zostały poddane drugiemu etapowi który skupiał się na pomiarach czasów wykonania oraz szczytowej zajętości pamięci. Wyniki zostały przeanalizowane z wykorzystaniem testów statystycznych. Uzyskane wyniki wykazały, iż Julia uzyskuje istotnie niższy czas wykonania dla wszystkich badanych instancji oraz konfiguracji. W przypadku badań szczytowego zużycia pamięci Julia również wykazała korzystniejsze wyniki, lecz z uwagi na charakterystykę wykorzystanych metod pomiaru pamięci interpretacja wyników jest utrudniona. Badania sugerują, iż Julia może być atrakcyjną alternatywą dla Javy w zastosowaniach typowo numerycznych, oferując lepszy czas wykonania oraz potencjalnie lepszą szczytową zajętość pamięci.

Słowa kluczowe: Julia, Java, Algorytm genetyczny, Problem komiwojażera

Abstract

This work concerns the comparison of the performance of a genetic algorithm implementation in Julia and Java for the traveling salesman problem in terms of execution time and peak memory usage. Equivalent versions of the algorithm were implemented in both languages. The experiments were conducted on selected instances of the traveling salesman problem. In the first stage of the study, algorithm configurations that achieved the best results in terms of path cost were identified. The selected configurations were then subjected to a second stage focusing on measurements of execution time and peak memory usage. The results were analyzed using statistical tests. The results showed that Julia achieved significantly lower execution times for all tested instances and configurations. In terms of peak memory usage, Julia also demonstrated more favorable results, however, due to the nature of the memory measurement methods used, the interpretation of these results is more challenging. The study suggests that Julia may be an attractive alternative to Java in numerically intensive applications, offering better execution times and potentially lower peak memory usage.

Keywords: Julia, Java, Genetic algorithm, Traveling salesman problem

Spis treści

Wstęp **4**
1. Przegląd wybranej literatury dotyczącej wykorzystania algorytmu genetycznego w optymalizacji problemu komiwojażera **6**
1.1. Wprowadzenie 6
1.2. Charakterystyka problemu komiwojażera 6
1.3. Algorytm genetyczny 7
1.4. Reprezentacja instancji 8
1.5. Inicjalizacja Populacji 9
1.6. Funkcja celu 11
1.7. Strategia selekcji 11
1.8. Strategia reprodukcji 13
1.9. Strategia zastępowania 17
1.10. Kryterium stopu 17
1.11. Podsumowanie 17
2. Opis implementacji algorytmu **19**
2.1. Wprowadzenie 19
2.2. Porównanie Julii oraz Javy 19
2.3. Generator liczb losowych 21
2.4. Opis implementacji algorytmu 24
2.5. Instancje problemu komiwojażera 32
3. Analiza otrzymanych wyników **34**
3.1. Wprowadzenie 34
3.2. Metoda badawcza 34
3.3. Warunki wstępne symulacji 37
3.4. I etap badań 39
3.5. II etap badań 42
Podsumowanie **58**
Literatura **60**

Wstęp

Rozwiązywanie złożonych problemów optymalizacyjnych stanowi podstawę współczesnej informatyki. Algorytmy optymalizacyjne pozwalają na lepsze wykorzystanie zasobów w takich dziedzinach jak robotyka, automatyzacja, produkcja czy transport. Problem komiwojażera jest klasycznym przykładem takiego zagadnienia. Wraz ze wzrostem rozmiaru problemu czas jego rozwiązania za pomocą algorytmów dokładnych wzrasta bardzo szybko. Z tego powodu do rozwiązywania tego rodzaju problemów zaczęto wykorzystywać metody przybliżone, takie jak algorytm genetyczny.

Algorytm Genetyczny w rozsądnym czasie pozwala osiągnąć zadowalające rozwiązanie, nawet dla dużych instancji problemu komiwojażera. Nazwa algorytmu nie jest przypadkowa, ponieważ symuluje on zjawiska występujące w biologii. Selekcja osobników, krzyżowanie i mutacje są kluczowym elementem który pozwala odkrywać coraz to lepsze rozwiązania, lecz również wymaga dużej ilości obliczeń jak i pamięci. Wobec obserwowanego wygasania prawa Moore'a, które zdaniem niektórych przestało obowiązywać, przyrost wydajności procesorów zaczyna być wolniejszy. Z tego powodu większą wagę należy przyłożyć do wydajności oprogramowania, dlatego istotnym czynnikiem poza samym algorytmem jest również dobór języka programowania.

Obecnie jednym z najbardziej popularnych języków programowania na świecie jest Java. Jest to spowodowane stabilnością oraz dojrzałością całego ekosystemu. Rozwinięta maszyna wirtualna Javy (JVM) zapewnia przenośność oprogramowania miedzy środowiskami uruchomieniowymi. Jednak duża złożoność całego środowiska Javy i mechanizmy takie jak garbage collector mogą negatywnie wpływać na wydajność.

Na przestrzeni ostatnich lat pojawiło się wiele języków programowania nastawionych na wydajność obliczeniową. Przykładem takiego języka jest Julia. Język ten został zaprojektowany typowo do zastosowań numerycznych i naukowych. Zdaniem twórców języka [7], Julia rozwiązuje Problem Dwóch Języków, eliminując potrzebę przepisywania prototypów kodu w języku wysokiego poziomu na kod w języku niskiego poziomu by osiągnąć wysoką wydajność.

Obecnie dostępne wyniki badań i eksperymentów nie wskazują jednoznacznie który z języków zapewnia większą wydajność. Syntetyczne testy wydajności (Julia Micro-Benchmarks)[2] wskazują na przewagę Julii przy czystych obliczeniach numerycznych. Z drugiej strony przedstawione przez NASA badania dotyczące symulacji lotów [22] wykazały, że w złożonym systemie obiektowym Julia była mniej wydajna od Javy. Jako przyczynę wskazano zależność wydajności Julii od znajomości szczegółów działania języka. Wskazano również iż łatwość pisania kodu w Julii jest niwelowana przez potrzebę znajomości odpowiednich praktyk i konstrukcji w celu osiągnięcia wysokiej wydajności. W społeczności programistycznej zostało to nazwane Problemem 1.5 Języka.

Głównym celem niniejszej pracy jest analiza porównawcza efektywności implementacji algorytmu genetycznego dla problemu komiwojażera w językach Julia i Java. Porównanie odbywa się na poziomie wydajności czasowej oraz szczytowej zajętości pamięci przez program.

W pracy została postawiona hipoteza badawcza, że język Julia pozwoli osiągnąć mniejszy czas wykonania algorytmu w porównaniu do języka Java. Uzasadnieniem hipotezy jest specyfika badanego problemu. Algorytm genetyczny dla problemu komiwojażera opiera się na prostych

strukturach danych oraz operacjach numerycznych, w przeciwieństwie do wysoko rozwiniętej architektury obiektowej użytej przez NASA.

Aspekt szczytowego zużycia pamięci został potraktowany jako otwarte pytanie badawcze: "Która z implementacji algorytmu genetycznego dla problemu komiwojażera charakteryzuje się mniejszym szczytowym zużyciem pamięci?". Sformułowanie zostało postawione w formie pytania otwartego a nie hipotezy ze względu na brak porównań dostępnych w literaturze naukowej.

Pierwszy rozdział niniejszej pracy zawiera przegląd wybranej literatury dotyczącej wykorzystania algorytmu genetycznego do optymalizacji problemu komiwojażera. W rozdziale drugim przedstawiono kluczowe różnice między oboma językami oraz opisano implementacje programów. W rozdziale trzecim przedstawione zostały uzyskane wyniki oraz przeprowadzone zostały badania statystyczne. Ostatni rozdział pracy zawiera podsumowanie uzyskanych wyników oraz wnioski końcowe wyciągnięte na ich podstawie.

Rozdział 1

Przegląd wybranej literatury dotyczącej wykorzystania algorytmu genetycznego w optymalizacji problemu komiwojażera

1.1. Wprowadzenie

W niniejszym rozdziale przedstawiono przegląd wybranej literatury dotyczącej algorytmu genetycznego i wykorzystania go w procesie optymalizacji problemu komiwojażera. Z uwagi na swoją prostotę a zarazem złożoność obliczeniową, problem komiwojażera stanowi istotny problem algorytmiki. Z uwagi na swoją specyfikę wykorzystywany jest jako punkt odniesienia dla badań wydajności i skuteczności algorytmów optymalizacyjnych.

Optymalizacja problemu komiwojażera w rozległy sposób została opisana w literaturze naukowej. Przedstawione zostało wiele metod różniących się złożonością obliczeniową oraz jakością rozwiązań. Istotną grupą sposobów rozwiązywania tego problemu jest grupa algorytmów heurystycznych, w tym algorytm genetyczny. Literatura naukowa zawiera liczne opracowania i badania dotyczące wykorzystywania algorytmu genetycznego dla problemu komiwojażera oraz wpływu jego poszczególnych elementów na jakość rozwiązań. Wiedza przedstawiona w niniejszym rozdziale opiera się na trzech źródłach: Luke S., Essentials of Metaheuristics [17], Michalewicz Z., D.B.Fogel, Jak to rozwiązać czyli nowoczesna heurystyka [10], El-Ghazali T., Metaheuristics: From Design to Implementation [12].

Celem niniejszego rozdziału jest prezentacja podstawowych koncepcji algorytmu genetycznego oraz jego optymalizacji. Wiedza uzyskana podczas analizy będzie stanowić podstawę do zaprojektowania i implementacji algorytmu genetycznego wykorzystywanego w kolejnych rozdziałach.

1.2. Charakterystyka problemu komiwojażera

Opis problem komiwojażera przedstawiony w poniższym podrozdziale opiera się na publikacji "Jak to rozwiązać, czyli nowoczesna heurystyka"[10].Problem komiwojażera polega na znalezieniu cyklu Hamiltona w grafie pełnym ważonym o minimalnej sumie wag. Każde miasto

musi być odwiedzone dokładnie raz, a trasa musi zakończyć się w punkcie startowym. Zagadnienie jest często przedstawiane na przykładzie obwoźnego sprzedawcy, który musi odwiedzić wszystkie miasta, pokonując minimalną drogę. Problem komiwojażera jest sklasyfikowany jako problem NP-trudny, ze względu na fakt że nie istnieją algorytmy umożliwiające znajdowanie rozwiązań idealnych ze złożonością wzrastającą wielomianowo do rozmiaru problemu. Rozmiar przestrzeni rozwiązań wzrasta wykładniczo wraz z liczbą miast, jej rozmiar wynosi $n!$[1].

Problem pierwszy raz został wspomniany przez Eurela w 1759 roku, przedstawiany wtedy był pod inną postacią, jako problem trasy konika szachowego. Polegał on na znalezieniu takiej sekwencji ruchów konika szachowego, dzięki której każde pole szachownicy zostanie odwiedzone przez niego tylko raz. Sam termin "Komiwojażer"pojawił się dopiero w 1932 roku, w książce napisanej przez obwoźnego sprzedawcę "Komiwojażer, jak i co powinien robić, żeby dostać zapłatę i odnieść sukces w swojej pracy". Problem w obecnej postaci powstał w 1948 roku za sprawą firmy Rand Corporation.

Ze względu na popularność problemu komiwojażera powstało wiele algorytmów generujących rozwiązania przybliżone. Część z nich optymalizuje i poprawia trasę wyprowadzając lokalne optymalizacje i zaburzenia. W dwóch ostatnich dekadach XX wieku problem skupił na sobie uwagę grupy naukowców zajmujących się metodami ewolucyjnymi. W krótkim okresie czasu powstało wiele prac opisujących to podejście i poszukujących idealnego algorytmu ewolucyjnego dla problemu komiwojażera.

Ważnym elementem badań wydajności algorytmów rozwiązujących problem komiwojażera jest sposób prezentacji wyników obliczeń. Czas wymagany do rozwiązania danej instancji problemu nie jest najpowszechniejszą formą prezentacji wyników. Częściej wyniki reprezentowane są przez ilość wyliczeń funkcji oceniającej. Utrudnia to porównanie jakości końcowego wyniku, a zarazem całego algorytmu, ze względu na różną złożoność czasową operatorów.

1.3. Algorytm genetyczny

Algorytm genetyczny należy do grupy algorytmów ewolucyjnych, jak podaje Luke w książce Essentials of Metaheuristics [17], ta grupa algorytmów opiera się na procesach występujących w biologii. Opierają się one na tworzeniu nowych populacji rozwiązań, na podstawie poprzednich populacji. Poza wspomnianym algorytmem genetycznym do grupy algorytmów ewolucyjnych należą również strategie ewolucyjne.

Algorytm genetyczny został stworzony przez Johna Hollanda w 1970 roku. Luke podaje, że algorytm genetyczny jest mocno zbliżony do strategii ewolucyjnej (μ, λ) w kontekście oceny populacji, selekcji, rozmnażania oraz tworzenia nowych populacji. Autor również wskazuje główną różnicą między algorytmem genetycznym i strategią ewolucyjną. W algorytmie genetycznym iteracyjnie odbywa się selekcja rodziców, a później krzyżowanie aż zostanie stworzona wymagana liczba dzieci. W odróżnieniu od strategi ewolucyjnej(μ, λ) gdzie najpierw następuje selekcja całej puli rodziców, a później tworzona jest populacja dzieci całkowicie zastępująca rodziców. W algorytmie genetycznym selekcja i krzyżowanie odbywa się stopniowo. Pusta populacja dzieci jest zapełniana poprzez wybór dwóch rodziców i ich krzyżowanie. Proces kończy się w momencie, gdy populacja osiągnie pożądany rozmiar. Jednak warto zwrócić uwagę na fakt, że w wielu współczesnych implementacjach stosowane są schematy selekcji i reprodukcji zbliżone do tych z strategi ewolucyjnej.

W swojej książce El-Ghazali [12] przedstawił najważniejsze elementy algorytmów ewolucyjnych, kluczowe na etapie projektowania:

- reprezentacja,
- inicjalizacja populacji,
- funkcja celu,
- strategia selekcji,
- strategia reprodukcji,
- strategia zastępowania,
- kryterium stopu,

W kolejnych podrozdziałach powyższe pojęcia zostaną rozwinięte i szczegółowo opisane na podstawie dostępnej literatury w kontekście problemu komiwojażera.

```
popsize ← desired population size

P ← ∅
for popsize times do
    P ← P ∪ {new random individual}

Best ← □
repeat

    for each individual P_i in P do
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    end for

    Q ← ∅

    for popsize / 2 times do
        Parent_a ← SelectWithReplacement(P)
        Parent_b ← SelectWithReplacement(P)
        (C_a, C_b) ← Crossover(Copy(Parent_a), Copy(Parent_b))
        Q ← P ∪ {Mutate(C_a), Mutate(C_b)}
    end for

    P ← Q
until Best is the ideal solution or we have run out of time

return Best
```

Listing 1.1: Algorytm Genetyczny (GA) – Luke, S. *Essentials of Metaheuristics*, 2012, p. 35

Listing 1.1 przedstawia pseudokod algorytmu genetycznego. Wyraźnie widać poszczególne fazy algorytmu. W pierwszej kolejności wyłaniana jest populacja początkowa, następnie jest ona oceniania. Kolejnym krokiem jest selekcja osobników oraz operacje genetyczne. Następnie nowa populacja zastępuje starą. Korki powtarzane są aż do warunku stopu.

1.4. Reprezentacja instancji

Według Michalewicza i Fogela [10], w początkowej fazie rozwoju algorytmów ewolucyjnych skłaniano się do korzystania z binarnych reprezentacji problemu. Panował pogląd, iż niezależnie od problemu taka reprezentacja przyniesie benefity. Jednak przy problemie komiwojażera takie podejście nie daje żadnych realnych korzyści. Korzystanie z reprezentacji binarnej wymagało by używania operatorów, których zadaniem była by naprawa błędnych ścieżek.

Jak podaje El-Ghazali [12] w kontekście algorytmów genetycznych zakodowane rozwiązanie, nazywa się chromosomem. Poszczególne zmienne składające się na chromosom to geny, możliwe wartości genu to allele, a pozycja danego genu w chromosomie to locus. W kontekście problemu komiwojażera chromosom odpowiada trasie przejścia, gen to jedno miasto znajdujące się w chromosomie, allel to konkretne miasto, a locus to numer porządkowy miasta w kolejności odwiedzania.

Michalewicz i Fogel [10] podają iż w latach 80 XX wieku wyklarowały się trzy reprezentacje wektorowe stworzone dla problemu komiwojażera: lista sąsiedztwa, reprezentacja porządkowa oraz reprezentacja ścieżkowa. Ze względu na specyfikę tych reprezentacji każda z nich posiada własne operatory różnicowania. Poniżej zostały przedstawione wspomniane w literaturze sposoby prezentacji tras.

Lista Sąsiedztwa

Przedstawia ona trasę w postaci listy o długości *n* miast. Miasto *j* znajduje się na pozycji *i* jeżeli trasa przebiega z miasta *i* do miasta *j*. Autorzy jako zaletę tej metody podają możliwość korzystania z wzorców związanych z dobrymi rozwiązaniami. Pozwalały one ograniczyć zbiór przeszukiwań i szukać osobników posiadających obiecujące krawędzie.
Trasa: 4-3-1-2 przedstawiona będzie jako: (2 4 1 3),

Reprezentacja porządkowa

Przedstawia ona trasę w formie listy *n* miast, element listy na pozycji *i* jest reprezentowany przez liczbę z zakresu od *i* do *n-i-1*. Liczba ta oznacza indeks miasta z listy miast dostępnych. Lista miast dostępnych jest uporządkowanym zbiorem miast, które nie zostały jeszcze umieszczone na liście.
Trasa: 4-3-1-2 przedstawiona będzie jako: (4 3 1 1),

Reprezentacja ścieżkowa

Przedstawia ona trasę w postaci listy o długości *n* miast, miasta ułożone są w kolejności występowania.
Trasa: 4–3–1–2 przedstawiona będzie jako: (4 3 1 2),

Metody Alternatywne

Poza trzema głównymi opisanymi wyżej metodami, autorzy wyróżniają jeszcze inne podejścia [10]. Warto zaznaczyć, że poniższe reprezentacje charakteryzują się mniejszą czytelnością dla człowieka. Reprezentacja poprzez wektor liczb zmiennoprzecinkowych - kolejność elementów po sortowaniu określa trasę. Reprezentacja macierzowa - element m_{ij} macierzy zawiera 1 jeżeli miasto *i* występuje na trasie przed miastem *j*.

W kontekście niniejszej pracy reprezentacje takie jak poprzez wektor liczb zmiennoprzecinkowych oraz macierzowa mogą być mniej wydajne ze względu na zajętość pamięciową oraz złożoność obliczeń. Reprezentacja ścieżkowa ze względu na swoją prostotę zapewnia większą wydajność w porównaniu z innymi metodami reprezentacji tras.

1.5. Inicjalizacja Populacji

Jak wskazuje El-Ghazali [12] inicjalizacja populacji stanowi kluczowy element algorytmu genetycznego. Zapewnienie różnorodności pozwala zwiększyć efektywność i skuteczność algo-

rytmu, pomaga również zapobiec skupieniu populacji w przestrzeni o niższej jakości rozwiązań. Poniżej przedstawiono opisane przez niego metody.

Generacja losowa

Najpopularniejszy ze sposobów inicjalizacji populacji to generacja losowa. Autor podaje że może ona zostać przeprowadzona z wykorzystaniem liczb pseudolosowych lub sekwencji quasi-losowych. Sekwencje quasi-losowe poza niezależnością kolejnych liczb charakteryzują się również lepszym rozłożeniem liczb w przestrzeni przeszukiwań, co pozwala na zwiększoną różnorodność. Przy korzystaniu z generatorów liczb pseudolosowych warto zwrócić uwagę na właściwości generatorów, tak by zapewniały największą różnorodność.

Zróżnicowanie sekwencyjne

Kolejną z opisanych metod jest zróżnicowanie sekwencyjne. Jej celem jest jak najrównomierniejsze pokrycie przestrzeni rozwiązań. Jako przykład autor podaje proste sekwencyjne wykluczanie (Simple Sequential Inhibition – SSI). Generowane jest jedno rozwiązanie początkowe, następnie generowane są kolejne pseudolosowe rozwiązania, które muszą spełnić warunek minimalnej odległości od utworzonej już populacji. W kontekście problemu komiwojażera odległość rozwiązań można przedstawić jako ilość miast powtarzających się na danych pozycjach permutacji. Mniejsza liczba powtarzających się miast oznacza większą odległość. Metoda ta zapewnia dobrą dystrybucje populacji w przestrzeni rozwiązań, zwłaszcza dla wyższej wartości progu. Wadą jest natomiast wysoki koszt obliczeniowy.

Zróżnicowanie równoległe

Metoda ta również generuje nowe rozwiązania celem jak najrównomierniejszego pokrycia przestrzeni, natomiast każde rozwiązanie jest niezależne od poprzedniego i są one generowane są równolegle. Przytoczony jest przykład generowania populacji początkowej w taki sposób by każde miasto występowało na danej pozycji dokładnie raz. Permutacja początkowa jest generowana losowo, następnie kolejne permutacje są tworzone poprzez operatory mapowania w taki sposób by miasta nie powtarzały się na pozycjach. Autor zaznacza, że generowanie populacji początkowej można rozważać jako problem optymalizacyjny, który może być trudniejszy do rozwiązania niż oryginalny problem. Podobnie jak poprzednia, metoda ta charakteryzuje się dużym zużyciem zasobów.

Podejście hybrydowe polega na pseudolosowym generowaniu części populacji i dopełnieniu jej do docelowego rozmiaru. Dopełnienie odbywa się poprzez poszukiwanie rozwiązań najbardziej oddalonych od dotychczasowych rozwiązań.

Inicjalizacja heurystyczna

Ostania wymieniona przez El-Ghazaliego [12] metoda polega na heurystycznym generowaniu populacji początkowej. Zastosowany może być np. algorytm zachłanny. Ważne jest urozmaicenie uzyskiwanych rozwiązań tak by uniknąć minimów lokalnych i zapewnić różnorodność.

Metoda ta szybko znajduje obiecujące obszary przestrzeni rozwiązań, co może być przydatne w optymalizacji problemu komiwojażera dla dużych instancji. Połączenie inicjalizacji heurystycznej wraz z losową generacją rozwiązań zapewniło by urozmaiconą populację początkową z dużym potencjałem osiągnięcia dobrych wyników.

1.6. Funkcja celu

Funkcja celu definiuje cel który chcemy osiągnąć rozwiązując problem. Pozwala ona uporządkować wszystkie uzyskane rozwiązania przestrzeni przeszukiwań. El-Ghazali [12] zwraca uwagę na to że funkcja celu prowadzi nas przez przestrzeń rozwiązań w kierunku rozwiązań wartościowych. W przypadku problemu komiwojażera funkcja celu wynika z definicji problemu. Zdefiniowana jest ona przez całkowitą długość cyklu Hamiltona.

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \tag{1.1}$$

gdzie π oznacza permutację miast, n liczbę miast, a $d_{a,b}$ koszt przejścia z miasta a do miasta b (El-Ghazali 2009).

Warto zaznaczyć że w powyższym opisie została pominięta funkcja przystosowania. W przypadku problemu komiwojażera często pojawia się ona jako osobne pojęcie. Mówi ona o tym, jak dobre jest rozwiązanie w kontekście całej populacji, natomiast funkcja oceny jest ściśle związana z modelem problemu. Zwykle dla problemu komiwojażera jest ona odwrotnością funkcji celu. Dzięki temu wraz ze wzrostem jakości rozwiązania, wartość funkcji przystosowania będzie rosnąć.

$$Fitness(\pi) = \frac{1}{f(\pi)} \tag{1.2}$$

1.7. Strategia selekcji

Selekcja ma na celu wybór osobników z obecnej populacji. Opisane w literaturze metody selekcji mają zróżnicowany wpływ na dywersyfikację rozwiązań oraz tempo zawężania przestrzeni przeszukiwań [17, 10, 12].

Michalewicz i Fogel wskazują na podział metod selekcji na deterministyczne oraz stochastyczne. Zastosowanie selekcji deterministycznej dla danej populacji zawsze da ten sam wynik, *n* najgorszych osobników zostanie odrzucone. Zaletą takiego podejścia jest szybkość implementacji ale równocześnie szybko zatracana jest różnorodność populacji. Poza selekcją deterministyczną autorzy wskazali również trzy metody selekcji stochastycznej: selekcję proporcjonalną oraz dwie metody turniejowe.

El-Ghazali wyróżnił dwie metody przypisywania wartości przystosowania rodziców przy selekcji. W pierwszej z nich do rodzica przypisywana jest bezpośrednio wartość jego przystosowania. Osobnik o większym przystosowaniu ma większe szanse na bycie wybranym. Drugi ze sposobów to przypisywanie osobnikom rang zgodnie z ich wartościami przystosowania posortowanymi malejąco. Takie podejście zapobiega zdominowaniu populacji przez osobnika o znacząco wyższej wartości przystosowania. Opisał on selekcję metodą ruletki, stochastyczne próbkowanie uniwersalne, selekcję turniejową oraz rankingową. Podobnie jak El-Ghazali, Luke również zwrócił uwagę na selekcję metodą ruletki, stochastyczne próbkowanie uniwersalne oraz selekcję turniejową.

Selekcja ruletkowa

Jest to najpopularniejsza metoda selekcji. Każdemu osobnikowi w populacji przyporządkowujemy wartość proporcjonalną do jego przystosowania. Wartość przystosowania pojedynczego osobnika jest dzielona przez sumę przystosowań osobników w całej populacji. Im większą wartość przystosowania posiada osobnik, tym większy wycinek teoretycznego koła ruletki zajmuje. El-Ghazali wskazuje, że przy osobnikach znacząco odstających od reszty populacji na początku poszukiwań, może dojść do nieodwracalnej utraty obiecujących rozwiązań.

W literaturze selekcja ruletkowa znana jest również pod nazwą selekcji proporcjonalnej. Definicja selekcji proporcjonalnej przedstawiona przez Michalewicza i Fogela pod pewnym względami odbiega od tej przedstawionej przez Luke'a. Przedstawiają oni prawdopodobieństwo wybrania danego osobnika jako proporcjonalne do stosunku jego przystosowania do średniego przystosowania całej populacji.

Wadą selekcji ruletkowej jest etap obliczania prawdopodobieństw wyboru dla poszczególnych osobników, który trzeba wykonać dla każdej generacji. Problem pojawia się również przy minimalnych różnicach przystosowania osobników, wtedy lepsze rozwiązania nie są w żaden sposób faworyzowane.

Stochastyczne próbkowanie uniwersalne

Zgodnie z definicją El-Ghazaliego metoda ta działa podobnie do selekcji ruletkowej, lecz pozwala zachować większą różnorodność, która zatraca się przy znaczących różnicach wartości przystosowania. Prawdopodobieństwo wyboru poszczególnych osobników oblicza się w ten sam sposób. Różnica pojawia się w sposobie ich wyboru. Na teoretycznym kole ruletki ustawiane jest n wskaźników, rozmieszczone są one w tych samych odległościach od siebie. Podczas jednego losowania wybierane jest n osobników. Takie podejście zwiększa szanse osobników o mniejszym przystosowaniu, nie wykluczając tych o większym.

Zdaniem Luke'a stochastyczne próbkowanie uniwersalne przynosi dwie główne korzyści. Złożoność obliczeniowa dla wyboru n osobników wynosi $O(n)$, a nie $O(n \log n)$ jak w przypadku selekcji ruletkowej. Takie podejście gwarantuje również wybranie osobników z przystosowaniem większym od sumy przystosowań podzielonej przez liczbę wybieranych osobników. W przypadku selekcji ruletkowej może się zdarzyć że najlepsze osobniki zostaną pominięte przy każdym z n losowań. Autor zwraca również uwagę na zachowanie metody pod koniec działania algorytmu, gdy wartości przystosowań wszystkich osobników są wysokie. Podobnie jak przy selekcji ruletkowej, małe różnice w prawdopodobieństwie wyboru poszczególnych osobników mogą doprowadzić do zatracenia tych najlepszych.

Selekcja turniejowa

Selekcja turniejowa polega na wyborze t osobników z populacji, a następnie wyborze osobnika najlepiej przystosowanego z wylosowanej puli. Jak zauważa Luke metoda ta nie jest wrażliwa na wartość funkcji przystosowania. Zwraca również uwagę na jej prostotę oraz możliwość równoległego wykonywania. Manipulowanie wielkością turnieju czyli wartością t pozwala osiągnąć zróżnicowane wyniki. Niski rozmiar turnieju zwiększa losowość, a większy zwiększa szanse na wybór najlepszego osobnika. Autor jako optymalny rozmiar turnieju podaje liczbę t=2. Zwraca również uwagę na możliwość wyboru t z akresu liczb rzeczywistych od 1.0 do 2.0. Wtedy z prawdopodobieństwem t-1 przeprowadzana będzie selekcja turniejowa o rozmiarze t=2 lub losowe wybieranie osobnika z populacji.

Michalewicz i Fogel zwracają uwagę na alternatywną wersję selekcji turniejowej. Polega ona na wykonywaniu serii porównań osobników w ramach jednego podzbioru populacji. Każdemu osobnikowi przypisywana jest liczba punktów. Osobnik co najmniej tak samo dobry jak jego przeciwnik otrzymuje punkt. Powoduje to, że nawet osobniki o niskiej wartości przystosowania mogą zdobywać punkty gdy trafią na słabszych przeciwników. Maksymalna liczba punktów możliwa do zdobycia przez jednego osobnika jest równa rozmiarowi podzbioru populacji. Po zakończeniu turnieju dokonywana jest deterministyczna selekcja osobników na podstawie zdobytej liczby punktów.

Selekcja rankingowa

Oprócz wyżej wymienionych metod El-Ghazali zwrócił uwagę na selekcje rankingową, która została pominięta zarówno przez Luke'a jak i Michalewicza i Fogela. Metoda ta zamiast skupiać się na wartościach przystosowania osobników, koncentruje się na ich randze na tle innych. Nie liczy się to o ile lepszy jest osobnik, tylko to czy jest lepszy, osobnik z wyższym przystosowaniem otrzymuje wyższą rangę. Autor przedstawia następujący wzór na liniowe skalowanie rang:

$$P(i) = \frac{2-s}{N} + \frac{2\,r(i)\,(s-1)}{N(N-1)} \tag{1.3}$$

s – nacisk selekcyjny , przy czym $1.0 < s \leq 2.0$,
N – rozmiar populacji,
$r(i)$ – ranga przypisana osobnikowi i.

Im więcej wynosi nacisk selekcyjny, tym bardziej wzrasta waga osobników o wyższej randze.

1.8. Strategia reprodukcji

Operatory różnicowania, które stosuje się w fazie reprodukcji podzielone są na dwie kategorie: operatory krzyżowania oraz operatory mutacji. Krzyżowanie wymaga dwóch osobników, na których wykonujemy operacje, natomiast mutacja przeprowadzana jest na pojedynczej jednostce. Operatory te stosujemy na osobnikach wybranych z populacji poprzez zastosowanie jednej z metod opisanych w poprzednim podrozdziale. Dobrze dobrane operatory mogą zapewnić wzrost jakości oraz pomóc uniknąć skupienia w minimach lokalnych przestrzeni rozwiązań.

Krzyżowanie

W swojej książce Michalewicz i Fogel [10] podzielili operatory krzyżowania zgodnie z reprezentacjami problemu komiwojażera, których one dotyczą. El-Ghazali [12] przedstawił operatory krzyżowania dla permutacji, nie skupiając się wyłącznie na problemie komiwojażera. Z kolei Luke [17] przedstawił krzyżowanie na reprezentacjach binarnych, zgodnie z oryginalna ideą algorytmów ewolucyjnych oraz algorytmu genetycznego. Ze względu na kontekst niniejszej pracy uwaga zostanie poświęcona głównie metodom przedstawionym przez Michalewicza i Fogela oraz El-Ghazaliego.

Ważnym elementem wpływającym na efekty krzyżowania jest współczynnik krzyżowania. Jest to prawdopodobieństwo z jakim dojdzie do krzyżowania między wylosowaną parą osobników. El-Ghazali wskazuje, że wartość współczynnika krzyżowania ma ścisły związek z rozmiarem populacji, współczynnikiem mutacji oraz metodą selekcji. Jako typowy współczynnik krzyżowania podaje on wartość od 0.45 do 0.95.

Krzyżowanie dla listy sąsiedztwa

Pierwszy zbiór metod krzyżowania opisany przez Michalewicza i Fogela [10] dotyczy reprezentacji w postaci listy sąsiedztwa. Dla tej reprezentacji wyróżnione zostały trzy operatory krzyżowania: krzyżowanie przez naprzemienne wybieranie krawędzi, krzyżowanie przez wymianę podtras oraz krzyżowanie heurystyczne.
Przypomnienie reprezentacji trasy w postaci listy sąsiedztwa:
Trasa: 4-3-1-2 Lista sąsiedztwa: (2 4 1 3)

Krzyżowanie przez naprzemienne wybieranie krawędzi polega na losowym wybraniu krawędzi z jednego rodzica i odpowiedniej krawędzi w drugim rodzicu. Krawędzie z rodziców wybie-

rane są naprzemiennie. Jeżeli krawędź któregoś z rodziców tworzy cykl, to losowo wybierana jest inna z krawędzi tego rodzica.

Krzyżowanie przez wymianę podtras wybiera naprzemiennie podtrasy o losowej długości z obu rodziców. Podobnie jak w poprzednim operatorze, jeżeli któraś krawędź przedwcześnie wprowadza cykl to zastępowana jest ona losową krawędzią z niewybranych.

Krzyżowanie heurystyczne wybiera losowo miasto jako punkt początkowy potomka. Kolejna krawędź wybierana jest poprzez porównanie krawędzi wychodzących z tego miasta w obu rodzicach, wybierana jest krótsza. Ponownie jeżeli wybrana krawędź tworzy cykl, to wybierana jest losowa z niewybranych jeszcze krawędzi. Zmodyfikowana wersja zamiast losowej krawędzi w wypadku cyklu wybierała dłuższą, a jeżeli ona również tworzyła cykl to wybierana była najkrótsza z losowo q wybranych krawędzi.

Michalewicz i Fogel zaznaczają że korzystanie z powyższych operatorów różnicowania dla listy sąsiedztwa nie przynosiło zadowalających rezultatów. Krzyżowanie heurystyczne pozwalało uzyskać wyniki lepsze od pozostałych krzyżowań lecz w dalszym ciągu nie były one na zadowalającym poziomie.

Krzyżowanie dla reprezentacji porządkowej

Jedyna metoda wymieniona przez autorów przy reprezentacji porządkowej to klasyczna metoda cięcia i łączenia.
Przypomnienie reprezentacji trasy w postaci reprezentacji porządkowej:
Trasa: 4-3-1-2 Reprezentacja porządkowa: (4 3 1 1)

Metoda ta polega na wykonaniu cięcia na wybranej pozycji u obu rodziców i zamianie fragmentów za linią cięcia u obu rodziców. Uzyskamy w ten sposób dwóch potomków. Zdaniem autorów prostota tego rozwiązania jest jedyną zaletą reprezentacji porządkowej. Zwracają również uwagę na wadę takiego podejścia, trasy na lewo od punktu cięcia nie są zmienione u rodziców, a trasy na prawo są ułożone w losowy sposób.

Krzyżowanie dla reprezentacji scieżkowej

Dla reprezentacji ścieżkowej Michalewicz i Fogel wymienili trzy najbardziej znane operatory krzyżowania: z częściowym odwzorowaniem, z zachowaniem porządku oraz cykliczne. El-Ghazali opisał krzyżowanie z zachowaniem porządku, z częściowym odwzorowaniem oraz krzyżowanie dwupunktowe.
Przypomnienie reprezentacji trasy w postaci reprezentacji ścieżkowej:
Trasa: 4-3-1-2 Reprezentacja porządkowa: (4 3 1 2)

Krzyżowanie z częściowym odwzorowaniem (PMX) tworzy potomstwo poprzez wstawienie części trasy jednego z rodziców oraz zachowanie pozycji tak wielu miast z drugiego rodzica jak to możliwe. Losowane są dwa punkty cięcia, które ustalają podciąg miast do wstawienie do potomka. Miasta z obu podciągów rodziców tworzą odwzorowania, które będą wykorzystywane do rozwiązywania konfliktów. Następnie uzyskany podciąg jest wstawiany do potomka i jest on uzupełniany miastami z drugiego rodzica nie tworzącymi konfliktów. Jeżeli któreś z miast nie może być wstawione do potomka ponieważ występuje w podciągu uzyskanym z rodzica, wstawiane jest inne miasto zgodnie z odwzorowaniem. Zarówno Michalewicz i Fogel oraz El-Ghazali opisują proces w ten sam sposób. Przykład 1 przedstawia działanie operatora PMX.

Przykład 1. Przykład działania operatora PMX. Źródło: opracowanie własne.

$$Rodzic1 : (3\ 2\ 5\ 4\ 1), \quad Rodzic2 : (1\ 4\ 3\ 2\ 5)$$

Fragmenty do wymiany:

$$R1_{[2-4]} = (2\ 5\ 4), \quad R2_{[2-4]} = (4\ 3\ 2)$$

Odwzorowanie między genami z wymienionych fragmentów:

$$2 \leftrightarrow 4, \quad 5 \leftrightarrow 3, \quad 4 \leftrightarrow 2$$

Uzyskane potomstwo:

$$Potomek1 = (5\ 4\ 3\ 2\ 1), \quad Potomek2 = (1\ 2\ 5\ 4\ 3)$$

Krzyżowanie z zachowaniem porządku (OX) podobnie, jak poprzedni operator wybiera podciągi z rodziców i wstawia je do potomstwa. Różnica polega na uzupełnianiu pozostałych wartości. Po wstawieniu podciągu do potomka ustawiamy się na pierwszej pozycji za punktem cięcia. Począwszy od tego miejsca wstawiamy miasta w kolejności występującej w drugim z rodziców za drugim miejscem cięcia, pomijając wartości już występujące w potomku. Gdy dojedziemy do końca potomka, kopiowanie kontynuujemy od jego początku, aż dojdziemy do pierwszego punktu cięcia. Ponownie brak rozbieżności między definicjami w obu publikacjach. Przykład 2 przedstawia działanie operatora OX.

Przykład 2. Przykład działania operatora OX. Źródło: opracowanie własne.

$$Rodzic1 : (3\ 2\ 5\ 4\ 1), \quad Rodzic2 : (1\ 4\ 3\ 2\ 5)$$

Fragmenty do wymiany:

$$R1_{[2-4]} = (2\ 5\ 4), \quad R2_{[2-4]} = (4\ 3\ 2)$$

Uzyskane potomstwo:

$$Potomek1 = (3\ 2\ 5\ 4\ 1), \quad Potomek2 = (5\ 4\ 3\ 2\ 1)$$

Kolejny z operatorów dla reprezentacji scieżkowej wspomniany przez Michalewicza i Fogela to krzyżowanie cykliczne (CX). W tym operatorze pozycja każdego miasta w potomku pochodzi od jednego z rodziców. Przykład 3 przedstawia operator CX.

Przykład 3. Przykład działania operatora CX . Źródło: opracowanie własne.

$$Rodzic1 = (1\ 2\ 3\ 4\ 5), \quad Rodzic2 = (3\ 5\ 1\ 2\ 4)$$

Wybieramy pierwszą pozycję z rodzica 1, jest to miasto 1. Umieszczamy je na 1 pozycji. Kolejnym rozważanym miastem jest miasto 3, które jest na 1 pozycji rodzica 2. Umieszczamy je na pozycji 3, ponieważ tam znajduje się w rodzicu 1. W rodzicu 2 na pozycji 3 znajduje się miasto 1. Jako, że miasto 1 tworzy cykl, ponieważ znajduje się w potomku, pozostałe miasta są uzupełniane z drugiego rodzica.

Postępując w analogiczny sposób dla potomka 2 uzyskujemy:

$$Potomek1 = (1\ 5\ 3\ 2\ 4), \quad Potomek2 = (3\ 2\ 1\ 4\ 5)$$

Krzyżowanie dwupunktowe opisane przez El-Ghazaliego ponownie polega na wybraniu dwóch punktów cięcia. Miasta znajdujące się po zewnętrznych stronach punktów są kopiowane do potomka. Wolny fragment w środku jest uzupełniany miastami z drugiego rodzica, w kolejności ich występowania i z pominięciem miast już skopiowanych.

Michalewicz i Fogel przytaczają dwie zmodyfikowane wersje krzyżowania z zachowaniem porządku. W pierwszej z nich losowane są pozycje z pierwszego rodzica, kolejność miast znajdujących się na tych pozycjach zostanie narzucona miastom w drugim rodzicu. Przedstawiona jest ona na przykładzie 4.

Przykład 4. Przykład działania zmodyfikowanego operatora OX . Źródło: opracowanie własne.

$$Rodzic1 = (1\ 2\ 3\ 4\ 5), \quad Rodzic2 = (3\ 4\ 5\ 1\ 2)$$

Załóżmy, że zostały wylosowane pozycje 2 i 4. W rodzicu 1 na tych pozycjach znajdują się miasta 2 i 4. W rodzicu 2 te miasta znajdują się odpowiednio na pozycjach 5 i 2. W potomku należy przestawić miasta na tych pozycjach tak by występowały w porządku z rodzica 1. Postępując w ten sposób dla dwóch rodziców otrzymamy potomków:

$$Potomek1 = (4\ 2\ 3\ 1\ 5), \quad Potomek2 = (3\ 2\ 5\ 1\ 4)$$

Druga zmodyfikowana metoda zamiast wybierać podciąg miast do skopiowania, wybiera losowo kilka miast. Wylosowane miasta są wstawiane do potomka, proces uzupełniania przebiega w ten sam sposób co w oryginalnym krzyżowaniu z zachowaniem porządku.

Michalewicz i Fogel opisali również operatory, które kładą nacisk na wybór krawędzi, w tym opracowane przez Grefenstettena operatory heurystyczne oraz operator z rekombinacją krawędzi (ER). Wymagają one reprezentacji ścieżkowej wraz z listą z krawędzi, by przekazać wszystkie istotne podczas krzyżowania informacje. Autorzy przedstawiają również operatory krzyżowania dla reprezentacji macierzowej. Powyższe metody charakteryzują się wyższą złożonością pamięciową i obliczeniową. Z tego powodu ich dokładniejszy opis został pominięty.

Mutacja

Operatory mutacji wprowadzają zaburzenia w obrębie jednego osobnika. El-Ghazali [12] dla permutacji przedstawia trzy główne operatory mutacji: zamianę, inwersję oraz wstawianie. Wskazuje, że dobry operator mutacji nie powinien ograniczać przestrzeni poszukiwań, wprowadzać minimalne zmiany oraz tworzyć poprawne rozwiązania. Prawdopodobieństwo wykonania operacji mutacji, podobnie jak krzyżowania, określa się współczynnikiem. W klasycznych przypadkach dla reprezentacji binarnych określa on prawdopodobieństwo zamiany pojedynczego bitu. Jak podaje autor, zwyczajowo jest on niski i wynosi od 0.001 do 0.01. W przypadku permutacji i problemu komiwojażera, ze względu na charakterystykę mutacji nie da się przeprowadzać mutacji dla pojedynczych genów. Wprowadzałoby to ryzyko tworzenia nielegalnych rozwiązań. Z tego powodu współczynnik mutacji podaje prawdopodobieństwo wykonania mutacji dla całego osobnika. Zwykle jest on z większego przedziału np. od 0.01 do 0.1. Przedstawione operatory działają w przypadku reprezentacji ścieżkowej problemu komiwojażera. W przypadku listy sąsiedztwa oraz reprezentacji porządkowej mogą powstawać nielegalne rozwiązania.

Mutacja przez zamianę polega na zamianie miast na dwóch wylosowanych pozycjach. Wprowadza to minimalne zaburzenie i pozwala zachować większość cech osobnika.

Mutacja przez inwersję odwraca kolejność miast między dwoma wylosowanymi pozycjami. Zmiany wprowadzane przez mutację są duże i w przypadku skrajnym, gdy wylosowane punkty znajdować się będą na początku i końcu osobnika, powstanie zupełnie nowa permutacja miast.

Mutacja przez wstawianie podobnie jak mutacja przez zamianę jest mało inwazyjna. Losowane jest miasto oraz nowa pozycja, na którą zostanie wstawione. Wprowadza ona jeszcze mniej zmian niż zamiana.

1.9. Strategia zastępowania

Strategia zastępowania definiuje w jaki sposób tworzona będzie nowa populacja. Luke [17] wyróżnił dwie strategie zastępowania dla algorytmu genetycznego: elitaryzm oraz stan ustalony (Steady-State). W elitaryzmie zachowujemy *n* najlepszych osobników z pokolenia rodziców i przenosimy ich do nowej populacji. Zachowanie najlepszych osobników z poprzedniej populacji pozwala przekazać ich geny kolejnym pokoleniom, zwiększając tym eksploatację w algorytmie.

Przy podejściu Steady-State wykorzystywana jest tylko jedna populacja. W każdej iteracji operacje genetyczne dokonywane są tylko dla jednej pary rodziców, stworzone dzieci zastępują dwa osobniki z populacji. Osobniki do zastąpienia najczęściej wybierane są losowo, autor wspomina jednak, że wybór może odbyć się np. przez selekcję turniejową gdzie wybierany jest najsłabszy osobnik. Doprowadzić może to jednak do przedwczesnej zbieżności. Luke przytacza dwie główne zalety Stead-State: mniejsze zużycie pamięci dzięki stosowaniu tylko jednej populacji oraz zwiększenie eksploatacji rozwiązań dzięki rzadszym wymianom osobników.

El-Ghazali [12] zwrócił uwagę na całkowitą zamianę pokoleń oraz, podobnie jak Luke, na Steady-State. Wskazał jednak, że te dwa skrajne podejścia dają wiele możliwości ich kombinacji. Przykładem jest zachowywanie części poprzedniej populacji, czyli przytoczony wcześniej elitaryzm.

1.10. Kryterium stopu

Kryteria stopu algorytmu genetycznego mają kluczowy wpływ na uzyskane wyniki. El-Ghazali [12] podzielił kryteria stopu na dwie kategorie. Przy statycznych kryteriach, kryterium jest ustalane przed rozpoczęciem algorytmu. Może to być limit zużycia zasobów procesora, liczba pokoleń lub liczba wyliczeń funkcji oceny. Przy kryteriach adaptacyjnych przed rozpoczęciem algorytmu zakończenie nie jest jawnie znane. Warunkiem może być liczba pokoleń bez poprawy wyniku, znalezienie wyniku optymalnego lub jego przybliżenia.

Możliwe jest też wykorzystanie warunków odwołujących się bezpośrednio do różnorodności populacji. Gdy populacja stanie się zbyt mało różnorodna i spadnie poniżej wyznaczonego progu, algorytm zostanie przerwany, ze względu na brak potencjału uzyskania dobrych wyników. Podejście to wymaga operatorów mierzących zróżnicowanie osobników. W przypadku problemu komiwojażera może to być liczba unikalnych tras lub zróżnicowanie ułożenia miast w trasach.

1.11. Podsumowanie

Na podstawie przedstawionego przeglądu literatury można stwierdzić, że algorytm genetyczny jest bardzo złożonym zagadnieniem. Ze względu na wiele podejść naukowców dostępnych jest wiele wariantów poszczególnych elementów tego algorytmu, które pozwalają na manipulowanie jego działaniem. Odpowiednie dobranie parametrów pozwala na balansowanie między eksploatacją i eksploracją przestrzeni rozwiązań danej instancji problemu komiwojażera. Dzięki temu możliwe jest osiąganie zadowalających wyników nawet dla instancji problemu, gdzie optymalne rozwiązania znajdują się w początkowo odległym miejscu przestrzeni rozwiązań.

W niniejszej pracy zdecydowano się na zastosowanie ścieżkowej reprezentacji trasy. Przedstawienie trasy jako permutacji miast pozwala na łatwe generowanie poprawnych tras. Nie wprowadza ona potrzeby stosowania dodatkowych operatorów naprawiających trasy przy krzyżowaniu lub mutacji. W porównaniu do pozostałych reprezentacji, dzięki swojej prostocie reprezentacja ścieżkowa jest optymalna pod względem zajętości pamięciowej.

Inicjalizacja populacji odbywała się poprzez heurystyczne podejście uzupełnione o losowe generowanie tras. Jest to rozwiązanie hybrydowe. Zastosowany został algorytm zachłanny, który w prosty sposób pozwoli osiągnąć rozwiązania o dobrych cechach. Algorytm Najbliższego Sąsiada został zastosowany dla każdego miasta instancji. Pozwoli on na uzyskanie populacji zawierającej osobniki o obiecujących cechach, a dzięki losowemu dopełnieniu populacji przestrzeń przeszukiwań będzie równo pokryta. Przewagą takiego podejścia nad zróżnicowaniem równoległym oraz sekwencyjnym jest niski koszt obliczeniowy. Dla porównania zastosowano również inicjalizację całkowicie losową.

Do selekcji osobników zostały wybrane metody: turniejowa oraz ruletki. Reprezentują one dwa odmienne podejścia. Metoda turniejowa skupia się na tym, czy danym osobnik jest lepszy od innych, nie skupia się na jego wartości funkcji przystosowania. Metoda ruletki natomiast bezpośrednio wykorzystuje wartość funkcji przystosowania, daje to dużą przewagę osobnikom o dużej wartości funkcji. W sytuacjach gdy całe pokolenie będzie mieć podobną wartość funkcji przystosowania, przewaga lepszych osobników będzie znikoma.

Krzyżowanie zostało zaimplementowane poprzez krzyżowanie z zachowaniem porządku (OX) oraz krzyżowanie dwupunktowe. Pierwsza z tych metod pozwala na zachowanie względnego porządku miast w potomku. Druga metoda dzięki swojej prostocie jest szybka obliczeniowo. Operatorami mutacji, które zastosowano są dwie metody o odmiennym podejściu do wprowadzania zaburzeń. Mutacja przez zamianę oraz mutacja przez inwersję. Mutacja przez zamianę zapewnia małą zmianę w osobniku, która nie wprowadza dużych zaburzeń. Z kolei mutacja przez inwersję wprowadza większe skoki w przestrzeni rozwiązań, w skrajnym przypadku odwrócona zostanie kolejność całej trasy.

Nowe pokolenie jest tworzone poprzez zachowanie elity ze starego i dopełnienie nowymi osobnikami. Pozwala to zachować najlepsze cechy z poprzedniego pokolenia równocześnie nie ograniczając przeszukiwań do tylko najlepszych osobników. Ze względu na cel pracy, jakim jest porównanie wydajności, algorytm będzie ograniczony liczbą generacji populacji.

Rozdział 2

Opis implementacji algorytmu w języku Julia i języku Java

2.1. Wprowadzenie

Celem niniejszego rozdziału jest przedstawienie implementacji algorytmu genetycznego dla problemu komiwojażera w językach Julia i Java. Tworząc obie implementacje skupiono się na identyczności obu implementacji. Dla zapewnienia powtarzalności wyników oraz rzetelności badań zaimplementowany został generator liczb losowych. Umożliwi on uzyskanie identycznych populacji początkowych oraz sekwencji przy operacjach genetycznych.

W pierwszej kolejności w rozdziale skupiono się na porównaniu Julii i Javy pod względem charakterystyki obu języków oraz ich możliwości. Kolejnym elementem rozdziału jest opis zaimplementowanego zarówno w Julii jaki i Javie generatora liczb losowych. W dalszej kolejności opisane zostały implementacje algorytmu genetycznego wraz z fragmentami kodu oraz wskazane zostały główne różnice wynikające z charakterystyki obu języków. Przedstawione zostały kluczowe elementy implementacji, stanowiące fundament algorytmu genetycznego.

Przedstawione implementacje algorytmu genetycznego stanowiły podstawę do badań opisanych w kolejnym rozdziale, które to następnie poddane zostały testom statystycznym oraz dokładnej analizie.

2.2. Porównanie Julii oraz Javy

Julia i Java to języki programowania znacząco różniące się swoim przeznaczeniem oraz możliwościami. Java dzięki swojej popularności i dłuższej od Julii obecności na rynku, posiada znacznie lepiej rozwinięte środowisko, większą liczbę dostępnych narzędzi i bibliotek, a także rozbudowaną społeczność angażującą się w jej rozwój. Julia jest stosunkowo młodym językiem programowania, który jest w dalszym ciągu dynamicznie rozwijany, co przekłada się na coraz większą liczbę dostępnych narzędzi oraz bibliotek.

Java to język obiektowy szeroko wykorzystywany do tworzenia aplikacji na wszelkiego rodzaju platformach. Jest to możliwe dzięki oparciu o zasadę "Write Once, Run Anywhere", zgodnie z którą raz napisany program powinien działać na wielu platformach. Składnia Javy przypomina C++, jest silnie i statycznie typowana, typy danych nie mogą być zmieniane w trakcie działania kodu. Podstawą struktury kodu w Javie są klasy, umożliwiają one dziedziczenie, polimorfizm i wykorzystywanie interfejsów.

Julia jest językiem wykorzystywanym głównie do analizy danych i obliczeń matematycznych. W Julii typowanie jest dynamiczne. Oznacza to że typy zmiennych określane są w trakcie

wykonywania kodu. Ułatwia to pisanie kodu, lecz może prowadzić do błędów które widoczne będą podczas działania programu. Składnia Julii jest zbliżona do Pythona i jest prostsza niż w przypadku Javy. Julia umożliwia korzystania z podejścia proceduralnego jak i obiektowego. Warto zaznaczyć, że nie jest to typowa obiektowość jak w przypadku Javy. Tworzone są struktury umożliwiające przechowywanie danych które umożliwiają korzystanie z typów abstrakcyjnych. Funkcje nie są bezpośrednio związane ze strukturami, lecz można je powiązać z danym typem struktury wykorzystując Multiple Dispatch wybierający wersję funkcji na podstawie dostarczonych argumentów. Julia w odróżnieniu od Javy nie wspiera dziedziczenia. Istotną różnicą jest sposób indeksowania w obu językach, w Julii rozpoczyna się od 1, a w Javie od 0. W przypadku Julii do osiągnięcia wydajności kluczowe jest pisanie kodu w konkretnym stylu i przestrzegając zasad. Kluczowe jest dbanie o odpowiednie typy zmiennych i korzystanie z mark takich jak: "@simd"i "@inbounds", które zwiększają wydajność pętli. Na oficjalnej stronie Julii znajdują się wskazówki dotyczące pisania szybkiego kodu [5].

Kompilacja

Istnieją trzy metody kompilacji w Javie [19]. Pierwsza z nich to kompilacja kodu źródłowego do kodu bajtowego, który jest interpretowany przez JVM (Java Virtual Machine). Druga metoda to dwustopniowa kompilacja. Kod w pierwszej kolejności kompilowany jest do kodu bajtowego, a następnie kompilowany do kodu maszynowego z użyciem kompilatora just-in-time (JIT). Mechanizm ten wykorzystywany jest dla najczęściej wykonywanych fragmentów. Skraca to czas uruchamiania kodu. Obie z tych metod korzystają z kodu bajtowego, który umożliwia uruchamianie go na wielu platformach bez potrzeby ponownej kompilacji. Trzecia metoda to bezpośrednia kompilacja do kodu maszynowego danej platformy. Takie podejście oznacza utracenie przenośności kodu.

Kompilacja Julii odbywa się w inny sposób niż w przypadku Javy. W Julii kod jest kompilowany do kodu maszynowego podczas pierwszego wywołania danej funkcji dla konkretnego zestawu typów argumentów, powoduje to wolniejsze pierwsze wykonanie kodu. Realizowane to jest za pomocą kompilacji just-in-time (JIT). Nie ma elementu pośredniczącego, jakim w przypadku Javy jest kod bajtowy. Kod zawsze kompilowany jest pod platformę, na której jest uruchomiany. Kompilacja składa się z trzech faz [6]. Kod parsowany jest do drzewa abstrakcyjnej składni (Julia AST), które następnie parsowane są do reprezentacji pośredniej, która używana jest do wprowadzania optymalizacji. Następnie kod jest tłumaczony na reprezentacje pośrednią LLVM z której generowany jest kod maszynowy.

Zarządzanie pamięcią

W Javie oraz Julii zarządzanie pamięcią odbywa się za pomocą systemowych rozwiązań. Oba języki wykorzystują mechanizm odśmiecania pamięci (Garbage Collector). W przypadku Javy JVM oferuje wiele algorytmów odśmiecania pamięci, wybór odpowiedniego zapewnia optymalne wyniki. W artykule Zhao i Blackburna [23] wymieniono następujące Garbage Collectory: Lang and Dupont's Collector, CRE, SIM, G1, Shenandoah, C4 i ZGC. Od 2004 roku domyślnym rozwiązaniem dla JVM jest G1. Dwa główne etapy odśmiecania pamięci dla to śledzenie obiektów oraz ewakuacja. Najpierw obiekty używane są znajdowane, następnie przenoszone do nowej lokalizacji (w przypadku Lang and Dupont's Collector część obiektów pozostaje na miejscu). Algorytmy te poza Shenandoah, C4 i ZGC, wykorzystują metodę Stop-the-world, na co najmniej jednym z tych etapów. Oznacza to, że cały program jest zatrzymywany, aż etap odśmiecania pamięci się zakończy. Wadą dużej ilości dostępnych metod jest wymóg posiadania wiedzy, by ocenić jaki algorytm da najlepsze wyniki w danej sytuacji.

Julia wykorzystuje algorytm mark-and-sweep [11]. Znajduje on obiekty które nie są wykorzystywane i uwalania pamięć zajmowaną przez nie. Wadą odśmiecania pamięci w Julii jest działanie na zasadzie Stop-the-world, podobnie jak w Javie. W trakcie odśmiecania pamięci cały program jest zatrzymywany i algorytm przeszukuje pamięć w celu znalezienia obiektów do usunięcia. Sama faza usuwania może odbywać się częściowo współbieżnie z głównym programem i działać w tle. Obiekty umieszczone w pamięci nie zmieniają swojego miejsca. Sprawia to, że nie można określić wieku obiektu na podstawie miejsca w pamięci, w którym jest umieszczony. Do określania wieku wykorzystywane są dwa bity umieszczane w nagłówku obiektu. Takie podejście zapewnia większe bezpieczeństwo i spójność pamięci.

2.3. Generator liczb losowych

By zapewnić powtarzalność sekwencji generowanych liczb dla obu języków, zaimplementowany został generator liczb losowych Xoshiro256++. Jest to domyślny generator liczb w Julii, dostępny również w Javie od wersji 17. Jednak ze względu na odmienne inicjowanie generatora dla tego samego ziarna uzyskiwane są inne wyniki. Z tego powodu, zarówno w Julii jak i Javie na potrzeby badań zaimplementowany został generator Xoshiro256++ wraz z inicjowaniem stanu za pomocą generatora SplitMix64.

SplitMix64

SplitMix64 to szybki generator liczb pseudolosowych [15]. Do wygenerowania liczby 64 bitowej wykonywanych jest 9 operacji arytmetyczno-logicznych na liczbach 64 bitowych. Są to 3 operacje XOR, 3 przesunięcia bitowe, 2 mnożenia oraz dodawanie. Ważną cechą SplitMix64 jest możliwość podziału jednego generatora na dwa generatory, które mogą działać równolegle. Nie wymagają one synchronizacji ani blokowania wątków. Ze względu na charakter użycia generatora podział nie został zaimplementowany.

```
public class SplitMix64 {
    private long x;

    public SplitMix64(long seed) {
        this.x = seed;
    }

    public long next() {
        long z = (x += 0x9E3779B97F4A7C15L);
        z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
        z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
        return z ^ (z >>> 31);
    }
}
```

Listing 2.1: SplitMix64 zaimplementowany w Javie Źródło: opracowanie własne na podstawie [15]

Listing 2.1 przedstawia implementacje klasy SplitMix64. Generowanie nowej liczby rozpoczyna się przez uzyskanie nowego ziarna poprzez zwiększenie starego o stałą złotego podziału. Nowe ziarno nadpisuje obecne. Następnie na uzyskanej wartości Z jest dokonywane logiczne przesunięcie w prawo o 30 pozycji i operacja XOR na wartości przesuniętej oraz oryginalnej. Całość jest mnożona przez kolejną stałą. Uzyskana wartość nadpisuje dotychczasową liczbę Z. Ten sam proces powtarzany jest ponownie. Różnica jest w liczbie bitów o jaką dokonywane jest przesunięcie oraz w stałej. Zwracaną liczbą jest wynik operacji XOR na zmiennej Z i jej wartości po przesunięciu logicznym o 31 pozycji w prawo.

```
mutable struct SplitMix64
    x::Int64
end

const C1 = reinterpret(Int64, UInt64(0x9E3779B97F4A7C15))
const C2 = reinterpret(Int64, UInt64(0xBF58476D1CE4E5B9))
const C3 = reinterpret(Int64, UInt64(0x94D049BB133111EB))

function next(sm::SplitMix64)
    sm.x += C1
    z = sm.x
    z = (z ⊻ (z >>> 30)) * C2
    z = (z ⊻ (z >>> 27)) * C3
    return z ⊻ (z >>> 31)
end
```

Listing 2.2: SplitMix64 zaimplementowany w Julii Źródło: opracowanie własne na podstawie [15]

Listing 2.2 przedstawia implementacje struktury SplitMix64 w Julii. Generowanie liczb odbywa się identycznie jak w przypadku Javy. Jedyną różnicą, która występuje w przypadku Julii jest potrzeba reinterpretacji stałych. W Julii liczby szesnastkowe domyślnie interpretowane są jako bez znaku UInt64, przez co podane wartości stałych nie mieszczą się w zakresie dla Int64. Dzięki reinterpretacji zmieni się wartość liczby, ale reprezentacja bitowa pozostania taka sama.

Xoshiro256++

Jest to generator oferujący dobrą szybkość losowania oraz 256 bitową przestrzeń stanów, składającą się z czterech słów[8]. Nazwa Xoshiro256 wywodzi się od operacji XOR, SHIFT i ROTATE, które są wykonywane by generować liczby. ++ w nazwie odnosi się do scramblera ++, który wykonuje operacje: dodaj, obróć, dodaj , na dwóch słowach stanów. Stan początkowy generatora inicjowany jest za pomocą czterech kolejnych liczb uzyskanych z generatora SplitMix64.

```
public long nextLong() {
        long result = rotl(x0 + x3, 23) + x0;
        long t = x1 << 17;
        x2 ^= x0;
        x3 ^= x1;
        x1 ^= x2;
        x0 ^= x3;
        x2 ^= t;
        x3 = rotl(x3, 45);
        return result;
    }
```

Listing 2.3: Fragment Xoshiro256++ zaimplementowanego w Javie Źródło: opracowanie własne na podstawie [8]

Listing 2.3 przedstawia fragment implementacji generatora Xoshiro256++ w Javie. W pierwszej kolejności generowany jest wynik, słowo pierwsze i czwarte są do siebie dodawane, wynik jest obracany w lewo o 23 bity, a następnie dodawane jest do niego ponownie słowo pierwsze. Zmienna *t* tworzona jest poprzez przesunięcie bitowe wyrazu pierwszego w lewo o 17 bitów. Wykorzystywana jest ona do aktualizacji stanu generatora. Następnie dokonywana jest seria operacji XOR na słowach przestrzeni stanu i obrót w lewo wyrazu czwartego. Stan generatora został w ten sposób zaktualizowany.

```
function nextLong(rng::Xoshiro256PlusPlus)
    s0, s1, s2, s3 = rng.s
    result = rotl(s0 + s3, 23) + s0

    t = s1 << 17

    s2 = s2 ⊻ s0
    s3 = s3 ⊻ s1
    s1 = s1 ⊻ s2
    s0 = s0 ⊻ s3
    s2 = s2 ⊻ t
    s3 = rotl(s3, 45)

    rng.s = (s0, s1, s2, s3)
    return result
end
```

Listing 2.4: Fragment Xoshiro256++ zaimplementowanego w Julii Źródło: opracowanie własne na podstawie [8]

Na listingu 2.4 przedstawiono funkcję generującą kolejną liczbę 64 bitową w generatorze Xoshiro256++ w Julii. Implementacja działa w analogiczny sposób jak w przypadku Javy.

Powyższe funkcje przedstawiają sposób generowania liczb 64 bitowych ze znakiem, w przypadku Javy jest to typ Long a w przypadku Julii Int64. By uzyskać liczby typu Int oraz Double wynik funkcji jest przekształcany.

```
 public int nextInt() {
        long r = nextLong();
        return (int)(r >> 32)& 0x7FFFFFFF;
    }
```

Listing 2.5: Fragment funkcji generujący liczby 32 bitowe zaimplementowanej w Javie Źródło: opracowanie własne

Listing 2.5 przedstawia funkcje generującą liczby 32 bitowe bez znaku w Javie. Na wylosowanej liczbie dokonywane jest przesunięcie arytmetyczne w prawo o 32 bity. Dzięki temu 32 najstarsze bity znajdą się na 32 najmłodszych pozycjach. Następnie na 32 najmłodszych bitach stosowana jest maska bitowa ustawiająca bit znaku na 0.

```
 function nextInt(rng::Xoshiro256PlusPlus)
    r = nextLong(rng)
    return Int32((r >> 32) & 0x7FFFFFFF)
end
```

Listing 2.6: Fragment funkcji generujący liczby 32 bitowe zaimplementowanej w Julii Źródło: opracowanie własne

Przedstawiona na listingu 2.6 implementacja w Julii działa w identyczny sposób.

```
function nextDouble(rng::Xoshiro256PlusPlus)

    raw = nextLong(rng)
    shifted = raw >>> 11

    return Float64(shifted) / (1 << 53)
end
```

Listing 2.7: Fragment funkcji generujący liczby 64 bitowe zmiennoprzecinkowe zaimplementowanej w Julii Źródło: opracowanie własne

Listing 2.7 przedstawia funkcje generującą liczby 64 bitowe zmiennoprzecinkowe z przedziału [0,1) w Julii. Na wylosowanej liczbie 64 bitowej dokonywane jest przesunięcie logiczne o 11 bitów w prawo. Dzięki temu liczba jest nieujemna i zachowywane są 53 najsilniejsze bity. Bity młodsze mają niższą złożoność liniową i są bardziej przewidywalne [8]. Potrzebne są tylko 53 bity ze względu na rozmiar mantysy wynoszący 52 bity oraz 1 bit ukryty. Następnie całość jest rzutowana na liczbę zmiennoprzecinkową 64 bitową i dzielona jest przez 2^{53}.

```
public double nextDouble() {
       long raw = nextLong();
       long shifted = raw>>>11;
       return (double)(shifted) / (1L << 53);
   }
```

Listing 2.8: Fragment funkcji generujący liczby 64 bitowe zmiennoprzecinkowe zaimplementowanej w Javie Źródło: opracowanie własne

Przedstawione na listingu 2.8 Generowanie w Javie działa w analogiczny sposób.

2.4. Opis implementacji algorytmu

Opisane implementacje algorytmu genetycznego dla problemu komiwojażera w Julii i Javie zostały odwzorowane pod względem logicznym. Celem było uzyskanie najbardziej zbliżonych implementacji. Przy tworzeniu implementacji wykorzystano książkę Schildt H., Java. Kompendium programisty[13], dokumentację języka Java [1] oraz dokumentację z oficjalnej strony internetowej języka Julia [4].

Reprezentacja

Reprezentacja pojedynczego osobnika w Julii została zaimplementowana w postaci struktury Path.

```
mutable struct Path
   cities :: Vector{Int}
   fitness ::Float64
```

Listing 2.9: Fragment definicji struktury Path w Julii

Na listingu 2.9 przedstawiono definicje struktury Path. Użycie słowa kluczowego "mutable", pozwala na zmianę pól instancji struktury po jego utworzeniu. Struktura posiada dwa pola. Pierwsze z nich to "cities". Jest to dynamiczna tablica jednowymiarowa przechowująca liczby typu Int. W Julii "Vector"oraz jednowymiarowa tablica "Array"są równoważnymi strukturami danych. Pole "fitness"przechowuje wartość funkcji przystosowania dla danego osobnika. Jest ona odwrotnością całkowitego kosztu trasy. Zapewnia to wzrost jakości rozwiązania wraz ze wzrostem wartości funkcji dla problemu minimalizacyjnego, jakim jest problem komiwojażera.

```
public class Path {
    public ArrayList<Integer> cities;
    public double fitness;
```

Listing 2.10: Fragment definicji klasy Path w Javie

Listing 2.10 przedstawia definicje klasy Path w Javie. Pole "cities"zostało zaimplementowane poprzez strukturę danych "ArrayList". Jest ona zbliżona do jednowymiarowego wektora w Julii, ze względu na brak synchronizacji i większą wydajność.

Dla struktury Path zaimplementowane zostały trzy konstruktory. Konstruktor generujący losową ścieżkę, konstruktor generujący ścieżkę metodą Najbliższych Sąsiadów oraz generujący pustego osobnika.

```
function Path(weights ::  Vector{Vector{Int}},rng)

        n = length(weights)
        cities = collect(1:n)
        shuffle(cities,rng);
        obj =new(cities,0)
        calcFitness(obj, weights)
        return obj

    end
```

Listing 2.11: Fragment konstruktora generującego ścieżki w Julii

Przedstawiony na Listingu 2.11 konstruktor w Julii tworzy losowego osobnika. Jako parametry przyjmuje on dwuwymiarowy wektor z wagami krawędzi grafów oraz generator liczb losowych. Tworzony jest wektor z liczbami reprezentującymi miasta ułożonymi rosnąco. Następnie przy pomocy funkcji "shuffle()"miasta są mieszane i tworzony jest nowy obiekt przy pomocy konstruktora domyślnego. Funkcja "calcFitnnes()"oblicza wartość funkcji przystosowania i ją aktualizuje. Nowo utworzony obiekt jest zwracany.

```
public Path(ArrayList<ArrayList<Integer>> weights, Xoshiro256PlusPlus rng)
    {
        cities = new ArrayList<>();
        for(int i =0 ; i<weights.size(); i++)
        {
            cities.add(i);
        }
        shuffle(cities,rng);
        calcFitness(weights);
    }
```

Listing 2.12: Fragment konstruktora generującego ścieżki w Javie

Konstruktor w Javie przedstawiony na listingu 2.12 również tworzy nowy obiekt, lecz w porównaniu do Julii występują różnice. Kluczową zmianą jest sposób tworzenia nowego obiektu. W Julii instancja struktury tworzona jest wewnątrz konstruktora przy użyciu słowa kluczowego "new()". Konstruktor wywoływany jest jak normalna funkcja. Natomiast w Javie słowo kluczowe "new"używane jest przy wywołaniu konstruktora, a nie w jego środku. Obiekt tworzony jest w momencie wywołania konstruktora, a nie w jego wnętrzu.

```
 function Path(weights ::  Vector{Vector{Int}}, index :: Int)
        cities = Vector{Int}()
        push!(cities,index)
        bestConnection = typemax(Int)
        bestVertex  = -1
        size = length(weights)

        for i in 1:(size-1)
            bestConnection = typemax(Int)
            bestVertex = -1
            for k in 1:size
            if !(k in cities) && bestConnection > weights[cities[i]][k]
                bestConnection = weights[last(cities)][k]
                bestVertex = k
                end
            end
```

```
            push!(cities,bestVertex)
        end

        obj =new(cities,0)
        calcFitness(obj, weights)
        return obj
    end
```

Listing 2.13: Fragment konstruktora generującego ścieżki w Julii z użyciem algorytmu NN

Listing 2.13 przedstawia konstruktor tworzący instancję struktury z wykorzystaniem algorytmu Najbliższego Sąsiada. Jako parametr początkowy przyjmowany jest wektor z wagami krawędzi oraz indeks wierzchołka od, którego algorytm ma rozpocząć działanie. Jest on pierwszym wierzchołkiem ścieżki. Dla każdego wierzchołka począwszy od początkowego poszukiwany jest wierzchołek ze zbioru nie odwiedzonych, od którego dzieli go najmniejsza droga. Po znalezieniu dodawany jest on do ścieżki. Na końcu tworzony jest obiekt z uzyskaną listą miast i zerowym przystosowaniem. Przy pomocy funkcji "calcFitnnes"przystosowanie jest obliczane i zwracany jest nowy obiekt.

```
public Path(ArrayList<ArrayList<Integer>> weights, int index)
    {
        cities = new ArrayList<>();
        cities.add(index);
        int bestConnection = Integer.MAX_VALUE;
        int bestVertex=-1;
        int size = weights.size();
        for(int i=1;i<size;i++)
        {
            bestConnection = Integer.MAX_VALUE;
            bestVertex=-1;
            for(int k=0;k<size;k++)
            {
                if(!cities.contains(k) && bestConnection>weights.get(cities
                    ↪ .get(i-1)).get(k))
                {
                    bestConnection = weights.get(cities.get(i-1)).get(k);
                    bestVertex=k;
                }

            }
            cities.add(bestVertex);
        }
        calcFitness(weights);
    }
```

Listing 2.14: Fragment konstruktora generującego ścieżki w Javie z użyciem algorytmu NN

Przedstawiony na listingu 2.14 konstruktor w Javie działa w analogiczny sposób. Ponownie różnica polega na sposobie tworzenia nowych obiektów. W przypadku Javy obiekt nie jest tworzony wewnątrz konstruktora.

```
function shuffle(list::Vector{Int}, rng)
    n = length(list)
    for i in n-1:-1:1
        j  = Int(mod(nextLong(rng), i + 1)) + 1
        list[i+1], list[j] = list[j], list[i+1]
    end
    return list
end
```

Listing 2.15: Funkcja mieszająca zaimplementowana w Julii wykorzystująca algorytm Fisher–Yates

Funkcja mieszająca wykorzystana do generowania losowych ścieżek wykorzystująca algorytm Fisher–Yates znany również jako Knuth shuffle[16] w Julii została przedstawiona na listingu 2.15. Wektor miast jest iterowany od końca do początku, dla każdego elementu losowany jest indeks i element jest zamieniany z elementem o wylosowanym indeksie. Wylosowany indeks jest z zakresu od 0 do *i*. Gwarantuje to, że wylosowany indeks będzie należeć do elementu nie przetasowanego.

```
public static void shuffle(ArrayList<Integer> list, Xoshiro256PlusPlus rng
    ↪ ) {
        int n = list.size();
        for (int i = n - 1; i > 0; i--) {
            int j = (int) Math.floorMod(rng.nextLong(), i + 1);
            int temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
```

Listing 2.16: Funkcja mieszająca zaimplementowana w Javie wykorzystująca algorytm Fisher–Yates

Listing 2.16 przedstawia funkcję tasującą w Javie. Różnica w stosunku do implementacji w Julii polega na sposobie indeksowanie, od 0 w Javie, oraz wykorzystanie zmiennej pomocniczej przy zamianie wartości. Wykorzystanie w Julii "mod()"oraz "Math.FloorMod()"gwarantuje uzyskanie liczby zawsze nie ujemnej, nawet jeżeli wynik "nextLong()"generatora będzie ujemny.

Selekcja

W programach zaimplementowano dwa rodzaje selekcji, selekcję turniejową oraz selekcje metodą ruletki. Selekcja turniejowa charakteryzuje się większym naciskiem na jakość osobnika i zawsze wybiera najlepszego, nawet jeżeli wartości funkcji przystosowania są zbliżone. Z kolei selekcja metodą ruletki charakteryzuje się mniejszym naciskiem selekcyjnym, przez co przy zbliżonych wartościach funkcji przystosowania przewaga lepszych osobników maleje.

```
function tournamentSelection(population ::  Vector{Path},newPopSize :: Int,
    ↪  tournamentSize :: Int,rng)::Vector{Path}
     selected ::Vector{Path}= Vector{Path}(undef, newPopSize)
     popSize ::Int = length(population)


      @inbounds for i in 1:newPopSize
         bestValue =0.0
         bestPath = nothing
          for k in 1:tournamentSize
             index = mod(Int(nextInt(rng)), popSize) + 1
             drawn = population[index]
             if drawn.fitness>bestValue
                 bestValue = drawn.fitness
                 bestPath = drawn
             end
         end

         selected[i]=bestPath
     end
     return selected
end
```

Listing 2.17: Funkcja implementująca selekcje turniejową w Julii

Na listingu 2.17 przedstawiono implementacje selekcji turniowej w Julii. Kluczowymi parametrami dla funkcji jest rozmiar turnieju oraz rozmiar nowej populacji. Z populacji losowo

wybierane są osobniki w ilości odpowiadającej rozmiarowi turnieju i najlepszy z nich dodawany jest do nowej populacji. Proces powtarzany jest aż do zapełniania wektora osobników wybranych o rozmiarze nowej populacji. Makro "@inbounds"na początku pętli wyłącza sprawdzanie poprawności indeksów przy odczycie i zapisie, co zwiększa wydajność pętli.

```
 private ArrayList<Path> tournamentSelection(ArrayList<Path>population,
    ↪  int newPopSize, int tournamentSize, Xoshiro256PlusPlus rng)
{
    ArrayList<Path> selected = new ArrayList<>(newPopSize);
    int popSize = population.size();

    for (int i = 0; i < newPopSize; i++) {
        double bestValue = 0;
        Path bestPath = null;

        for (int j = 0; j < tournamentSize; j++) {
            int index = rng.nextInt()%popSize;
            Path drawn = population.get(index);
            double fitness = drawn.fitness;
            if (fitness > bestValue) {
                bestValue = fitness;
                bestPath = drawn;
            }
        }
        selected.add(bestPath);
    }
    return selected;
}
```

Listing 2.18: Funkcja implementująca selekcje turniejową w Javie

Przedstawiona na listingu 2.18 implementacja selekcji turniejowej działa w analogiczny sposób jak w przypadku Julii. W przypadku Javy nie występuje korekcja indeksów by uwzględniać indeksowanie od 1. Warto zwrócić uwagę na sposób przypisania wybranej ścieżki do wektora. W przypadku Julii możliwe jest bezpośrednie przypisanie wartości przez indeks. W przypadku Javy "ArrayList"nie wspiera takiej operacji, z tego powodu wykorzystana jest operacja "add()".

```
function rouletteSelection(population ::  Vector{Path},newPopSize :: Int,
   ↪ rng)::Vector{Path}
    selected ::Vector{Path}= Vector{Path}(undef, newPopSize)
    fitnessSum :: Float64 = 0.0

    @inbounds @simd for i in 1:length(population)
        fitnessSum += population[i].fitness
    end

     @inbounds for i in 1:newPopSize
        partialSum =0.0
        r :: Float64 = nextDouble(rng) * fitnessSum
        for path in population
        partialSum+=path.fitness
        if partialSum>=r
            selected[i]=path;
            break;

        end
    end
    end
    return selected
```

```
end
```

Listing 2.19: Funkcja implementująca selekcję metodą ruletki w Julii

Listing 2.19 przedstawia implementacje selekcji metodą ruletki w Julii. W pierwszej kolejności sumowane są wartości funkcji przystosowania dla wszystkich osobników populacji. Następnie losowana jest liczba z zakresu [0,1) i mnożona przez uzyskaną sumę. Wynik to miejsce na teoretycznym kole ruletki, które zostało wybrane. Następnie odbywa się iterowanie po populacji i sumowanie wartości dla osobników, aż suma cząstkowa będzie równa lub przekroczy obliczone wcześniej miejsce na kole ruletki. Znaleziony osobnik dodawany jest do wektora osobników wybranych. Całość powtarzana jest, aż wektor będzie mieć rozmiar nowej populacji. Użyte na początku pętli makro "@simd"zapewnia kompilator, że iteracje są niezależne i ich kolejność może być zamieniana.

```
private ArrayList<Path> rouletteSelection(ArrayList<Path>population, int
    ↪ newPopSize,Xoshiro256PlusPlus rng)
    {
        ArrayList<Path> selected = new ArrayList<>(newPopSize);
        double fitnessSum =0.0;

        for (Path p : population){
            fitnessSum+=p.fitness;}

        for (int i = 0 ;i<newPopSize;i ++)
        {
            double r =rng.nextDouble() * fitnessSum;
            double partialSum =0.0;

            for(Path p :population){
                partialSum+=p.fitness;
                if(partialSum>=r){
                     selected.add(p);
                    break;}
            }
        }
        return selected;

    }
```

Listing 2.20: Funkcja implementująca selekcje metodą ruletki w Javie

Implementacja w Javie przedstawiona na listingu 2.20 działa w prawie identyczny sposób. Występują różnice związane z dodawaniem do listy osobników oraz brak makr dotyczących pętli.

Krzyżowanie

Zdecydowano się na zaimplementowanie dwóch operatorów krzyżowania. Pierwszy z nich to krzyżowanie z zachowaniem porządku (OX), który pozwala zachować cykliczny porządek z rodzica. Drugi z operatorów to krzyżowanie dwupunktowe, gdzie miasta z rodzica uzupełniane są w sposób liniowy, od początku. Te dwa operatory prezentują odmienne podejścia, które w kontekście niniejszej pracy mogą wpłynąć na wydajność algorytmu.

```
    indexP = mod1(crossPoint2 + 1, size)
    indexC = indexP
    while indexC != crossPoint1
        val = parent2.cities[indexP]
```

```
        if !in(val, set1)
            child1.cities[indexC] = val
            indexC = mod1(indexC + 1, size)
            push!(set1, val)
        end
        indexP = mod1(indexP + 1, size)
    end
```

Listing 2.21: Fragment funkcji implementującej krzyżowanie OX w Julii

Na listingu 2.21 przedstawiono kluczowy fragment krzyżowania OX w Julii, uzupełnianie potomka 1 przez miasta z rodzica 2. Informacje na temat obecności danego miasta w potomku przechowywane są za pomocą struktury "BitSet". Jest to bardzo wydajna struktura, która przechowuje informacje na temat obecności elementu w zbiorze za pomocą bitów, nie przechowując pełnych liczb. Jeżeli dana liczba jest w zbiorze to liczba mapowana jest na bit opowiadający jej wartości i ustawiany na 1. W pierwszej kolejności obliczane są indeksy, od których rozpocząć ma się przenoszenie w rodzicu i potomku. W obu przypadkach jest to pierwszy indeks za drugim punktem cięcia. Następnie dopóki indeks w potomku nie będzie równy pierwszemu punktowi cięcia, wykonywana jest pętla. Z rodzica, rozpoczynając od obliczonego indeksu pobierane jest miasto, następnie sprawdzane jest czy miasto znajduje się w potomku. Jeżeli miasto nie znajduje się w potomku, to jest do niego wstawiane. Indeks potomka jest zwiększany, jeżeli przekroczy on rozmiar osobnika, to zwiększanie kontynuowane jest od zera. Niezależnie od obecności miasta rodzica w potomku indeks rodzica jest zwiększany, w przypadku przekroczenia rozmiaru osobnika również jest zapętlany. Po zakończeniu pętli otrzymujemy pierwszego potomka, analogiczna pętla wykonywana jest dla drugiego potomka.

```
int indexP = (crossPoint2 + 1) % size;
int indexC = (crossPoint2 + 1) % size;
while (indexC != crossPoint1) {
    int city = parent2.cities.get(indexP);
    if (!child1Set.get(city)) {
        child1.cities.set(indexC, city);
        child1Set.set(city);
        indexC = (indexC + 1) % size;
    }
    indexP = (indexP + 1) % size;
}
```

Listing 2.22: Fragment funkcji implementującej krzyżowanie OX w Javie

Przedstawiony na listingu 2.22 fragment krzyżowania w Javie działa w analogiczny sposób. Informacje na temat obecności miast również przechowywane są w strukturze "BitSet". Jest ona zbliżona działaniem do struktury w Julii. W Javie obecność miasta zaznaczana jest poprzez operacje "set()", która bezpośrednio ustawia wybrany bit, w Julii natomiast odbywa się mapowanie liczby na odpowiadający jej bit.

```
positionC1 :: Int = crossPoint1+1
positionC2 :: Int = crossPoint1+1

for i in 1:size

    value :: Int = parent2.cities[i]
    if !in(value, set1)
        child1.cities[positionC1]=value
        push!(set1, value)
        positionC1+=1
```

```
        end

        value  = parent1.cities[i]
        if !in(value, set2)
            child2.cities[positionC2]=value
            push!(set2, value)
            positionC2+=1

        end

    end
```

Listing 2.23: Fragment funkcji implementującej krzyżowanie dwupunktowe w Julii

Listing 2.23 przedstawia fragment krzyżowania dwupunktowego w Julii. Odpowiada on za uzupełnianie potomków miastami z rodziców. Indeksy w potomkach ustawiane są bezpośrednio za pierwszym punktem przecięcia tak, by wypełnić lukę miedzy oboma punktami. Pętla iteruje po obu całych rodzicach równocześnie. Jeżeli dane miasto nie znajduje się w potomku to jest wstawiane do niego, a indeks zwiększa się. Informacje na temat obecności miast ponownie przechowywane są w strukturze "BitSet". Następnie pobierane jest miasto z drugiego rodzica i warunek sprawdzany jest dla drugiego potomka.

```
            int positionC1 = crossPoint1+1;
        int positionC2 = crossPoint1+1;
        for(int i= 0;i<size;i++)
        {
            int value = parent2.cities.get(i);
            if(!child1Set.get(value))
            {
                child1.cities.set(positionC1,value);
                child1Set.set(value);
                positionC1++;
            }
            value = parent1.cities.get(i);
            if(!child2Set.get(value))
            {
                child2.cities.set(positionC2,value);
                child2Set.set(value);
                positionC2++;
            }
        }
```

Listing 2.24: Fragment funkcji implementującej krzyżowanie dwupunktowe w Javie

Przedstawiony na listingu 2.24 fragment krzyżowania w Javie działa w identyczny sposób jak krzyżowanie w Julii. Ponownie wykorzystano strukturę "BitSet".

Mutacja

W kodzie zdecydowano się na zaimplementowanie mutacji przez zamianę oraz inwersję. Są to skrajne podejścia, zamiana wprowadza minimalne zmiany, natomiast inwersja w skrajnym przypadku zamienia kolejność miast w całym osobniku.

```
         if nextDouble(rng) <mR
        p1 :: Int = mod(Int(nextInt(rng)), size) + 1
        p2 :: Int = mod(Int(nextInt(rng)), size) + 1
        @inbounds individual.cities[p1], individual.cities[p2] = individual
           ↪ .cities[p2], individual.cities[p1]
```

```
end
```

Listing 2.25: Fragment funkcji implementującej mutacje przez zamianę w Julii

Na listingu 2.25 przedstawiono fragment funkcji mutacji przez zamianę. Fakt wykonania mutacji losowany jest w funkcji zgodnie z podanym współczynnikiem. Losowane są dwie liczby które są normalizowane do rozmiaru osobnika, a geny na wylosowanych pozycjach są zamieniane. Przy zamianie miast wyłączone zostało sprawdzanie poprawności indeksów.

```
if(rng.nextDouble()<mR) {
    int p1 = rng.nextInt()%size;
    int p2 = rng.nextInt()%size;
    Collections.swap(individual.cities, p1, p2);
}
```

Listing 2.26: Fragment funkcji implementującej mutacje przez zamianę w Javie

Listing 2.26 przedstawia analogiczną operację w Javie. Do zamiany genów wykorzystana jest funkcja "swap()"z biblioteki Collections, ze względu na brak możliwości bezpośredniej zamiany zmiennych jak w przypadku Julii.

```
@inbounds while p1 < p2
        individual.cities[p1], individual.cities[p2] = individual.
            ↪ cities[p2], individual.cities[p1]
        p1 += 1
        p2 -= 1
    end
```

Listing 2.27: Fragment funkcji implementującej mutacje przez inwersję w Julii

Mutacja przez inwersję przedstawiono została na listingu 2.27. Kolejność miast między dwoma wylosowanymi punktami jest odwracana. Punkty losowane są bez ograniczenia i mogą znajdować się w dowolnej odległości od siebie. Po losowaniu punktu zamieniane są tak, by punkt pierwszy był mniejszy od punktu drugiego, następnie dopóki punkt pierwszy jest mniejszy od drugiego wykonywana jest pętla. Wartości na indeksach wyznaczonych przez punkty są zamieniane, a punkty zbliżane są do siebie. Punkt pierwszy jest zwiększany, a drugi zmniejszany. Przy pętli zamieniającej wyłączone sprawdzanie poprawności indeksów.

```
while(p1<p2)
        {
            Collections.swap(individual.cities, p1, p2);
            p1++;
            p2--;
        }
```

Listing 2.28: Fragment funkcji implementującej mutacje przez inwersję w Javie

Listing 2.28 przedstawia analogiczną operację w Javie. Ponownie ze względu na brak możliwości bezpośredniej zamiany wartości, wykorzystywana jest funkcja "swap()".

2.5. Instancje problemu komiwojażera

Do badań wykorzystane zostały instancje problemu komiwojażera z bazy TSPLIB [20]. Zapewnia ona instancje problemu reprezentowane w wielu formatach. Do badań wykorzystane zostały instancje asymetrycznego problemu komiwojażera reprezentowane przez macierz sąsiedztwa z wagami.

```
NAME: ftv47
TYPE: ATSP
COMMENT: Asymmetric TSP (Fischetti)
DIMENSION: 48
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
   100000000          39         156          141          135          183
```

Listing 2.29: Fragment macierzy ftv47.atsp z bazy TSPLIB

Listing 2.29 przedstawia fragment macierzy ftv47.atsp z bazy TSPLIB. Zgodnie z dokumentacją TSPLIB [21] parametr "EXPLICT"oznacza, że wagi krawędzi podane są jawnie. Parametr "FULL_MATRIX"oznacza, że wagi podane są poprzez pełną macierz. Koszt 100000000 reprezentuje przejście z danej krawędzi do niej samej.

W programie zaimplementowanym w języku Java dane wczytywane są poprzez funkcję "loadData()"z klasy "DataLoader", wagi przechowywane są w dwuwymiarowej strukturze "ArrayList". Każda waga to jedna liczba 32 bitowa ze znakiem.

W przypadku Julii wczytywane poprzez funkcję "LoadData". Wagi przechowywane są w dwuwymiarowej strukturze typu "Vector". Każda waga to jedna liczba 32 bitowa ze znakiem.

Rozdział 3

Analiza otrzymanych wyników

3.1. Wprowadzenie

W niniejszym rozdziale opisano metodę badawczą oraz wyniki przeprowadzonych badań. Celem badania było porównania wykonywania obliczeń w Julii i Javie pod względem czasowym, a także szczytowej zajętości pamięciowej.

W celu uzyskania wyników o wysokiej jakości badanie zostało podzielone na dwa etapy. W pierwszym etapie obliczenia zostały przeprowadzone na szerokim zakresie konfiguracji algorytmu. Celem tego było wyłonione konfiguracji metod algorytmu gwarantujących jak najlepsze wyniki pod względem kosztu problemu komiwojażera. W drugim etapie wybrane konfiguracje poddane zostały obszerniejszym pod względem ilości powtórzeń badaniom, tak by uzyskać wyniki o najwyższej jakości dotyczące czasu i pamięci.

3.2. Metoda badawcza

Zastosowana metoda badawcza opierała się na implementacjach algorytmu w Julii i Javie, które były możliwie najbardziej do siebie zbliżone. Wszystkie parametry wejściowe oraz metody stosowane podczas poszczególnych etapów algorytmu były identyczne. Dzięki własnej implementacji generatora liczb losowych wywołania w obu językach dla identycznych parametrów dawały identyczne wyniki pod względem kosztów znalezionej optymalnej trasy. Jako kryterium zatrzymania obrano liczbę generacji.

Badania zostało przeprowadzone w środowisku wykorzystującym wersję 1.12.0 Julii oraz OpenJDK 23.0.1 z HotSpot 64-Bit Server w przypadku Javy. Badania przeprowadzone na jednostce moblinej podłączonej stale do zasilania, żadne inne zadania nie były wykonywane. Program napisany w Julii uruchamiany był w wierszu poleceń przy pomocy komendy "julia". Program napisany w Javie najpierw był kompilowany przy pomocy komendy "javac", następnie uruchamiany z użyciem komendy "java". Obie czynności odbywały się w wierszu poleceń. Badania wszystkich konfiguracji dla danej instancji problemu odbywały się bezpośrednio po sobie, program po ukończeniu badania danej konfiguracji od razu rozpoczynał badanie kolejnej, nie był uruchamiany ponownie. Jednostka, która została wykorzystana do przeprowadzania badań posiadała konfigurację przedstawioną w tabeli 3.1.

Tab. 3.1: Konfiguracja sprzętowa użyta w eksperymentach

Parametr	**Wartość**
CPU	11th Gen Intel Core i5-1135G7 @ 2.40GHz
RAM	16 GB, 4266 MHz
System operacyjny	Windows 11 64-bit

Źródło: opracowanie własne

Zaimplementowany algorytm posiadał możliwość zmiany metod wykorzystywanych podczas jego poszczególnych etapów. Wybór metod dokonywany był podczas jego uruchamiania. Dostępne warianty przedstawione zostały w tabeli 3.2.

Tab. 3.2: Zastosowane operatory i metody w algorytmie genetycznym

Kategoria	**Warianty**
Inicjalizacja	Hybrydowa, Losowa
Selekcja	Turniejowa, Metoda ruletki
Krzyżowanie	Z zachowaniem porządku, Dwupunktowe
Mutacja	Przez zamianę, Przez inwersję

Źródło: opracowanie własne

Algorytm przyjmował następujące parametry dotyczące jego działania:

- rozmiar populacji,
- rozmiar elity [%],
- limit pokoleń,
- rozmiar turnieju dla selekcji turniejowej,
- współczynnik krzyżowania,
- współczynnik mutacji

Parametry były stałe dla każdego wywołania i nie były zmieniane dla danej konfiguracji.

Pomiar czasu rozpoczynany był po inicjalizacji pokolenia, przed rozpoczęciem głównej pętli algorytmu. Pomiar zatrzymywany był w momencie osiągnięcia warunku końcowego oraz opuszczenia głównej pętli. W Julii pomiar czasu odbywał się za pomocą polecania "time_ns()", natomiast w Javie z użyciem "System.nanoTime()". Błędy przypadkowe prawdopodobnie spowodowane były działaniem procesora. Działanie systemu oraz wzrost chwilowego obciążenia wpływało na uzyskiwane wyniki. Błędy systematyczne prawdopodobnie wynikały z różnicy między kompilacją Julii i Javy. Pierwsze uruchomienie funkcji w Julii wymaga więcej czasu. Błędy grube spowodowane prawdopodobnie były działaniem mechanizmów oczyszczania pamięci, przełączania kontekstu procesora lub innymi nietypowymi zachowaniami systemu operacyjnego.

Pomiar pamięci odbywał się w odstępach 100 milisekund. Mierzone było aktualne zużycie pamięci, a wartość maksymalna zaobserwowana w trakcie działania algorytmu była zapisywana. Pozwoliło to określić przybliżone szczytowe zużycie pamięci podczas działania programu. W Julii pomiar pamięci odbywał się z użyciem funkcji asynchronicznej, która działała równolegle z głównym programem. Wykorzystana została funkcja "Base.gc_live_bytes()", która zwraca rozmiar żywych obiektów w pamięci, które przetrwały ostatnie oczyszczanie pamięci oraz ilość bajtów zaalokowanych od tego czasu [3]. W Javie do pomiaru pamięci został wykorzystany osobny wątek. Zajęcie pamięci obliczane jest jako "rt.totalMemory() - rt.freeMemory();", jest to różnica całkowitej pamięci w maszynie wirtualnej oraz wolnej pamięci w maszynie wirtualnej[18]. Różnica reprezentuje przybliżoną ilość pamięci zajętą przez obiekty na stercie.

Przed rozpoczęciem pomiarów pamięci w obu implementacjach została wywołana funkcja wymuszająca oczyszczanie pamięci. W Julii wywołana jest funkcja "GC.gc()", natomiast w Javie "System.gc();". Należy zaznaczyć iż ten sposób pomiaru powoduje błąd systematyczny. Pomiar pamięci w Julii nie obejmuje obiektów czekających na usunięcie, które pomiar w Javie uwzględnia. Różnica tych podejść może spowodować zawyżenie wyników po stronie Javy. Błędy przypadkowe prawdopodobnie powodowane były przez uruchomienie odśmiecania pamięci przed lub po pomiarze pamięci. Błędy grube powodowane mogły być przez nagłe zapotrzebowanie pamięci przez inne procesy w tle lub uruchomienie pełnego odśmiecania w przypadku Javy.

Wszystkie uzyskane wyniki po każdym pomiarze wraz z parametrami pomiaru zapisywane były do pliku CSV. Umożliwiło to sprawną analizę uzyskanych danych. Podczas pierwszego etapu badania testowane były konfiguracje posiadające następujące zmienne:

- dwa rozmiary populacji,
- dwie metody inicjalizacji populacji,
- dwie metody selekcji,
- dwie metody krzyżowania,
- dwie metody mutacji

Parametry te pozwalają na stworzenie 32 konfiguracji algorytmu. W pierwszym etapie każdej konfiguracji badania zostały przeprowadzone dla 10 ziaren generatora. Czynnikiem decydującym o wyborze konfiguracji do kolejnego etapu była mediana błędu względnego dla danej konfiguracji. Badania przeprowadzone zostały dla dwóch instancji problemu.

W kolejnym etapie najlepsza konfiguracja dla danej instancji problemu została przetestowana dla 50 jednakowych ziaren w obu językach. W tym etapie kluczowy był czas wykonania oraz maksymalne zużycie pamięci przez algorytm. Porównanie czasów wykonania algorytmów oraz szczytowej zajętości pamięci zostało przeprowadzone z wykorzystaniem testów statystycznych.

Podczas drugiego etapu badań wszystkie uzyskane wyniki zostały uwzględnione w analizie. Wartości odstające w czasie wykonania oraz zużyciu pamięci interpretowano jako wynik działania kilku czynników: zróżnicowanych sekwencji liczb losowych, procesu rozgrzewania kompilatora JIT i zmiennej pracy oczyszczania pamięci. Z tych powodów wartości odstające traktowano jako część zmiennej natury całego układu. W przypadku identyfikacji oczywistych błędów wynikających z awarii środowiska uruchomieniowego zostałyby one wykluczone z analizy a odpowiednie eksperymenty powtórzone.

Procedura badawcza zaprojektowana została tak, by uwzględnić wszystkie anomalie wyników. Wykorzystanie testu Shpiro-Wilka do oceny normalności rozkładu determinowało użycie testu t-studenta sparowanego lub Wilcoxona. Test Wilcoxona jest testem nieparametrycznym, dzięki czemu jest relatywnie odporny na wartości odstające.

W pierwszej kolejności wykonany został test Shapiro-Wilka, by ocenić normalność rozkładu uzyskanych wyników. Test przeprowadzono na różnicach badanych wartości uzyskanych dla danych ziaren. Hipoteza zerowa testu brzmi: dane mają rozkład normalny. Jeżeli uzyskany parametr p jest mniejszy od 0.05, H0 jest odrzucane. Dane nie mają rozkładu normalnego.

Następnie w zależności od uzyskanego wyniku przeprowadzano t-Studenta sparowany lub test Wilcoxona. Jeżeli różnice miały rozkład normalny wykonano test t-Studenta sparowany, a w przeciwnym wypadku test Wilcoxona [14]. Testy przeprowadzone zostały w wariantach obustronnym i jednostronnym. Wariant obustronny pozwala ocenić czy różnica między czasami wykonywania algorytmu między oboma językami jest statystycznie istotna. Natomiast test jednostronny pozwala ocenić czy różnica występuje w określonym kierunku [9]. W obu przypadkach przyjęto poziom istotności na poziomie $\alpha = 0.05$.

Tab. 3.3: Hipotezy dla testów statystycznych parowanych

Test	**Hipoteza zerowa (H_0)**	**Hipoteza alternatywna (H_1)**
Test obustronny		
t-Studenta (sparowany)	Średnia różnic = 0	Średnia różnic $\neq 0$
Wilcoxona	Mediana różnic = 0	Mediana różnic $\neq 0$
Test jednostronny (pierwsza próbka > druga)		
t-Studenta (sparowany)	Średnia różnic ≤ 0	Średnia różnic > 0
Wilcoxona	Mediana różnic ≤ 0	Mediana różnic > 0
Test jednostronny (pierwsza próbka < druga)		
t-Studenta (sparowany)	Średnia różnic ≥ 0	Średnia różnic < 0
Wilcoxona	Mediana różnic ≥ 0	Mediana różnic < 0

Źródło: opracowanie własne

W tabeli 3.3 przedstawiono hipotezy dla testów statystycznych dla danych sparowanych. Test t-studenta sparowany opiera się na średniej, natomiast test Wilcoxona ze względu na brak rozkładu normalnego używa mediany. Jeżeli uzyskany parametr *p* jest mniejszy od 0.05, H0 zostaje odrzucone.

3.3. Warunki wstępne symulacji

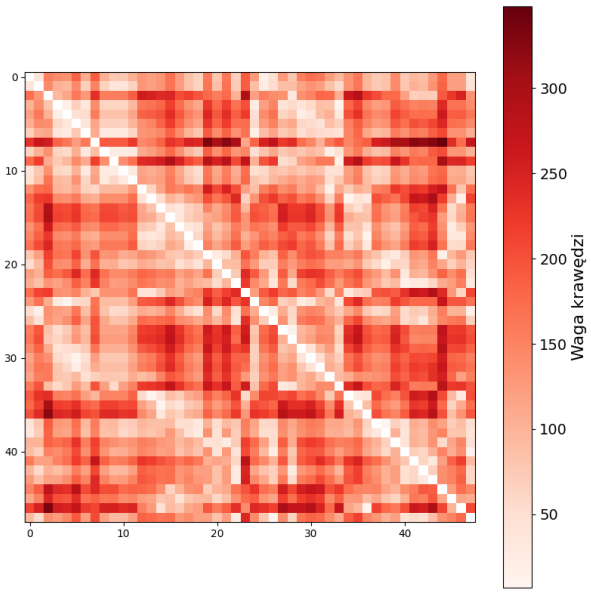
Badania zostały przeprowadzone wykorzystując dwie instancje asymetrycznego problemu komiwojażera z zasobów TSPLIB [20]. Instancje wraz z najlepszymi rozwiązaniami przedstawione zostały w tabeli 3.4.

Tab. 3.4: Użyte instancje problemu ATSP

Nazwa instancji	**Liczba wierzchołków**	**Optymalny koszt**
ftv47	47	1776
ftv170	170	2755

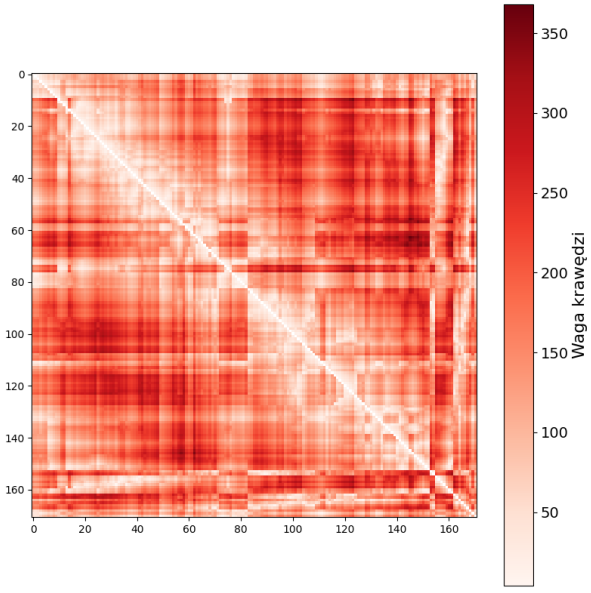
Źródło: opracowanie własne

Charakteryzują się one zróżnicowaniem pod względem rozmiaru oraz pod względem trudności znalezienia rozwiązania o wysokiej jakości.



Rys. 3.1: Wizualizacja wag krawędzi grafu Ftv47

Na rysunku 3.1 przedstawiono wizualizację wag krawędzi grafu Ftv47. Zaobserwować można równomierne rozłożenie jaśniejszych obszarów na całym obszarze mapy. Oznaczają one krawędzie o niższych wagach. Świadczy to o równomiernym rozłożeniu krawędzi, o niskim koszcie w całym grafie, co ułatwia znajdowanie jakościowych rozwiązań, szczególnie ze względu na mały rozmiar grafu.



Rys. 3.2: Wizualizacja wag krawędzi grafu Ftv170

Na rysunku 3.2 przedstawiono wizualizację wag krawędzi grafu Ftv170. Zaobserwować można mniej jaśniejszych obszarów. Głównie znajdują się one przy przekątnej mapy. Świadczy to o skupiskach krawędzi o niskim koszcie w pewnych obszarach grafu, co może utrudniać znajdowanie globalnie dobrych rozwiązań, szczególnie ze względu na duży rozmiar grafu.

Tab. 3.5: Parametry algorytmu genetycznego użyte w eksperymentach

Parametr	**Wartość**
Wielkość populacji	1000, 2000
Limit pokoleń	20000
Rozmiar elity	5%
Prawdopodobieństwo mutacji mR	0.01
Prawdopodobieństwo krzyżowania cR	0.95
Rozmiar turnieju	2

Źródło: opracowanie własne

Dla przeprowadzonych badań przyjęto parametry przedstawione w tabeli 3.5, ustalone zostały one eksperymentalnie. Przedstawione konfiguracje okazały się najbardziej uniwersalne i pozwoliły uzyskać satysfakcjonujące wyniki dla każdej instancji problemu. W pierwszym etapie badań ziarno ustalone było jako dziesięć kolejnych wyrazów ciągu Fibonacciego, rozpoczynając od wyrazu 3. W drugiej fazie ziarno ustalone było jako pierwsze pięćdziesiąt kwadratów liczb naturalnych.

3.4. I etap badań

Celem pierwszego etapu badań było wyłonienie konfiguracji metod algorytmu pozwalających osiągnąć wyniki o najwyższej jakości. Dla każdej konfiguracji badanie powtórzono 10 razy ze zmiennym ziarnem, jakość całej konfiguracji oceniona została poprzez mediany błędów względnych. Na tym etapie badań czas oraz szczytowa zajętość pamięci nie są brane pod uwagę.

Tab. 3.6: Mediany błędu dla pliku Ftv47

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
1	1000	losowa	OX	zamiana	truniejowa	8,53%
2	1000	losowa	OX	inwersja	truniejowa	7,85%
3	1000	losowa	TPX	zamiana	truniejowa	22,58%
4	1000	losowa	TPX	inwersja	truniejowa	41,33%
5	1000	losowa	OX	zamiana	ruletka	2,36%
6	1000	losowa	OX	inwersja	ruletka	2,42%
7	1000	losowa	TPX	zamiana	ruletka	16,95%
8	1000	losowa	TPX	inwersja	ruletka	8,64%
9	1000	hybrydowa	OX	zamiana	truniejowa	3,66%
10	1000	hybrydowa	OX	inwersja	truniejowa	4,50%
11	1000	hybrydowa	TPX	zamiana	truniejowa	14,41%
12	1000	hybrydowa	TPX	inwersja	truniejowa	19,65%
13	1000	hybrydowa	OX	zamiana	ruletka	2,42%
14	1000	hybrydowa	OX	inwersja	ruletka	2,42%
15	1000	hybrydowa	TPX	zamiana	ruletka	10,78%
16	1000	hybrydowa	TPX	inwersja	ruletka	10,39%
17	2000	losowa	OX	zamiana	truniejowa	5,63%
18	2000	losowa	OX	inwersja	truniejowa	5,55%
19	2000	losowa	TPX	zamiana	truniejowa	25,96%
20	2000	losowa	TPX	inwersja	truniejowa	39,22%
21	2000	losowa	OX	zamiana	ruletka	1,69%

Kontynuacja tabeli 3.6

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
22	2000	losowa	OX	inwersja	ruletka	2,39%
23	2000	losowa	TPX	zamiana	ruletka	16,27%
24	2000	losowa	TPX	inwersja	ruletka	6,76%
25	2000	hybrydowa	OX	zamiana	truniejowa	4,50%
26	2000	hybrydowa	OX	inwersja	truniejowa	4,50%
27	2000	hybrydowa	TPX	zamiana	truniejowa	14,41%
28	2000	hybrydowa	TPX	inwersja	truniejowa	18,83%
29	2000	hybrydowa	OX	zamiana	ruletka	2,42%
30	2000	hybrydowa	OX	inwersja	ruletka	2,42%
31	2000	hybrydowa	TPX	zamiana	ruletka	13,06%
32	2000	hybrydowa	TPX	inwersja	ruletka	8,50%

Źródło: opracowanie własne

W tabeli 3.6 przedstawiono mediany błędów względnych dla badanych konfiguracji algorytmów dla grafu Ftv47. Mediany błędów kształtują się w zakresie od 1,69% do 41,33 %. Jako wartość progową mediany ustalono 2,4 %. Pozwoliło to wyodrębnić 3 konfiguracje do następnego etapu badań. Konfiguracje wybrane przedstawiono w tabeli 3.7.

Tab. 3.7: Wybrane konfiguracje dla pliku Ftv47

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
5	1000	losowa	OX	zamiana	ruletka	2,36%
21	2000	losowa	OX	zamiana	ruletka	1,69%
22	2000	losowa	OX	inwersja	ruletka	2,39%

Źródło: opracowanie własne

Tab. 3.8: Wyniki testów statystycznych dla wybranych konfiguracji - Ftv47

lp.	Konf. 1	Konf. 2	Konf.1 - Mdn	Konf.2 - Mdn	Rodzaj testu	p(A > B)	p(A < B)
1	5	21	0.023649	0.016892	t-Studenta p.	0.460237	0.539763
2	5	22	0.023649	0.023930	t-Studenta p.	0.443330	0.556670
3	21	22	0.016892	0.023930	t-Studenta p.	0.510848	0.489152

Źródło: opracowanie własne

W tabeli 3.8 przedstawiono testy jednostronne dla prób sparowanych przeprowadzone dla wybranych konfiguracji algorytmu. Żadna z median nie wskazuje statystycznie istotnej różnicy w porównaniu do pozostałych.

Tab. 3.9: Mediany błędu dla pliku Ftv170

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
1	1000	losowa	OX	zamiana	truniejowa	42,14%
2	1000	losowa	OX	inwersja	truniejowa	36,59%
3	1000	losowa	TPX	zamiana	truniejowa	145,32%
4	1000	losowa	TPX	inwersja	truniejowa	279,07%
5	1000	losowa	OX	zamiana	ruletka	30,44%
6	1000	losowa	OX	inwersja	ruletka	32,07%

Kontynuacja tabeli 3.9

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
7	1000	losowa	TPX	zamiana	ruletka	105,54%
8	1000	losowa	TPX	inwersja	ruletka	70,64%
9	1000	hybrydowa	OX	zamiana	truniejowa	18,38%
10	1000	hybrydowa	OX	inwersja	truniejowa	19,75%
11	1000	hybrydowa	TPX	zamiana	truniejowa	22,43%
12	1000	hybrydowa	TPX	inwersja	truniejowa	28,35%
13	1000	hybrydowa	OX	zamiana	ruletka	10,56%
14	1000	hybrydowa	OX	inwersja	ruletka	10,09%
15	1000	hybrydowa	TPX	zamiana	ruletka	19,46%
16	1000	hybrydowa	TPX	inwersja	ruletka	22,27%
17	2000	losowa	OX	zamiana	truniejowa	27,10%
18	2000	losowa	OX	inwersja	truniejowa	34,03%
19	2000	losowa	TPX	zamiana	truniejowa	138,69%
20	2000	losowa	TPX	inwersja	truniejowa	248,80%
21	2000	losowa	OX	zamiana	ruletka	22,89%
22	2000	losowa	OX	inwersja	ruletka	24,81%
23	2000	losowa	TPX	zamiana	ruletka	103,76%
24	2000	losowa	TPX	inwersja	ruletka	54,21%
25	2000	hybrydowa	OX	zamiana	truniejowa	15,88%
26	2000	hybrydowa	OX	inwersja	truniejowa	16,73%
27	2000	hybrydowa	TPX	zamiana	truniejowa	22,41%
28	2000	hybrydowa	TPX	inwersja	truniejowa	28,35%
29	2000	hybrydowa	OX	zamiana	ruletka	10,09%
30	2000	hybrydowa	OX	inwersja	ruletka	8,55%
31	2000	hybrydowa	TPX	zamiana	ruletka	19,46%
32	2000	hybrydowa	TPX	inwersja	ruletka	21,14%

Źródło: opracowanie własne

W tabeli 3.9 przedstawiono mediany błędów względnych dla badanych konfiguracji algorytmów dla grafu Ftv170. Widoczna jest duża rozbieżność między medianami poszczególnych konfiguracji. Mediany błędów kształtują się w zakresie od 8,55 % do 279,07 %. Jako wartość progową przyjęto medianę na poziomie 15 %. Pozwoliło to wyodrębnić 4 najlepsze konfiguracje algorytmu do dalszych badań, zostały one przedstawione w tabeli 3.10.

Tab. 3.10: Wybrane konfiguracje dla pliku Ftv170

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja	Mediana błędu
13	1000	hybrydowa	OX	zamiana	ruletka	10,56%
14	1000	hybrydowa	OX	inwersja	ruletka	10,09%
29	2000	hybrydowa	OX	zamiana	ruletka	10,09%
30	2000	hybrydowa	OX	inwersja	ruletka	8,55%

Źródło: opracowanie własne

Tab. 3.11: Wyniki testów statystycznych dla wybranych konfiguracji - Ftv170

lp.	Konf. 1	Konf. 2	Konf.1 - Mdn	Konf.2 - Mdn	Rodzaj testu	p(A > B)	p(A < B)
1	13	14	0.105626	0.100907	t-Studenta p.	0.381584	0.618416
2	13	29	0.105626	0.100907	t-Studenta p.	0.114862	0.885138
3	13	30	0.105626	0.085481	Wilcoxona	0.002930	0.999023
4	14	29	0.100907	0.100907	t-Studenta p.	0.167467	0.832533
5	14	30	0.100907	0.085481	Wilcoxona	0.041992	0.967773
6	29	30	0.100907	0.085481	Wilcoxona	0.065430	0.947266

Źródło: opracowanie własne

W tabeli 3.11 przedstawiono testy jednostronne dla prób sparowanych przeprowadzone dla wybranych konfiguracji algorytmu. Mediana dla konfiguracji 30 jest najniższa i testy wskazują iż w porównaniu do konfiguracji 13 oraz 14, ma ona istotnie mniejszy błąd. W porównaniu do konfiguracji 29 nie można jednoznacznie wskazać konfiguracji o mniejszym błędzie, ponieważ w obu przypadkach hipoteza zerowa nie jest odrzucana.

3.5. II etap badań

Celem drugiego etapu było statystyczne porównanie wybranych konfiguracji algorytmu. Badania dla każdej konfiguracji zostało powtórzone 50 razy ze zmiennym ziarnem. Parametrami ocenianymi podczas tego etapu były czas wykonania algorytmu oraz zmierzona szczytowa zajętość pamięci.

W dalszej części dane odnoszące się do procentowej wydajności czasów oraz pamięci obliczane były jako (Wynik w Javie - Wynik w Julii)/(Wynik w Javie) * 100 %.

Ftv47

W dalszej części odniesienia do poszczególnych konfiguracji używane będą zgodnie z porządkiem przedstawionym w tabeli 3.12.

Tab. 3.12: Wybrane konfiguracje dla pliku Ftv47 - II etap badań

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja
1	1000	losowa	OX	zamiana	ruletka
2	2000	losowa	OX	zamiana	ruletka
3	2000	losowa	OX	inwersja	ruletka

Źródło: opracowanie własne

Konfiguracja 1

Tab. 3.13: Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 1

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	25051,2265	25791,3976	22616,2393	28847,8765	1507,334137
Java	32080,29682	33396,56325	26210,9853	35515,6019	2806,724338

Źródło: opracowanie własne

W tabeli 3.13 przedstawiono statystyki opisowe dla czasów konfiguracji 1. Mediana oraz średnia jest niższa w Julii, natomiast maksymalna wartości w Julii jest większa niż minimalna w Javie.

Tab. 3.14: Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 1

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,920727115	0,002505251
Wilcoxon (obustronny)	1	3,55271E-15
Wilcoxon (jednostronny, Julia>Java)	1	1
Wilcoxon (jednostronny, Julia<Java)	1	1,77636E-15

Źródło: opracowanie własne

W tabeli 3.14 przedstawiono wyniki testów statystycznych dla czasów konfiguracji 1. Test Shapiro-Wilka wykazał, iż różnice pomiarów nie posiadają cech rozkładu normalnego. Test Wilcoxona obustronny wskazuje na istotne różnice między wartościami. Testy jednostronne wskazują na fakt, iż większość różnic czasów (Julia - Java) jest ujemna, suma rang dodatnich różnic wynosi 1. Czasy w Julii są mniejsze niż w Javie.

Tab. 3.15: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 1

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	7,8407978	5,2786505	2,307862	29,950734	7,423092106
Java	79,53928	79,576	78,429	79,645	0,168814599

Źródło: opracowanie własne

Tabela 3.15 przedstawia statystyki opisowe dla szczytowej zajętości pamięci konfiguracji 1. Średnia i mediana Julii jest zdecydowanie niższa, natomiast dane w Javie charakteryzują się mniejszym rozproszeniem.

Tab. 3.16: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,584217038	1,14364E-10
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.16 przedstawiono wyniki testów statystycznych dla szczytowej zajętości pamięci konfiguracji 1. Test Shapiro-Wilka wykazał, iż różnice pomiarów nie posiadają cech rozkładu normalnego. Test Wilcoxona obustronny wskazuje na istotna różnicę między danymi. Testy jednostronne wskazują na istotną mniejszość szczytowej zajętości pamięci w Julii.

Konfiguracja 2

Tab. 3.17: Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 2

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	72642,42856	73875,44475	64012,4873	77331,3529	3596,011376
Java	87155,10279	87216,29405	85709,5035	89105,2549	851,9435911

Źródło: opracowanie własne

W tabeli 3.17 przedstawiono statystyki opisowe dotyczące czasów wykonania konfiguracji 2. Dane dla Julii mają mniejsze wartości niż dane dla Javy, natomiast dane dla Julii są bardziej rozproszone i posiadają większe odchylenie standardowe.

Tab. 3.18: Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 2

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,771890441	2,0919E-07
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.18 przedstawiono wyniki testów statystycznych dla czasów konfiguracji 2. Test Shapiro-Wilka nie wykazał normalności rozkładu danych. Test Wilcoxona obustronny wskazuje, że mediana różnic znacznie różni się od 0. Testy jednostronne wskazują na istotnie mniejszą od 0 medianę różnic. Wszystkie różnice w testach Wilcoxna mają ujemny znak.

Tab. 3.19: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 2

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	19,8599035	21,0867955	3,381639	30,392078	7,754059536
Java	80,08788	80,104	79,816	80,183	0,068087471

Źródło: opracowanie własne

W tabeli 3.19 przedstawiono statystki opisowe dla szczytowej zajętości pamięci dla konfiguracji 2. Wartości dla Julii są zdecydowanie niższe niż dla Javy. Dane dla Julii są bardziej rozproszone i posiadają większe odchylenie standardowe.

Tab. 3.20: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,928262889	0,004752525
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.20 przedstawiono wyniki testów statystycznych dla szczytowej zajętości pamięci konfiguracji 2. Test Shapiro-Wilka nie wykazał normalności rozkładu danych. Obustronny test Wilcoxona wykazał, iż mediana różnic istotnie różni się od 0. Testy Wilcoxona jednostronne wykazały, iż wszystkie różnice są ujemne i dane dla Julii są istotnie mniejsze.

Konfiguracja 3

Tab. 3.21: Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 3

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	73174,03283	73449,357	64322,1663	75400,7371	1838,788441
Java	87417,3288	87448,3439	85654,6773	89148,6953	781,5119366

Źródło: opracowanie własne

W tabeli 3.21 przedstawiono statystki opisowe dla czasów konfiguracji 3. Średnia oraz mediana dla Julii są mniejsze, dane dla Javy są mniej rozproszone.

Tab. 3.22: Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 3

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,642803951	8,89513E-10
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.22 przedstawiono wyniki testów statystycznych dla czasów konfiguracji 3. Test Shapiro-Wilka wykazał że różnice czasów nie mają rozkładu normalnego. Test Wilcoxona obustronny wykazał, że mediana jest istotnie różna od 0. Testy jednostronne wykazały, iż mediana jest istotnie mniejsza od 0, czasy w Julii są mniejsze. Wszystkie różnice w testach Wilcoxona są ujemne.

Tab. 3.23: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 3

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	18,50059792	21,043667	3,528474	29,317	7,855193844
Java	80,07254	80,082	79,928	80,158	0,050378247

Źródło: opracowanie własne

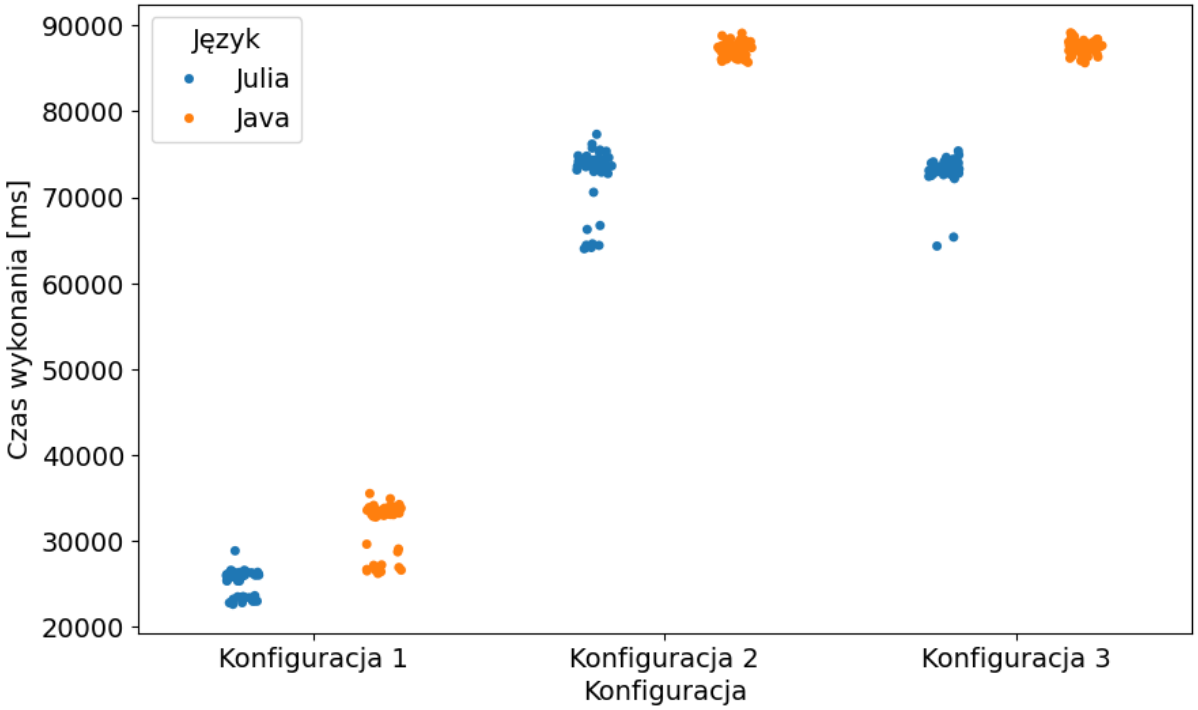
W tabeli 3.23 przedstawiono statystyki opisowe dla szczytowej zajętości pamięci konfiguracji 3. Dane dla Julii są zdecydowanie mniejsze niż dla Javy. Dane dla Javy posiadają minimalne rozproszenie.

Tab. 3.24: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,909541355	0,001008024
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

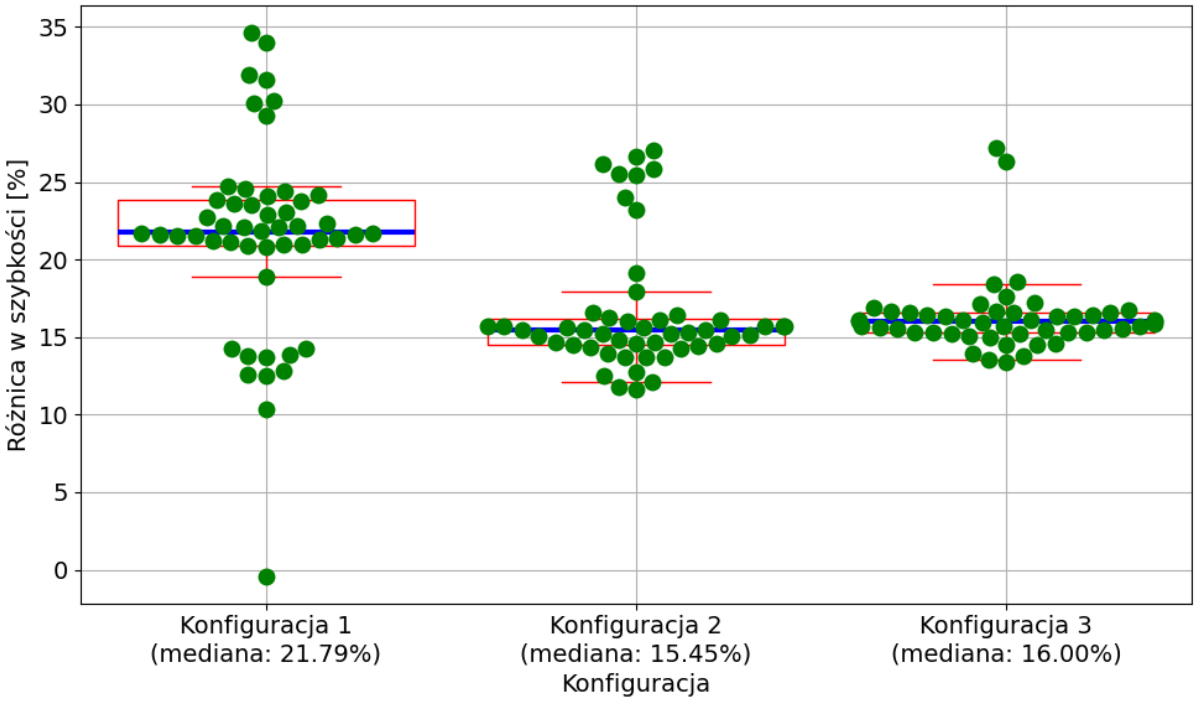
Źródło: opracowanie własne

W tabeli 3.24 przedstawiono wyniki testów statystycznych dla szczytowej zajętości pamięci konfiguracji 3. Test Shapiro-Wilka nie wykazał normalności rozkładu różnic. Test Wilcoxona obustronny wskazuje na fakt iż mediana różnic jest istotnie różna od 0. Natomiast testy jednostronne informują o tym iż mediana różnic jest istotnie mniejsza od 0. W testach Wilcoxona brak różnic od dodatnim znaku.



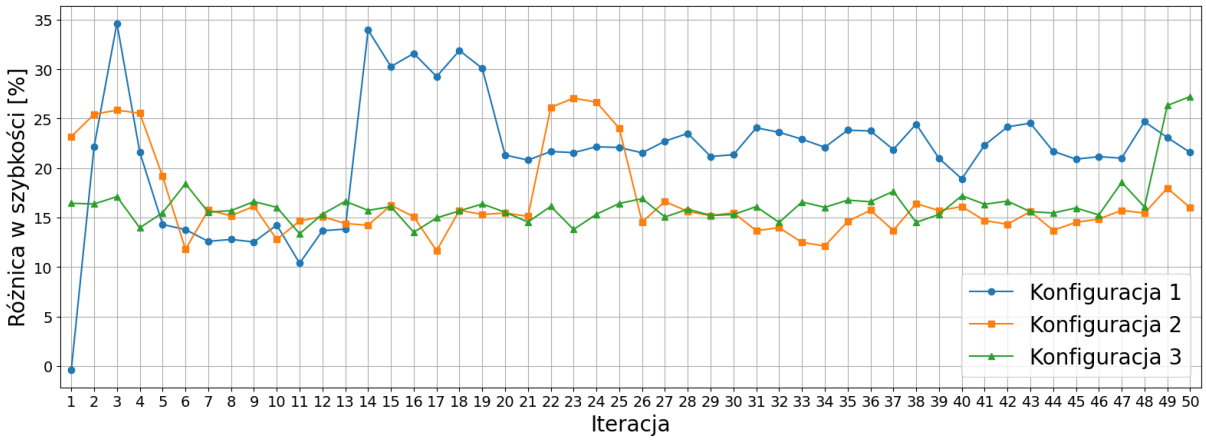
Rys. 3.3: Porównanie czasów wykonania dla konfiguracji 1,2,3 - Ftv47

Rysunek 3.3 przedstawia wykres porównujący czasy wykonania dla wszystkich 3 konfiguracji w obu językach. Zaobserwować można różnice czasów wykonania w obu językach. Dla konfiguracji 1 wyniki obu języków są bardziej zbliżone niż dla pozostałych. W obu językach można zaobserwować dwa skupiska pomiarów. Widoczne jest niskie rozproszenie danych w Javie dla konfiguracji 2 oraz 3. W przypadku Julii dla konfiguracji 2 oraz 3 zaobserwować można pojedyncze pomiary odstające od pozostałych.



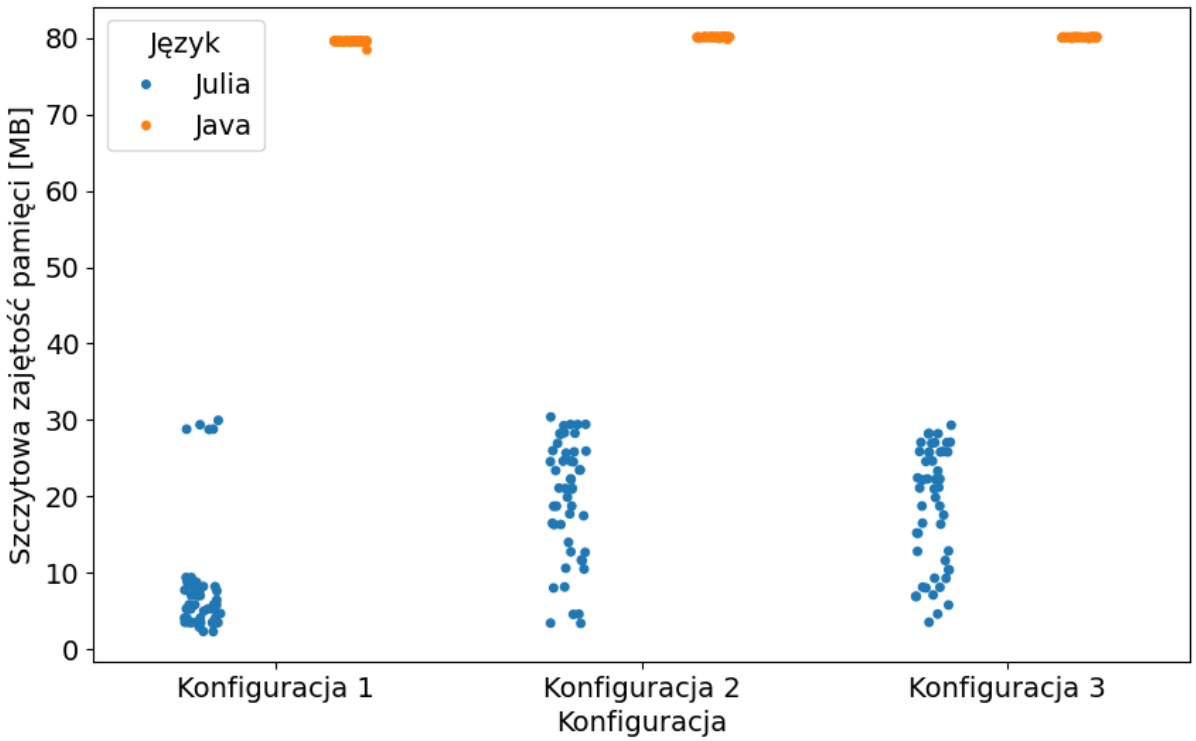
Rys. 3.4: Różnice czasów wykonania dla konfiguracji 1,2,3 - Ftv47

Rysunek 3.4 przedstawia różnice procentowe szybkości dla instancji Ftv47. Widoczna jest zdecydowana przewaga szybkości na korzyść Julii w większości różnic. Dla konfiguracji 1 jedna wartość zdecydowanie odstaje od pozostałych i jako jedyna posiada znak ujemny. W przypadku konfiguracji 1 rozrzut różnic jest największy. Dla konfiguracji 2 oraz 3 rozrzut jest mniejszy, wszystkie różnice są na korzyść Julii.



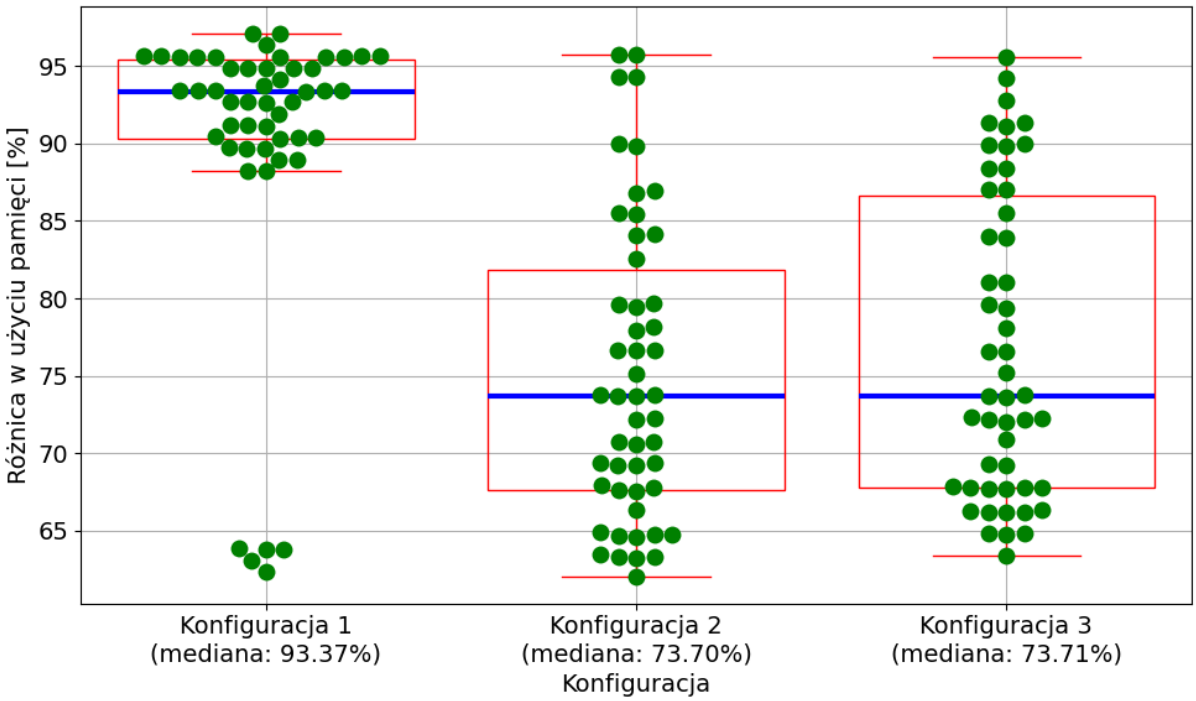
Rys. 3.5: Procentowe różnice czasów wykonania dla konfiguracji 1,2,3 w kolejnych iteracjach - Ftv47

Rysunek 3.5 przedstawia zmianę procentowej różnicy szybkości dla instancji Ftv47 w kolejnych iteracjach. Widoczny jest wzrost znaczący różnicy w ciągu pierwszych iteracji konfiguracji 1. Wahania różnicy dla konfiguracji 1 są większe, w porównaniu do konfiguracji 2 i 3.



Rys. 3.6: Porównanie szczytowej zajętości pamięci dla konfiguracji 1,2,3 - Ftv47

Rysunek 3.6 przedstawia wykres porównujący szczytowej zajętości pamięci dla wszystkich 3 konfiguracji w obu językach. Dane dla Javy są zdecydowanie mniej rozproszone. W przypadku Julii dla konfiguracji 1 możemy zaobserwować pojedynczy pomiary odstające od pozostałych, dla konfiguracji 2 i 3 w Julii rozproszenie jest zdecydowanie widoczne.



Rys. 3.7: Różnice szczytowej zajętości pamięci dla konfiguracji 1,2,3 - Ftv47

Rysunek 3.7 przedstawia różnice procentowe szczytowej zajętości pamięci dla instancji Ftv47. Różnice dla konfiguracji 1 podzieliły się na dwa zbiory. Jeden większy dla większej różnicy oraz mniejszy około 63%. Dla konfiguracji 2 oraz 3 dane są zdecydowanie bardziej rozproszone. Różnice dla obu konfiguracji rozciągają się od 63% do 96%.

Ftv170

W dalszej części odniesienia do poszczególnych konfiguracji używane będą zgodnie z porządkiem przedstawionym w tabeli 3.25.

Tab. 3.25: Wybrane konfiguracje dla pliku Ftv170 - II etap badań

lp.	Rozmiar populacji	Inicjalizacja	Krzyżowanie	Mutacja	Selekcja
1	1000	hybrydowa	OX	zamiana	ruletka
2	1000	hybrydowa	OX	inwersja	ruletka
3	2000	hybrydowa	OX	zamiana	ruletka
4	2000	hybrydowa	OX	inwersja	ruletka

Źródło: opracowanie własne

Konfiguracja 1

Tab. 3.26: Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 1

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	52296,14263	51959,46605	51322,0353	60376,0052	1403,082782
Java	89884,25519	92746,17885	69690,8621	95338,2202	7368,552993

Źródło: opracowanie własne

Tabela 3.26 przedstawiano statystyki opisowe dla czasów uzyskane dla konfiguracji 1. W przypadku Julii zaobserwować można mniejsze rozproszenie wyników niż w przypadku Javy, co potwierdzają wartości Min i Max, a także odchylenie standardowe.

Tab. 3.27: Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 1

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,565475038	6,18485E-11
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.27 przedstawiono wyniki przeprowadzonych testów statystycznych dla czasów konfiguracji 1. Test Shapiro-Wilka odrzucił hipotezę zerową, rozkład danych nie jest normalny. Test Wilcoxona obustronny potwierdził obecność istotnej różnicy między danymi. Testy jednostronne wskazują na istotną mniejszość mediany różnic od 0. Wartość statystyki testów Wilcoxona równa 0 świadczy o tym, że wszystkie różnice są ujemne, każdy czas w Julii jest niższy.

Tab. 3.28: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	18,8538979	18,6526965	9,266191	29,263342	5,462018858
Java	97,92148	99,3365	81,417	111,375	5,273122443

Źródło: opracowanie własne

W tabeli 3.28 przedstawiono statystyki opisowe dla szczytowej zajętości pamięci programów. Odchylenia standardowe dla obu języków mają podobną wartość.

Tab. 3.29: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,894365614	0,000314709
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.29 przedstawiono wyniki testów statystycznych przeprowadzonych dla szczytowej zajętości pamięci konfiguracji 1. Test Shapiro-Wilka wykazał iż różnice pomiarów nie posiadają cech rozkładu normalnego. Test Wilcoxona obustronny wskazuje na istotną różnicę mediany od 0. Testy jednostronne silnie wskazują na mniejszość mediany różnic od 0, co świadczy o mniejszych szczytowych zajętościach pamięci uzyskanych w Julii. Statystyka testów Wilcoxona ponownie posiada wartość 0, wszystkie różnice są ujemne.

Konfiguracja 2

Tab. 3.30: Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 2

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	51839.30	51787.91	50973.43	52814.49	403.80
Java	93152.62	93180.24	90872.15	96980.85	883.30

Źródło: opracowanie własne

W tabeli 3.30 przedstawiono statystyki opisowe dla czasów konfiguracji 2. W przypadku obu języków wyniki są skoncentrowane, wartości odchyleń standardowych nie odbiegają od siebie znacznie.

Tab. 3.31: Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 2

Test	Statystyka	p-value
Shapiro-Wilk dla rożnic	0.9199	0.0023
Wilcoxon (obustronny)	0.0000	0.0000
Wilcoxon (jednostronny, Julia>Java)	0.0000	1.0000
Wilcoxon (jednostronny, Julia<Java)	0.0000	0.0000

Źródło: opracowanie własne

W tabeli 3.31 przedstawiono wyniki testów statystycznych przeprowadzonych dla czasów konfiguracji 2. Test Shapiro-Wilka wykazał, iż różnice pomiarów nie posiadają cech rozkładu normalnego. Test Wilcoxona obustronny wskazuje, iż mediana różnic istotnie różni się od 0, natomiast testy jednostronne informują iż mediana jest istotnie mniejsza od 0.

Tab. 3.32: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	17,61053506	17,2564135	7,463166	27,888303	5,210952605
Java	94,40742	97,26	81,257	99,49	6,239688274

Źródło: opracowanie własne

W tabeli 3.32 przedstawiono statystyki opisowe dla szczytowej zajętości pamięci konfiguracji 2. Oba języki wykazują podobne odchylenie standardowe. Średnia wartość jak i mediana w Julii są zdecydowanie niższe.

Tab. 3.33: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,935127314	0,008683833
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.33 przedstawiono wyniki testów statystycznych przeprowadzonych dla szczytowych zajętości pamięci konfiguracji 2. Hipoteza zerowa testu Shapiro-Wilka zostaje, rozkład danych nie jest normalny. Test Wilcoxona obustronny wskazuję na istotną różnice między danymi. Natomiast testy jednostronne wskazują że mediana różnic istotnie jest mniejsza od 0, co wskazuje na mniejsze szczytowe zużycie pamięci w Julii.

Konfiguracja 3

Tab. 3.34: Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 3

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	125135,6455	125047,2995	123510,898	127102,4234	657,8275522
Java	210753,7483	211228,6447	166139,2094	224335,6263	7334,874463

Źródło: opracowanie własne

W tabeli 3.34 przedstawiono statystki opisowe dotyczące czasów konfiguracji 3. Średnia oraz mediana czasów Julii jest niższa. Dane w przypadku Julii są bardziej skoncentrowane niż w przypadku Javy.

Tab. 3.35: Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 3

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,514462522	1,26641E-11
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

Źródło: opracowanie własne

W tabeli 3.35 przedstawiono testy statystyczne przeprowadzone dla czasów konfiguracji 3. Test Shapiro-Wilka nie wykazał normalności rozkladu danych. Test obustronny Wilcoxona wskazuje na silną różnicę między danymi. Testy obustronne wykazały iż, mediana różnic jest istotnie mniejsza od 0, czasy wykonania w Julii są mniejsze. Dla testów Wilcoxona statystka ma wartość 0, wszystkie różnice są ujemne.

Tab. 3.36: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	61,47125642	63,749505	39,022401	78,18767	10,45386124
Java	140,9138	142,716	117,17	163,956	7,974376928

Źródło: opracowanie własne

W tabeli 3.36 przedstawiono statystki opisowe szczytowej zajętości pamięci dla konfiguracji 3. Dane dla Julii posiadają większe odchylenie standardowe. Pozostałe wartości w przypadku Javy są większe.

Tab. 3.37: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,985748379	0,80325116
t-test (obustronny)	-43,05285714	1,33704E-40
t-test (jednostronny, Julia>Java)	-43,05285714	1
t-test (jednostronny, Julia<Java)	-43,05285714	6,68522E-41

Źródło: opracowanie własne

W tabeli 3.37 przedstawiono wyniki testów statystycznych dla szczytowej zajętości pamięci konfiguracji 3. Test Shapiro-Wilka wykazał, iż dane wykazują cechuy rozkładu normalnego. Z tego powodu wykorzystany został t-test sparowany. Test obustronny wskazuje, iż średnia różnic istotnie różni się od 0, natomiast testy jednostronne wskazują na mniejszą zajętości pamięci Julii. Średnia różnic jest istotnie ujemna.

Konfiguracja 4

Tab. 3.38: Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 4

Język	Średnia [ms]	Mediana [ms]	Min [ms]	Max [ms]	SD [ms]
Julia	125087,1307	124944,6823	123829,6111	127773,2299	696,0704209
Java	211429,3159	211325,7052	208638,2846	215583,4809	1609,118012

Źródło: opracowanie własne

W tabeli 3.38 przedstawiono statystki opisowe czasów konfiguracji 4. Średnia oraz mediana dla Julii są mniejsze. Dane dla Javy są bardziej rozproszone.

Tab. 3.39: Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 4

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,966809995	0,171160776
t-test (obustronny)	-371,6464775	3,34472E-86
t-test (jednostronny, Julia>Java)	-371,6464775	1
t-test (jednostronny, Julia<Java)	-371,6464775	1,67236E-86

Źródło: opracowanie własne

W tabeli 3.39 przedstawiono wyniki testów statystycznych dla czasów konfiguracji 4. Test Shapiro-Wilka wykazał, iż różnice pomiarów posiadają cechy rozkładu normalnego. Wyniki t-testu obustronnego wskazują, iż średnia różnic istotnie różni się od 0. Testy jednostronne wskazują na mniejsze czasy wykonania w Julii, średnia różnic jest istotnie ujemna. Wysoka ujemna wartość statystyki t oznacza, iż średnia różnic jest silnie ujemna.

Tab. 3.40: Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 4

Język	Średnia [MB]	Mediana [MB]	Min [MB]	Max [MB]	SD [MB]
Julia	56,01930868	53,8220695	38,552033	82,464347	11,67104689
Java	140,97478	141,491	117,491	147,489	5,246165473

Źródło: opracowanie własne

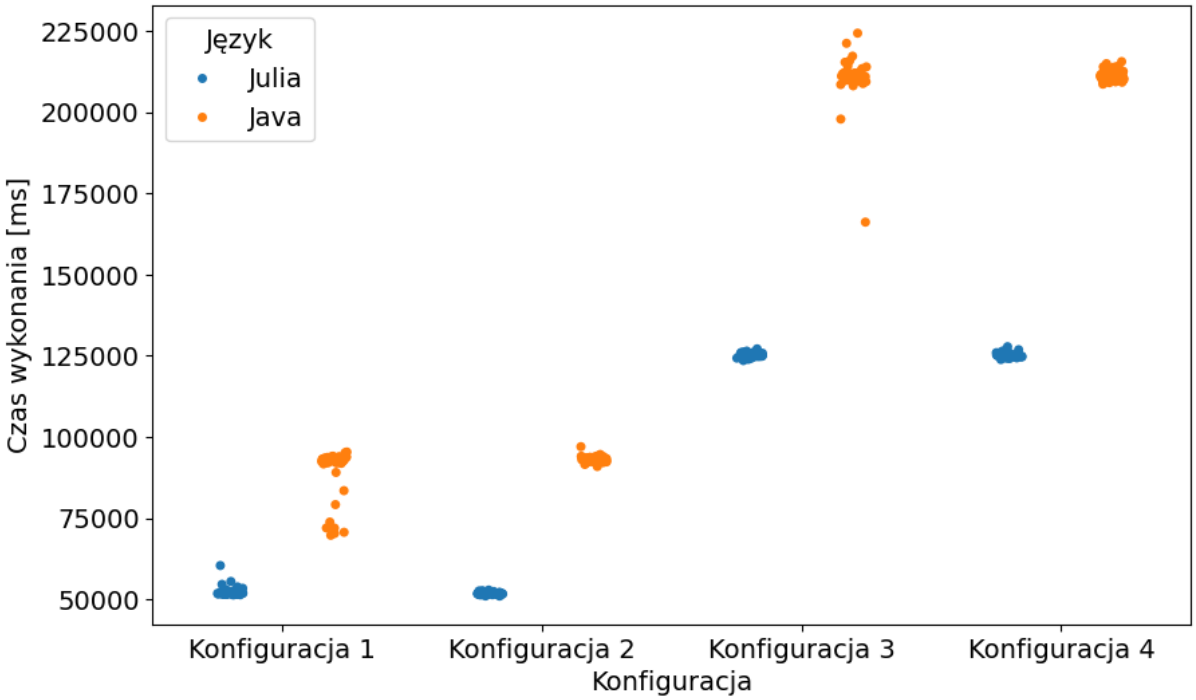
W tabeli 3.40 przedstawiono statystki opisowe szczytowej zajętości pamięci konfiguracji 4. Dane dla Javy są bardziej skupione. Pozostałe wartości w przypadku Julii są niższe.

Tab. 3.41: Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 4

Test	Statystyka	p-value
Shapiro-Wilk dla różnic	0,9392745	0,012612403
Wilcoxon (obustronny)	0	1,77636E-15
Wilcoxon (jednostronny, Julia>Java)	0	1
Wilcoxon (jednostronny, Julia<Java)	0	8,88178E-16

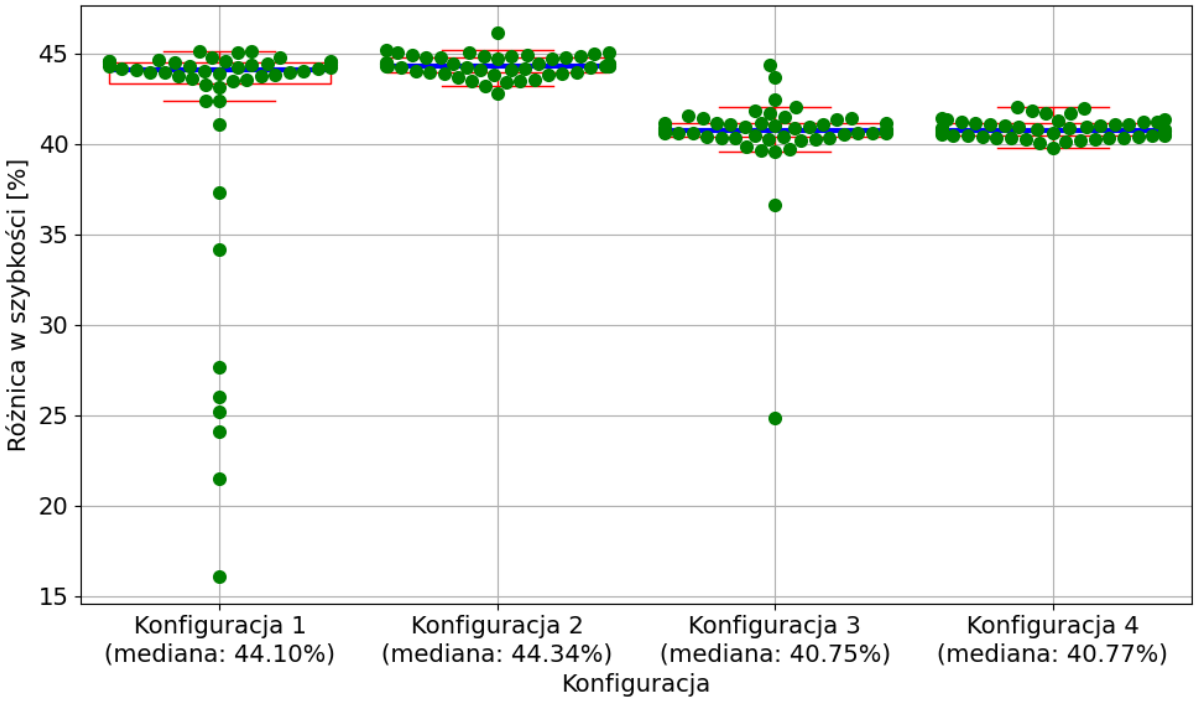
Źródło: opracowanie własne

W tabeli 3.41 przedstawiono wyniki testów statystycznych dla szczytowej zajętości pamięci konfiguracji 4. Test Shapiro-Wilka wykazał, iż różnice pomiarów nie posiadają cech rozkładu normalnego. Test Wilcoxona obustronny wskazuje na istotna różnicę miedzy danymi. Testy jednostronne wskazują na mniejszą szczytową zajętość pamięci w Julii, mediana różnic jest istotnie mniejsza od 0.



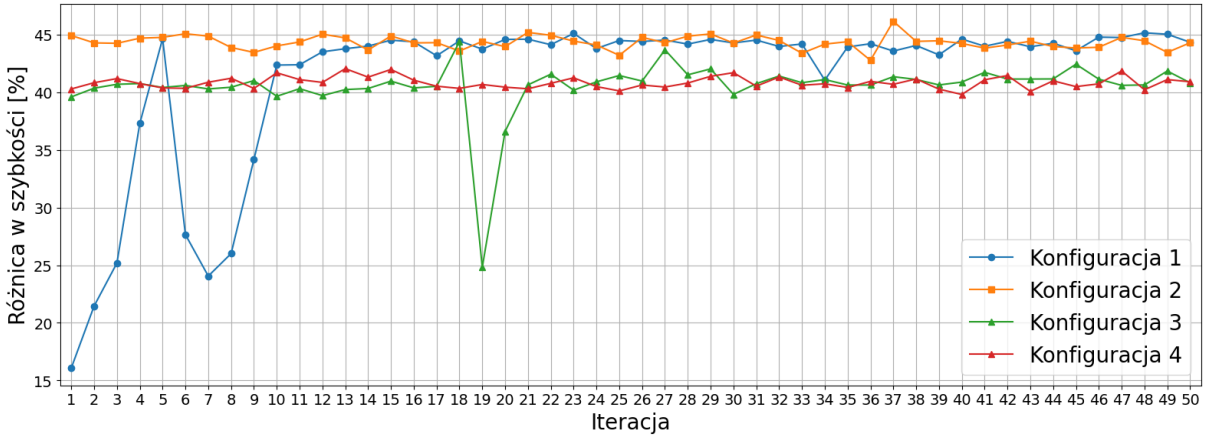
Rys. 3.8: Porównanie czasów wykonania dla konfiguracji 1,2,3,4 - Ftv170

Rysunek 3.8 przedstawia wykres porównujący czasy w obu językach dla wszystkich 4 konfiguracji. Widoczne są znaczące różnice w wartościach, wyniki dla Julii są mocno skupione dla wszystkich konfiguracji. W przypadku Javy dla konfiguracji 1 oraz 3 widoczne są pojedyncze wartości odstające od pozostałych, natomiast konfiguracje 2 oraz 4 są mocno skupione.



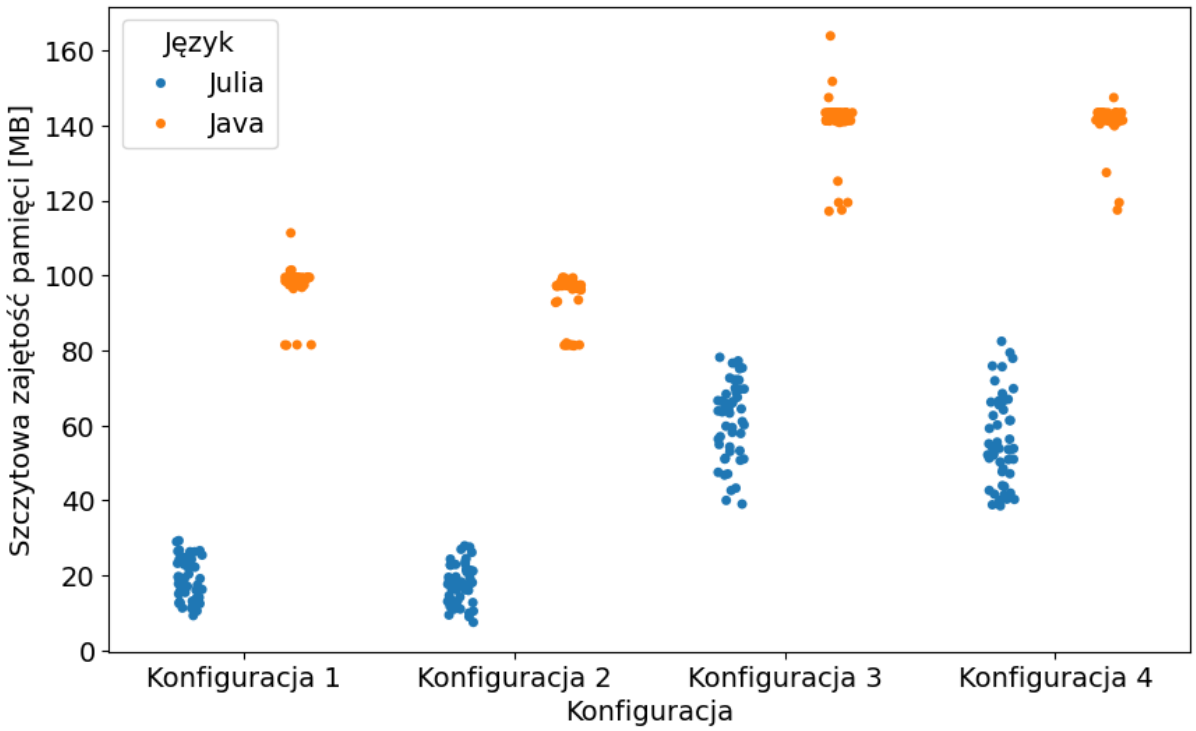
Rys. 3.9: Różnice czasów wykonania dla konfiguracji 1,2,3,4 - Ftv170

Rysunek 3.9 przedstawia różnice procentowe szybkości dla instancji Ftv170. Dla konfiguracji 1 można zaobserwować główne skupienie różnic około 43% oraz pojedyncze wyniki odstające. Dla konfiguracji 2 oraz 4 wyniki są mocno zbite. Dla konfiguracji 3 dwa wyniki odstają od pozostałych. Widoczny jest odmienny poziom wysokości różnicy dla konfiguracji 1 i 2 oraz 3 i 4.



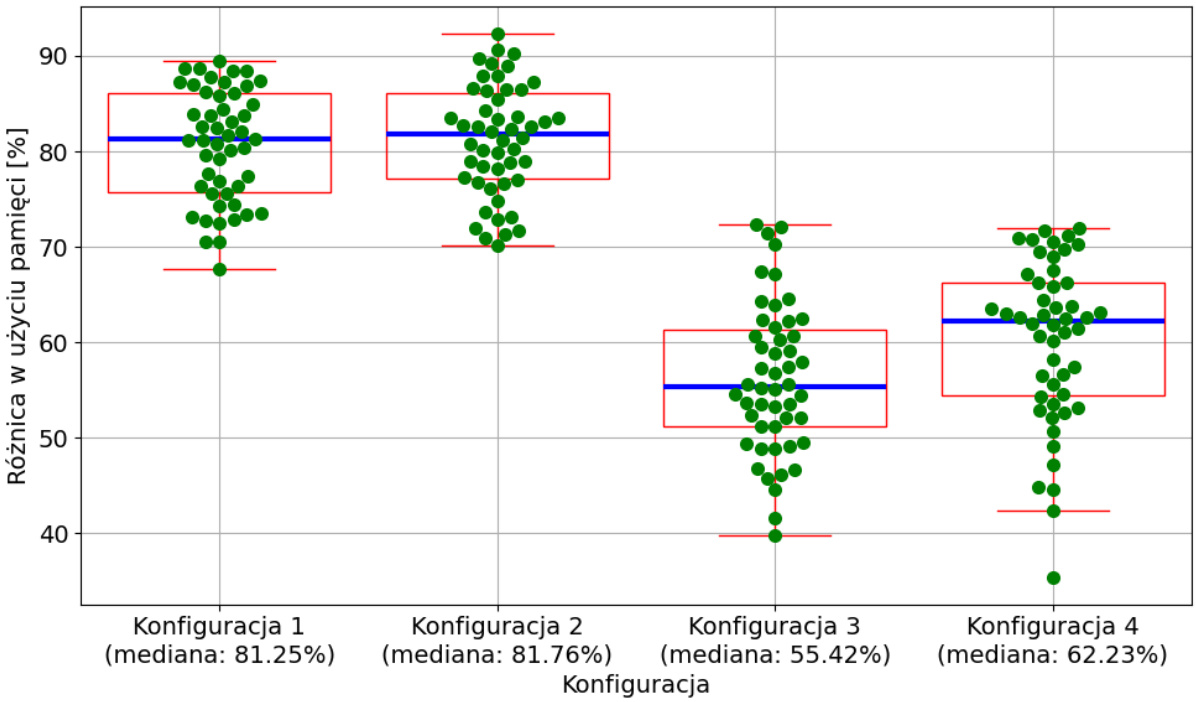
Rys. 3.10: Procentowe różnice czasów wykonania dla konfiguracji 1,2,3,4 w kolejnych iteracjach - Ftv170

Rysunek 3.10 przedstawia zmianę procentowej różnicy szybkości dla instancji Ftv170. Widoczny jest znaczący wzrost oraz spadek różnicy w ciągu pierwszych iteracji dla konfiguracji 1. Wraz z kolejnymi iteracjami różnice dla konfiguracji 1 stabilizują się. Dla konfiguracji 3 widoczny jest spadek różnicy dla iteracji 19. Poza wskazanymi odchyleniami różnice w trakcie trwania symulacji są na stabilnym poziomie.



Rys. 3.11: Porównanie szczytowej zajętości pamięci dla konfiguracji 1,2,3,4 - Ftv170

Rysunek 3.11 przedstawia wykres porównujący szczytową zajętość pamięci w obu językach dla wszystkich 4 konfiguracji. Widoczne jest większe rozproszenie danych w przypadku Julii. W przypadku Javy próbek odstających od pozostałych jest niewiele. Wartości dla Julii są zauważalnie niższe.



Rys. 3.12: Różnice szczytowej zajętości pamięci dla konfiguracji 1,2,3,4 - Ftv170

Rysunek 3.12 przedstawia różnice procentowe szczytowej zajętości pamięci dla instancji Ftv170. Widoczna jest znacząca różnice między różnicami dla konfiguracji 1 i 2 oraz 3 i 4. Wyniki dla konfiguracji 1 oraz 2 są bardziej skupione oraz wyższe w stosunku do konfiguracji 3 i 4. Natomiast wyniki dla konfiguracji 3 i 4 są bardziej rozproszone oraz niższe.

Podsumowanie

Celem przeprowadzonych badań było porównanie efektywności implementacji w językach Julia i Java algorytmu genetycznego dla problemu komiwojażera. Badania przeprowadzone zostały w kontekście wydajności czasowej oraz szczytowej zajętości pamięci. Badanie składało się z dwóch etapów. W pierwszym etapie wyłoniono konfiguracje algorytmu umożliwiające uzyskanie najlepszych rozwiązań. W drugim etapie wybrane konfiguracje badane były z wykorzystaniem testów statystycznych pod względem szybkości oraz szczytowej zajętości pamięci.

Z otrzymanych wyników jednoznacznie można wywnioskować że kod napisany w Julii charakteryzuje się większą szybkością. Dane dotyczące szybkości wykonywania kodu w większości przypadków nie posiadały cech rozkładu normalnego.

Przy danych nie posiadających rozkładu normalnego przeprowadzone testy obustronne Wilcoxona w każdym przypadku wskazywały, iż mediana różnic istotnie różni się od 0. Natomiast test jednostronny przy teście (Julia>Java) odrzucały hipotezę alternatywną mówiącą iż mediana różnic jest większa od 0. Przy teście jednostronnym (Julia<Java) hipoteza zerowa odrzucana była przy bardzo małej wartości p, co świadczy o fakcie, iż różnice były istotnie mniejsze od 0. Wartość Statystyki dla testu Wilcoxona w zdecydowanej większości wynosiła 0, oznacza to że suma rang różnic większych od 0 jest równa 0. Jedynie dla konfiguracji 1 dla instancji Ftv47 suma rang różnic dodatnich wynosiła 1. Różnica taka uzyskana została tylko dla pierwszego ziarna z jakim uruchomiano 1 konfigurację.

Tylko w przypadku ostatniej konfiguracji dla pliku Ftv170 dane posiadały cechy rozkładu normalnego. Test t-studenta sparowany obustronny wskazywał że średnia różnic istotnie różni się od 0. Test jednostronny (Julia>Java) nie odrzucił hipotezy zerowej mówiącej, że czas w Javie jest większy. Natomiast test jednostronny (Julia<Java) odrzucił hipotezę zerową, co oznacza, iż średnia różnic jest istotnie mniejsza od zera. Wartość statystyki t była silnie ujemna, co potwierdza mniejsze czasy w Julii.

Wykresy przedstawione na rysunkach 3.5 oraz 3.10 wskazują na początkową niską różnicę czasu dla 1 konfiguracji zarówno dla instancji Ftv47 jak i Ftv170. Spadek ten występuje tylko przy konfiguracji 1, co może świadczyć iż spowodowany jest różnicami w kompilacji kodu w Julii i Javie. W Julii kod kompilowany jest przy pierwszym wywołaniu funkcji do kodu maszynowego, w Javie kod jest wcześniej kompilowany do kodu bajtowego. Po uruchomieniu kod w Javie jest od razu interpretowany, a kompilacja do kodu maszynowego i optymalizacja przebiega w tle. Z tego powodu zaobserwowano dodatnią różnice czasów dla konfiguracji 1 dla instancji Ftv47. Podobne obserwacje zostały opisane w badaniach NASA dotyczących symulacji lotów [22]. Zwrócono w nich uwagę na wzrost wydajności Julii po kilku uruchomieniach funkcji.

Dla podanych instancji można zaobserwować, iż dla Ftv47 posiadającej 47 wierzchołków procentowa różnica była większa niż dla Ftv170 posiadającej 170 wierzchołków, dla Ftv47 mediany procentowej przewagi Julii od 15,45% do 21,79% natomiast dla Ftv170 od 40,75% do 44,34%. Również w obrębie instancji zauważalna jest większa różnica na korzyść Julii przy populacji o rozmiarze 1000. Przy rozmiarze 2000 przewaga Julii spadała. Dla Ftv47 różnica ta wynosiła $\sim$ 6 punktów procentowych, natomiast w przypadku Ftv170 $\sim$ 4 punktów procentowych.

Przeprowadzone badania potwierdziły hipotezę badawczą. Implementacja w Julii pozwoliła osiągnąć mniejszy czas wykonania niż w Javie. Jednocześnie obserwacje z etapu implementacji wskazują iż uzyskanie wysokiej wydajności Julii wymaga określonych praktyk programistycznych. Jest to przykład opisanego wcześniej Problemu 1,5 Języków, w którym język wymaga stosowania określonych praktyk i zachowań by uzyskać wydajność.

Wyniki testów statystycznych dotyczących szczytowej zajętości pamięci jednoznacznie wskazały implementację w Julii jako używającą mniej pamięci. Dane tylko dla jednej konfiguracji dla instancji Ftv170 wykazały cechy rozkładu normalnego.

Dla danych nie posiadających cech rozkładu normalnego test obustronny Wilcoxona wskazywał, iż mediana różnic istotnie jest różna do 0. Testy jednostronne (Julia>Java) nie potwierdziły hipotezy alternatywnej mówiącej iż mediana różnic jest większa od 0. Natomiast testy (Java<Julia) z dużą istotnością odrzucały hipotezę zerową mówiącą, że mediana różnic jest większa równa 0.

W przypadku 3 konfiguracji dla pliku Ftv170 przeprowadzono t-testy sparowane. Test obustronny z dużą istotnością odrzucił hipotezę zerową mówiącą iż średnia różnic jest równa 0. Test jednostronny (Julia>Java) odrzucił hipotezę alternatywną mówiącą iż średnia różnic jest większa od 0. Natomiast test (Julia<Java) z dużą istotnością odrzucił hipotezę zerową mówiącą iż średnia różnic jest większa równa zero.

Wszystkie przeprowadzone testy jednoznacznie wskazały Julię jako charakteryzującą się mniejszym szczytowym zużyciem pamięci. Podobnie jak w przypadku wydajności czasowej zaobserwować można zmianę różnicy w zależności od instancji. Dla mniejszej instancji różnica na korzyść Julii była większa. Przy instancji Ftv47 mediana procentowej przewagi Julii dla szczytowego zużycia pamięci była w zakresie od 73,70% do 93,37%, natomiast w przypadku Ftv170 od 55,42% do 81,76%. Podobnie jak w przypadku czasu dla populacji o rozmiarze 1000 różnica na korzyść Julii była większa niż w przypadku populacji 2000. Dla Ftv47 wynosiła ona $\sim$ 20 punktów procentowych, a w przypadku Ftv170 $\sim$ 20 -25 punktów procentowych.

W odpowiedzi na postawione pytanie badawcze, przeprowadzona analiza wskazuje, że implementacja w Julii charakteryzuje się mniejszym szczytowym zużyciem pamięci. Należy jednak podkreślić iż metody pomiaru w Julii i Javie ze względu na ograniczenia obu języków różnią się od siebie. W Julii pomiar obejmował rozmiar żywych obiektów po ostatnim uruchomieniu oczyszczania pamięci oraz obiekty zaalokowane od tego czasu, natomiast w Javie mierzono całą zajętą część sterty wraz z obiektami nieosiągalnymi, które nie zostały jeszcze usunięte. Z powyższych powodów wyniki należy interpretować jako jakościową wskazówkę, iż implementacja w Julii wykazuje korzystniejsze zużycie pamięci, jednocześnie pamiętając o jednostronnym błędzie pomiarowym na niekorzyść Javy powodowanym metodyką pomiarów.

Dodatkowo w przypadku Julii zaobserwowano znaczne błędy przypadkowe, wynikające z charakterystyki pracy odśmiecania pamięci. Zwiększenie częstotliwości próbkowania powyżej obranych 100ms pozwoliłoby zminimalizować te odchylenia i uzyskać bardziej precyzyjne wyniki.

W dalszych badaniach warto rozszerzyć metodę badawczą o alternatywne metody pomiaru pamięci, które pozwoliłyby na dokładną ocenę szczytowej zajętości pamięci. Zmiana struktury całej implementacji algorytmu i nastawienie na uzyskanie maksymalnej wydajności w Julii mogłaby pozwolić na uzyskanie jeszcze większej przewagi Julii. W celu dokładniejszej analizy wydajności języków pomiar wydajności ograniczony do pojedynczych funkcji mógłby wskazać dokładne różnice między obiema implementacjami.

Literatura

[1] *Java SE Documentation*. Dostęp: 30 listopada 2025.

[2] Julia micro-benchmarks. Dostęp: 30 listopada 2025.

[3] Julia — plik ‘base/timing.jl‘. Dostęp: 30 listopada 2025.

[4] Julian language. Dostęp: 29 listopada 2025.

[5] Performance tips. Dostęp: 30 listopada 2025.

[6] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, L. Zoubritzky. Julia: Dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–23, List. 2018.

[7] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[8] D. Blackman, S. Vigna. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software*, 47(4), Gru. 2021.

[9] P. Bruce, A. Bruce, P. Gedeck. *Statystyka praktyczna w data science: 50 kluczowych zagadnień w językach R i Python*. Helion, Gliwice, wydanie 2, 2021.

[10] Z. M. David B. Fogel. *Jak to rozwiązać czyli nowoczesna heurystyka*. WNT, 2006.

[11] L. E. de Souza Amorim, Y. Lin, S. M. Blackburn, D. Netto, G. Baraldi, N. Daly, A. L. Hosking, K. Pamnany, O. Smith. Reconsidering garbage collection in julia: A practitioner report. *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM '25)*, strony 1–12, Seoul, Republic of Korea, Czerw. 2025. ACM.

[12] T. El-Ghazali. *Metaheuristics: From Design to Implementation*. Willey, 2009.

[13] S. H. *Java. Kompendium programisty*. Helion, wydanie 12, 2023.

[14] A. Imam, U. Mohammed, C. M. Abanyam. On consistency and limitation of paired t-test, sign and wilcoxon sign rank test. *OSR Journal of Mathematics (IOSR-JM)*, 10(1), Luty 2014.

[15] G. L. S. Jr., D. Lea, C. H. Flood. Fast splittable pseudorandom number generators. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, strony 453–472. ACM, Paz. 2014.

[16] D. E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison-Wesley, wydanie 3, 1997.

[17] S. Luke. *Essentials of Metaheuristics*. Lulu, 2013. http://cs.gmu.edu/∼sean/book/metaheuristics/.

[18] Oracle. *Java SE 8 Runtime Class Documentation*. Dostęp: 30 listopada 2025.

[19] Punit. Java bytecode compilation. *IJRDO - Journal of Computer Science and Engineering*, 1(12):42–48, Gru. 2015.

[20] G. Reinelt. TSPLIB95 – a library of sample instances for the tsp. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Dostęp: 11 listopada 2025.

[21] G. Reinelt. Tsplib95. Raport instytutowy, Universität Heidelberg, Institut für Angewandte Mathematik, Im Neuenheimer Feld 294, D-69120 Heidelberg, 1995. Technical Report.

[22] R. Sells. Julia programming language benchmark using a flight simulation. *Proceedings of the 2020 IEEE Aerospace Conference*, Big Sky, MT, USA, Mar. 2020.

[23] W. Zhao, S. M. Blackburn. Deconstructing the garbage-first collector. 2020.

Spis rysunków

3.1. Wizualizacja wag krawędzi grafu Ftv47 38
3.2. Wizualizacja wag krawędzi grafu Ftv170 38
3.3. Porównanie czasów wykonania dla konfiguracji 1,2,3 - Ftv47 46
3.4. Różnice czasów wykonania dla konfiguracji 1,2,3 - Ftv47 47
3.5. Procentowe różnice czasów wykonania dla konfiguracji 1,2,3 w kolejnych iteracjach - Ftv47 47
3.6. Porównanie szczytowej zajętości pamięci dla konfiguracji 1,2,3 - Ftv47 48
3.7. Różnice szczytowej zajętości pamięci dla konfiguracji 1,2,3 - Ftv47 48
3.8. Porównanie czasów wykonania dla konfiguracji 1,2,3,4 - Ftv170 54
3.9. Różnice czasów wykonania dla konfiguracji 1,2,3,4 - Ftv170 55
3.10. Procentowe różnice czasów wykonania dla konfiguracji 1,2,3,4 w kolejnych iteracjach - Ftv170 55
3.11. Porównanie szczytowej zajętości pamięci dla konfiguracji 1,2,3,4 - Ftv170 56
3.12. Różnice szczytowej zajętości pamięci dla konfiguracji 1,2,3,4 - Ftv170 56

Spis tabel

3.1. Konfiguracja sprzętowa użyta w eksperymentach 35
3.2. Zastosowane operatory i metody w algorytmie genetycznym 35
3.3. Hipotezy dla testów statystycznych parowanych 37
3.4. Użyte instancje problemu ATSP 37
3.5. Parametry algorytmu genetycznego użyte w eksperymentach 39
3.6. Mediany błędu dla pliku Ftv47 39
3.7. Wybrane konfiguracje dla pliku Ftv47 40
3.8. Wyniki testów statystycznych dla wybranych konfiguracji - Ftv47 40
3.9. Mediany błędu dla pliku Ftv170 40
3.10. Wybrane konfiguracje dla pliku Ftv170 41
3.11. Wyniki testów statystycznych dla wybranych konfiguracji - Ftv170 42
3.12. Wybrane konfiguracje dla pliku Ftv47 - II etap badań 42
3.13. Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 1 42
3.14. Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 1 43
3.15. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 1 . . . 43
3.16. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1 43
3.17. Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 2 44
3.18. Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 2 44
3.19. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 2 . . . 44
3.20. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2 44
3.21. Statystyki opisowe czasu wykonania - plik Ftv47 - konfiguracja 3 45
3.22. Wyniki testów statystycznych dla czasów - plik Ftv47 - konfiguracja 3 45
3.23. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv47 - konfiguracja 3 . . . 45
3.24. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3 46
3.25. Wybrane konfiguracje dla pliku Ftv170 - II etap badań 49
3.26. Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 1 49
3.27. Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 1 49
3.28. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1 . . . 50
3.29. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 1 50
3.30. Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 2 50
3.31. Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 2 51
3.32. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2 . . . 51
3.33. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 2 51
3.34. Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 3 52
3.35. Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 3 52

3.36. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3 . . . 52
3.37. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 3 . . . 52
3.38. Statystyki opisowe czasu wykonania - plik Ftv170 - konfiguracja 4 . . . 53
3.39. Wyniki testów statystycznych dla czasów - plik Ftv170 - konfiguracja 4 . . . 53
3.40. Statystyki opisowe szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 4 . . . 53
3.41. Wyniki testów statystycznych dla szczytowej zajętości pamięci - plik Ftv170 - konfiguracja 4 . . . 54

Spis listingów

1.1. Algorytm Genetyczny (GA) – Luke, S. *Essentials of Metaheuristics*, 2012, p. 35 8
2.1. SplitMix64 zaimplementowany w Javie Źródło: opracowanie własne na podstawie [15] . . . 21
2.2. SplitMix64 zaimplementowany w Julii Źródło: opracowanie własne na podstawie [15] 22
2.3. Fragment Xoshiro256++ zaimplementowanego w Javie Źródło: opracowanie własne na podstawie [8] . . . 22
2.4. Fragment Xoshiro256++ zaimplementowanego w Julii Źródło: opracowanie własne na podstawie [8] . . . 23
2.5. Fragment funkcji generujący liczby 32 bitowe zaimplementowanej w Javie Źródło: opracowanie własne . . . 23
2.6. Fragment funkcji generujący liczby 32 bitowe zaimplementowanej w Julii Źródło: opracowanie własne . . . 23
2.7. Fragment funkcji generujący liczby 64 bitowe zmiennoprzecinkowe zaimplementowanej w Julii Źródło: opracowanie własne . . . 23
2.8. Fragment funkcji generujący liczby 64 bitowe zmiennoprzecinkowe zaimplementowanej w Javie Źródło: opracowanie własne . . . 24
2.9. Fragment definicji struktury Path w Julii . . . 24
2.10. Fragment definicji klasy Path w Javie . . . 24
2.11. Fragment konstruktora generującego ścieżki w Julii . . . 25
2.12. Fragment konstruktora generującego ścieżki w Javie . . . 25
2.13. Fragment konstruktora generującego ścieżki w Julii z użyciem algorytmu NN . . . 25
2.14. Fragment konstruktora generującego ścieżki w Javie z użyciem algorytmu NN . . . 26
2.15. Funkcja mieszająca zaimplementowana w Julii wykorzystująca algorytm Fisher–Yates 26
2.16. Funkcja mieszająca zaimplementowana w Javie wykorzystująca algorytm Fisher–Yates . . . 27
2.17. Funkcja implementująca selekcje turniejową w Julii . . . 27
2.18. Funkcja implementująca selekcje turniejową w Javie . . . 28
2.19. Funkcja implementująca selekcję metodą ruletki w Julii . . . 28
2.20. Funkcja implementująca selekcje metodą ruletki w Javie . . . 29
2.21. Fragment funkcji implementującej krzyżowanie OX w Julii . . . 29
2.22. Fragment funkcji implementującej krzyżowanie OX w Javie . . . 30
2.23. Fragment funkcji implementującej krzyżowanie dwupunktowe w Julii . . . 30
2.24. Fragment funkcji implementującej krzyżowanie dwupunktowe w Javie . . . 31
2.25. Fragment funkcji implementującej mutacje przez zamianę w Julii . . . 31
2.26. Fragment funkcji implementującej mutacje przez zamianę w Javie . . . 32
2.27. Fragment funkcji implementującej mutacje przez inwersję w Julii . . . 32
2.28. Fragment funkcji implementującej mutacje przez inwersję w Javie . . . 32
2.29. Fragment macierzy ftv47.atsp z bazy TSPLIB . . . 33