Projektowanie efektywnych algorytmów
Etap (3)

Asymetryczny problem komiwojażera

Jakub Kapłonek 263908

9 stycznia 2024

Spis treści

1 Wstęp Teoretyczny **1**
1.1 Opis Problemu TSP . . . 1
1.2 Cel projektu . . . 1

2 Algorytm genetyczny dla problemu komiwojażera **2**
2.1 Metody selekcji . . . 4
2.1.1 Tournament selection . . . 4
2.2 Metody krzyżowania . . . 5
2.2.1 Ordered crossover . . . 5
2.2.2 Cycle crossover . . . 8
2.3 Metody mutacji . . . 10
2.3.1 Swap mutation . . . 10
2.3.2 Inversion mutation . . . 12

3 Wyniki eksperymentów **14**
3.1 Tabele z wynikami i wykresy . . . 14
3.1.1 ftv47.atsp . . . 14
3.1.2 ftv170.atsp . . . 14
3.1.3 rbg403.atsp . . . 14
3.2 Wybór najlepszej metody . . . 14
3.3 Wybór najlepszej populacji . . . 16
3.4 Wpływ współczynników . . . 16
3.4.1 Współczynnik krzyżowania . . . 16
3.4.2 Współczynnik mutacji . . . 17

4 Wnioski **17**

1 Wstęp Teoretyczny

1.1 Opis Problemu TSP

Problem Komiwojażera (TSP) polega na znalezieniu najkrótszej ścieżki przechodzącej przez wszystkie miasta dokładnie raz i wracającej do punktu początkowego. Jest to problem optymalizacyjny, który ma zastosowanie w wielu dziedzinach, takich jak logistyka czy trasowanie.

1.2 Cel projektu

Celem tego projektu jest wprowadzenie oraz ocena metody heurystycznej: Algorytm genetyczny, mającej zastosowanie w efektywnym rozwiązaniu problemu Komiwojażera. Badania te mają na celu zanalizowanie skuteczności i efektywności tego algorytmu w kontekście optymalizacji tras podróży.

2 Algorytm genetyczny dla problemu komiwojażera

Algorytm genetyczny dla problemu komiwojażera zaczyna się od stworzenia losowej populacji tras komiwojażera. Każda trasa reprezentuje jedno potencjalne rozwiązanie problemu. Następnie ocenia się koszt każdej trasy.

W kolejnych krokach algorytmu trasy są wybierane do reprodukcji. Następnie pary rodziców są krzyżowane, aby utworzyć potomstwo, reprezentujące nowe trasy. Czasami potomstwo podlega mutacji, co oznacza losową zmianę niektórych elementów trasy.

Po ocenie długości tras potomstwa, wybierane są najlepsze trasy. Ten proces powtarza się, aż spełniony zostanie warunek stopu, w naszym projekcie jest to ograniczony czas działania algorytmu.

Na końcu algorytmu wybierana jest trasa z najlepszym przystosowaniem jako ostateczne rozwiązanie problemu komiwojażera. Algorytm genetyczny odzwierciedla procesy ewolucyjne, gdzie lepiej przystosowane rozwiązania mają większą szansę na przetrwanie i ewoluowanie w kierunku optymalnego rozwiązania.

```
public class Genetic implements Algorithm {
    ...
    @Override
    public AlgorithmResult runAlgorithm(
        Matrix matrix
    ) throws IncorrectDataException {
        run = true;
        distanceMatrix = matrix.getDistanceMatrix();

        // create population
        List<Tour> population = createPopulation();
        bestSolution = getBestTour(population);

        //mutate and crossover
        while (run) {
            population = createNewPopulation(population);
            population = crossover.crossover(population);
            population = mutation.mutate(population);
            recalculateDistances(population);

            Tour bestTour = getBestTour(population);
            if(bestSolution.getCost() > bestTour.getCost()) {
                bestSolution = bestTour.clone();
                if (observer != null)
                    observer.invoke(bestSolution.getTour(), bestSolution.getCost());
            }
        }

        return new AlgorithmResult(
            matrix.size,
            Arrays.stream(bestSolution.getTour().toArray(new Integer[0]))
                .mapToInt(Integer::intValue).toArray(),
            bestSolution.getCost(),
            getName()
        );
    }
    ...
}
```

Rysunek 1: Implementacja algorytmu genetycznego w kodzie programu

```
private List<Tour> createPopulation() {
    List<Tour> population = new ArrayList<>();
    for (int i = 0; i < populationSize; i++) {
        List<Integer> chromosome = new ArrayList<>();
        for (int j = 0; j < distanceMatrix.length; j++) {
            chromosome.add(j);
        }
        Collections.shuffle(chromosome);
        population.add(
            new Tour(chromosome, calculateTotalDistance(chromosome))
        );
    }

    return population;
}

private List<Tour> createNewPopulation(List<Tour> population) {
    List<Tour> newPopulation = new ArrayList<>();
    for(int i = 0; i < population.size(); i++) {
        newPopulation.add(selection.select(population));
    }

    return newPopulation;
}
```

Rysunek 2: Metody pomocnicze createPopulation i createNewPopulation

2.1 Metody selekcji

2.1.1 Tournament selection

Metoda selekcji Tournament Selection to jedna z popularnych technik wyboru osobników do reprodukcji w algorytmach genetycznych. Ta metoda opiera się na zorganizowaniu turnieju, w którym losowo wybrane osobniki konkurują ze sobą, a zwycięzca turnieju jest wybierany do reprodukcji.

Poniżej opisano kroki działania Tournament Selection:

1. Rozmiar turnieju:
   Określa się stały rozmiar turnieju (np. 2, 3, lub więcej). Jest to liczba osobników, które będą rywalizować ze sobą.

2. Losowy wybór uczestników turnieju:
   Wybiera się losowo k osobników (gdzie k to rozmiar turnieju) spośród populacji. Mogą to być dowolne osobniki, bez względu na ich wartość przystosowania.

3. Porównanie uczestników:
   Porównuje się osobniki biorące udział w turnieju na podstawie ich wartości przystosowania. Wygrywa ten, który ma lepszą wartość przystosowania.

4. Wybór zwycięzcy:
   Osobnik z najlepszym przystosowaniem w turnieju jest wybierany jako zwycięzca.

5. Powtarzanie procesu:
   Proces turnieju jest powtarzany, aby wybrać wymaganą liczbę osobników do reprodukcji.

```
package TSP.algorithms.Genetic;

import java.util.List;
import java.util.Random;

public class TournamentSelection implements Selection {
    private Random r = new Random();

    @Override
    public Tour select(List<Tour> population) {
        Tour specie1 = population.get(r.nextInt(population.size()));
        Tour specie2 = population.get(r.nextInt(population.size()));

        while(specie1 == specie2) {
            specie2 = population.get(r.nextInt(population.size()));
        }

        if(specie1.getCost() < specie2.getCost()) {
            return specie1;
        }

        return specie2;
    }
}
```

Rysunek 3: Implementacja Tournament Selection w kodzie programu

2.2 Metody krzyżowania

2.2.1 Ordered crossover

Ordered Crossover (OX) to operator krzyżowania używany w algorytmach genetycznych do generowania potomstwa z dwóch rodziców. Działa on na chromosomach, które reprezentują kolejność elementów w permutacji, jakimi są miasta w przypadku problemu komiwojażera.

Poniżej opisano kroki działania Ordered Crossover:

1. Wybór fragmentu:
   Losowo wybiera się pewien fragment chromosomu rodzica 1. Ten fragment będzie przeniesiony bez zmian do potomstwa.

2. Przeniesienie fragmentu:
   Przenosi się wybrany fragment z rodzica 1 na potomka.

3. Uzupełnienie potomstwa z rodzica 2:
   Uzupełnia się pozostałe pozycje w potomstwie, przechodząc przez rodzica 2. Jeśli elementy z rodzica 2 nie są już obecne w potomstwie, dodaje się je w kolejności, w jakiej występują w rodzicu 2.

4. Zapobieganie powtórzeniom:
   Jeżeli w wyniku kroku 3 pojawiają się powtórzenia w potomstwie, brakujące elementy uzupełnia się, omijając już obecne w potomstwie elementy z rodzica 1.

Poniżej znajduje się przykład, aby lepiej zrozumieć działanie Ordered Crossover. Załóżmy, że mamy rodziców:

Rodzic 1: 1 2 3 4 5 6 7 8 9
Rodzic 2: 4 7 2 1 9 5 3 8 6

Wybierzmy fragment od pozycji 3 do 6 (czyli 3, 4, 5, 6) od rodzica 1. Przenosimy ten fragment do potomstwa:

Potomek: _ _ 3 4 5 6 _ _ _

Następnie uzupełniamy resztę potomstwa korzystając z rodzica 2, omijając już obecne elementy z rodzica 1:

Potomek: 7 2 3 4 5 6 1 9 8

Tak otrzymujemy potomka w wyniku krzyżowania z użyciem Ordered Crossover. Ten operator pomaga zachować porządek elementów w permutacji, co jest ważne w problemach, gdzie kolejność ma znaczenie, takich jak problem komiwojażera.

```
public class OrderedCrossover implements Crossover {
    private double crossoverFactor;

    public OrderedCrossover(double crossoverFactor) {
        this.crossoverFactor = crossoverFactor;
    }

    @Override
    public List<Tour> crossover(List<Tour> population) {
        List<Tour> crossedPopulation = new ArrayList<>();

        for (int i = 0; i < population.size() - 1; i += 2) {
            List<Integer> parent1 = population.get(i).getTour();
            List<Integer> parent2 = population.get(i + 1).getTour();
            // Make use of crossover factor
            if(new Random().nextDouble() < crossoverFactor) {
                // Perform Ordered Crossover
                crossedPopulation.add(
                    new Tour(orderedCrossover(parent1, parent2))
                );
                crossedPopulation.add(
                    new Tour(orderedCrossover(parent2, parent1))
                );
            } else {
                crossedPopulation.add(new Tour(parent1));
                crossedPopulation.add(new Tour(parent2));
            }
        }

        // Return new population
        return crossedPopulation;
    }
.
.
.
```

Rysunek 4: Implementacja Ordered Crossover w kodzie programu

```
.
.
.
private List<Integer> orderedCrossover(
        List<Integer> parent1,
        List<Integer> parent2
    ) {
        int tourSize = parent1.size();
        int[] child = new int[tourSize];
        int startPos = getRandomPosition(tourSize);
        int endPos = getRandomPosition(tourSize);

        // Ensure startPos is less than endPos
        if (startPos > endPos) {
            int temp = startPos;
            startPos = endPos;
            endPos = temp;
        }

        // Copy a segment from parent1 to child
        for (int i = startPos; i <= endPos; i++) {
            child[i] = parent1.get(i);
        }

        // Fill the remaining positions with genes from parent2
        int parent2Index = 0;
        for (int i = 0; i < tourSize; i++) {
            if (!contains(child, parent2.get(i))) {
                while (child[parent2Index] != 0) {
                    parent2Index++;
                }
                child[parent2Index++] = parent2.get(i);
            }
        }

        return new ArrayList<>(toList(child));
    }
}
```

Rysunek 5: Implementacja Ordered Crossover w kodzie programu cd.

2.2.2 Cycle crossover

Cycle Crossover (CX) to operator krzyżowania stosowany w algorytmach genetycznych do generowania potomstwa z dwóch rodziców. Podobnie jak Ordered Crossover, Cycle Crossover również jest używany w problemach permutacyjnych, takich jak problem komiwojażera. Operator ten opiera się na identyfikacji cykli w permutacjach rodziców.

Poniżej opisano kroki działania Cycle Crossover:

1. Identyfikacja cykli:
   Zaczynając od pierwszego elementu rodzica 1, tworzy się cykl, przechodząc z jednego rodzica na drugiego według odpowiadających im pozycji. Cykl jest kontynuowany, aż wróci się do pierwszego elementu.

2. Przeniesienie cyklu:
   Przenosi się cykl z jednego rodzica do potomka.

3. Uzupełnienie potomstwa:
   Uzupełnia się pozostałe pozycje w potomstwie, używając elementów z drugiego rodzica, które nie są jeszcze obecne w potomstwie. Kolejność uzupełniania zachowuje się zgodnie z cyklem.

4. Kontynuacja cykli:
   Powtarza się kroki 1-3, aż wszystkie elementy są umieszczone w potomstwie.

Poniżej znajduje się przykład działania Cycle Crossover. Załóżmy, że mamy rodziców:

Rodzic 1: 1 2 3 4 5 6 7 8 9
Rodzic 2: 9 3 7 8 2 6 5 1 4

Identyfikujemy cykl, zaczynając od pierwszego elementu (1) rodzica 1:

1 -> 9 -> 4 -> 8 -> 1

(zamknięcie cyklu wraca do 1).

Przenosimy ten cykl do potomka:

Potomek: 1 _ _ 4 _ _ _ 8 9

Uzupełniamy pozostałe pozycje używając elementów z rodzica 2, które jeszcze nie są obecne w potomstwie:

Potomek: 1 3 7 4 2 6 5 8 9

Tak otrzymujemy potomka w wyniku krzyżowania z użyciem Cycle Crossover. Ten operator pomaga zachować pewne właściwości permutacji, a jego działanie opiera się na identyfikacji i przenoszeniu cykli między rodzicami.

```
public class CycleCrossover implements Crossover {
    @Override
    public List<Tour> crossover(List<Tour> population) {
        List<Tour> newTours = new ArrayList<>();

        for (int i = 0; i < population.size() - 1; i += 2) {
            Tour parent1 = population.get(i);
            Tour parent2 = population.get(i + 1);

            // Check if crossover should be applied
            if (Math.random() < crossoverFactor) {
                // Perform Cycle Crossover
                Tour child1 = cycleCrossover(parent1, parent2);
                Tour child2 = cycleCrossover(parent2, parent1);

                newTours.add(child1);
                newTours.add(child2);
            } else {
                // If crossover is not applied,
                // simply add the parents to the new population
                newTours.add(parent1);
                newTours.add(parent2);
            }
        }

        return newTours;
    }

    private Tour cycleCrossover(Tour parent1, Tour parent2) {
        int tourSize = parent1.getTour().size();
        int[] child = new int[tourSize];
        Arrays.fill(child, -1);

        int index = r.nextInt(tourSize);

        do {
            int currentCity = parent1.getTour().get(index);
            child[index] = currentCity;
            index = parent2.getTour().indexOf(currentCity);
        } while (child[index] == -1);

        for (int i = 0; i < tourSize; i++) {
            if (child[i] == -1) {
                child[i] = parent2.getTour().get(i);
            }
        }

        return new Tour(toList(child));
    }
}
```

Rysunek 6: Implementacja Cycle Crossover w kodzie programu

2.3 Metody mutacji

2.3.1 Swap mutation

Mutacja Swap jest jednym z operatorów mutacji używanych w algorytmach genetycznych do eksploracji przestrzeni rozwiązań poprzez zamianę miejscami dwóch losowo wybranych elementów w chromosomie. W kontekście problemu permutacyjnego, taki jak problem komiwojażera, elementy te reprezentują kolejność miast na trasie.

Poniżej opisano kroki algorytmu mutacji Swap:

1. Wybór dwóch pozycji:
   Losowo wybiera się dwie różne pozycje w chromosomie, czyli dwa różne miasta w przypadku problemu komiwojażera.

2. Zamiana miejscami elementów:
   Zamienia się miejscami elementy na wybranych pozycjach w chromosomie.

3. Aktualizacja chromosomu:
   Zaktualizowany chromosom reprezentuje nowe potomstwo uzyskane w wyniku mutacji.

Poniżej znajduje się prosty przykład mutacji Swap na chromosomie reprezentującym trasę komiwojażera:

Przed mutacją:

Chromosom: 1 2 3 4 5 6 7

Po mutacji:

Chromosom: 1 6 3 4 5 2 7

```
public class SwapMutation implements Mutation {

    private double mutationFactor;

    public SwapMutation(double mutationFactor) {
        this.mutationFactor = mutationFactor;
    }

    @Override
    public List<Tour> mutate(List<Tour> population) {
        List<Tour> mutatedPopulation = new ArrayList<>(population);
        for (Tour tour : mutatedPopulation) {
            if (Math.random() < mutationFactor) {
                int tourSize = tour.getTour().size();
                int pos1 = getRandomPosition(tourSize);
                int pos2 = getRandomPosition(tourSize);

                // Ensure pos1 is different from pos2
                while (pos1 == pos2) {
                    pos2 = getRandomPosition(tourSize);
                }

                // Swap the cities at pos1 and pos2
                Collections.swap(tour.getTour(), pos1, pos2);
            }
        }

        return mutatedPopulation;
    }

    private static int getRandomPosition(int maxValue) {
        return new Random().nextInt(maxValue);
    }
}
```

Rysunek 7: Implementacja Swap Mutation w kodzie programu

2.3.2 Inversion mutation

Inversion Mutation to operator mutacji stosowany w algorytmach genetycznych, zwłaszcza w problemach permutacyjnych, takich jak problem komiwojażera. Ten operator mutacji polega na odwróceniu kolejności pewnego fragmentu chromosomu. W przypadku problemu komiwojażera, fragment ten reprezentuje pewną sekwencję miast na trasie.

Poniżej przedstawiam kroki działania algorytmu Inversion Mutation:

1. Wybór fragmentu do odwrócenia:
   Losowo wybiera się dwa różne indeksy w chromosomie, oznaczające początek i koniec fragmentu do odwrócenia.

2. Odwrócenie fragmentu:
   Odwraca się kolejność elementów w wybranym fragmencie chromosomu.

3. Aktualizacja chromosomu:
   Zaktualizowany chromosom reprezentuje nowe potomstwo uzyskane w wyniku mutacji.

Poniżej znajduje się prosty przykład mutacji Inversion na chromosomie reprezentującym trasę komiwojażera:
Przed mutacją:

Chromosom: 1 2 3 4 5 6 7 8

Po mutacji:

Chromosom: 1 2 6 5 4 3 7 8

Operacja Inversion wprowadza zmianę w chromosomie poprzez odwrócenie pewnego fragmentu, co pomaga w zwiększeniu różnorodności populacji.

```
public class InversionMutation implements Mutation {
    private Random r = new Random();

    private double mutationFactor;

    public InversionMutation(double mutationFactor) {
        this.mutationFactor = mutationFactor;
    }

    @Override
    public List<Tour> mutate(List<Tour> population) {
        for (Tour tour : population) {
            if (r.nextDouble() < mutationFactor) {
                inversionMutate(tour);
            }
        }

        return population;
    }

    private void inversionMutate(Tour tour) {
        int tourSize = tour.getTour().size();
        int startPos = getRandomPosition(tourSize);
        int endPos = getRandomPosition(tourSize);

        // Ensure startPos is less than endPos
        if (startPos > endPos) {
            int temp = startPos;
            startPos = endPos;
            endPos = temp;
        }

        // Invert the order of cities between startPos and endPos
        while (startPos < endPos) {
            Collections.swap(tour.getTour(), startPos, endPos);
            startPos++;
            endPos--;
        }
    }

    private int getRandomPosition(int maxValue) {
        return r.nextInt(maxValue);
    }
}
```

Rysunek 8: Implementacja Inversion Mutation w kodzie programu

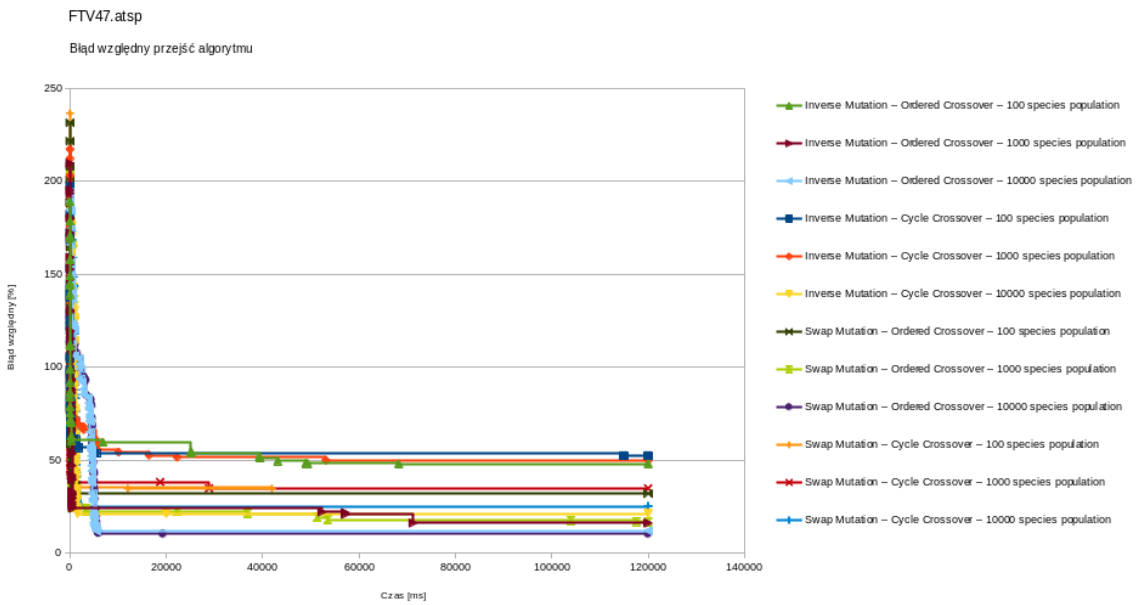
3 Wyniki eksperymentów

3.1 Tabele z wynikami i wykresy

3.1.1 ftv47.atsp

	100	1000	10000
Swap Mutation – Ordered Crossover	31,981981981982	16,8355855855856	10,3040540540541
Inverse Mutation – Ordered Crossover	48,0855855855856	16,1599099099099	11,9932432432432
Inverse Mutation – Cycle Crossover	52,3085585585586	49,8873873873874	21,1711711711712
Swap Mutation – Cycle Crossover	34,6846846846847	34,7972972972973	25,2252252252252

Tabela 1: Wyniki przejść algorytmu dla metod krzyżowania i mutacji, oraz wielkości populacji dla pliku ftv47.atsp - błąd względny [%]



Rysunek 9: Wykres błędu względnego w funkcji czasu - ftv47.atsp

3.1.2 ftv170.atsp

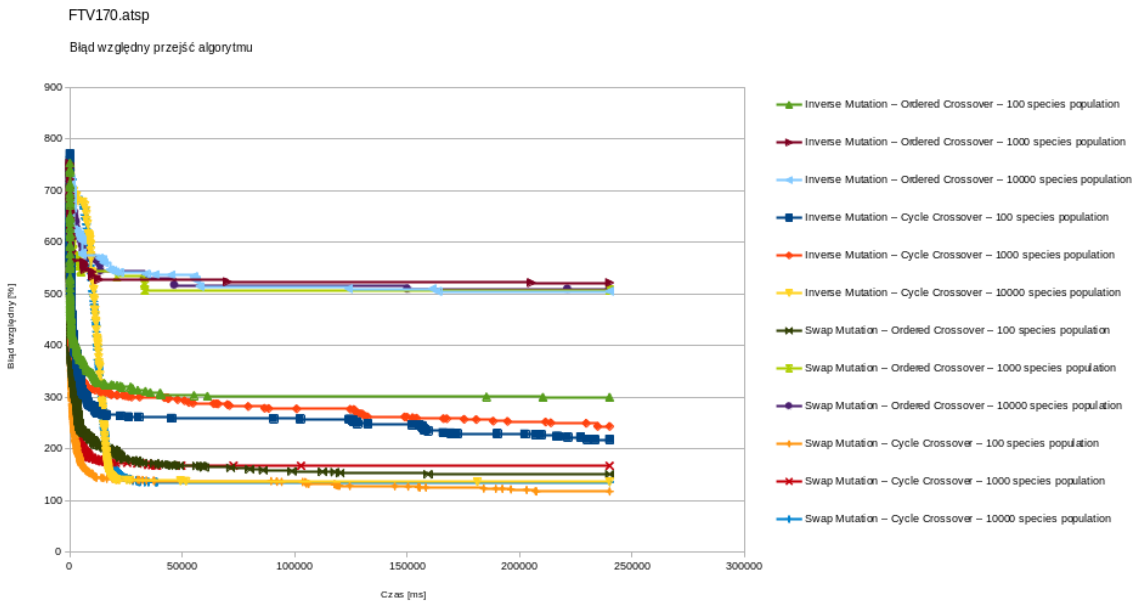
	100	1000	10000
Swap Mutation – Cycle Crossover	117,531760435572	166,969147005445	135,607985480944
Swap Mutation – Ordered Crossover	150,562613430127	507,150635208711	509,219600725953
Inverse Mutation – Cycle Crossover	217,894736842105	243,303085299456	135,862068965517
Inverse Mutation – Ordered Crossover	299,927404718693	521,597096188748	504,97277676951

Tabela 2: Wyniki przejść algorytmu dla metod krzyżowania i mutacji, oraz wielkości populacji dla pliku ftv170.atsp - błąd względny [%]

3.1.3 rbg403.atsp

3.2 Wybór najlepszej metody

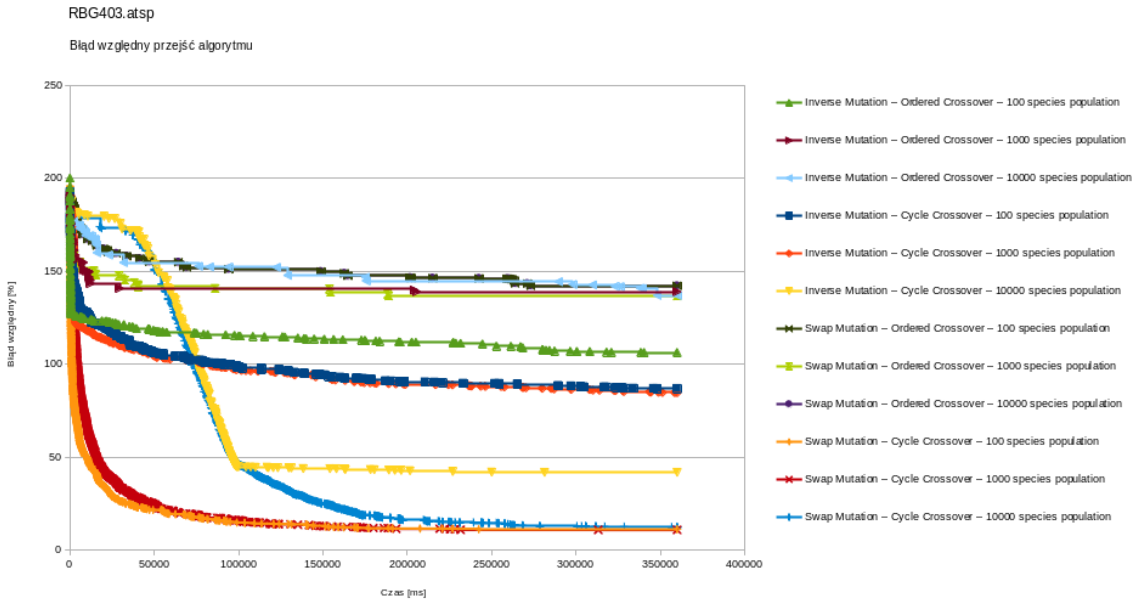
Na podstawie wyników dokonano wyboru najlepszych metod krzyżowania oraz mutacji. W każdej tabeli na samej górze znalazła się metoda z osiągniętymi najlepszymi wynikami, i im niżej - tym gorsze



Rysunek 10: Wykres błędu względnego w funkcji czasu - ftv170.atsp

	100	1000	10000
Swap Mutation – Cycle Crossover	11,237322515213	10,8722109533469	12,3732251521298
Inverse Mutation – Cycle Crossover	86,8965517241379	84,6653144016227	41,7849898580122
Inverse Mutation – Ordered Crossover	106,32860040568	138,985801217039	136,957403651116
Swap Mutation – Ordered Crossover	142,150101419878	136,632860040568	142,150101419878

Tabela 3: Wyniki przejść algorytmu dla metod krzyżowania i mutacji, oraz wielkości populacji dla pliku rbg403.atsp - błąd względny [%]



Rysunek 11: Wykres błędu względnego w funkcji czasu - rbg403.atsp

metody dla danego pliku. Na podstawie tych miejsc przydzielono metodom punkty:

Najlepszą metodą dla wszystkich plików została metoda mutacji: Swap Mutation z metodą krzy-

	ftv47.atsp	ftv170.atsp	rbg403.atsp	Suma
Swap Mutation – Cycle Crossover	1	4	4	9
Swap Mutation – Ordered Crossover	4	3	1	8
Inverse Mutation – Cycle Crossover	2	2	3	7
Inverse Mutation – Ordered Crossover	3	1	2	6

Tabela 4: Punkty oceny dla najlepiej dobranej metody

żowania: Cycle Crossover

3.3 Wybór najlepszej populacji

Podobnie jak przy wyborze metody, przydzielono odpowiednie punkty liczbą populacji w zależności od pliku

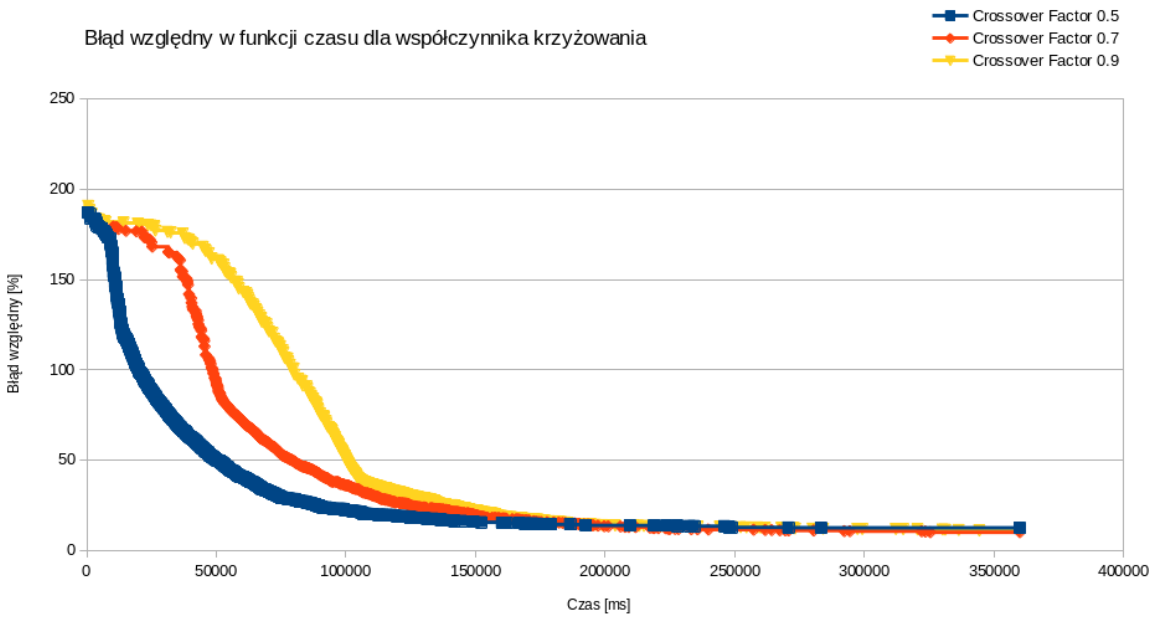
	ftv47.atsp	ftv170.atsp	rbg403.atsp	Suma
100	1	3	1	5
1000	2	1	3	6
10000	3	2	2	7

Tabela 5: Punkty oceny dla najlepiej dobranej populacji

Więc najlepszą populacją okazało się 10000 osobników, kolejne badania zostały przeprowadzone dla najlepszych metod.

3.4 Wpływ współczynników

3.4.1 Współczynnik krzyżowania

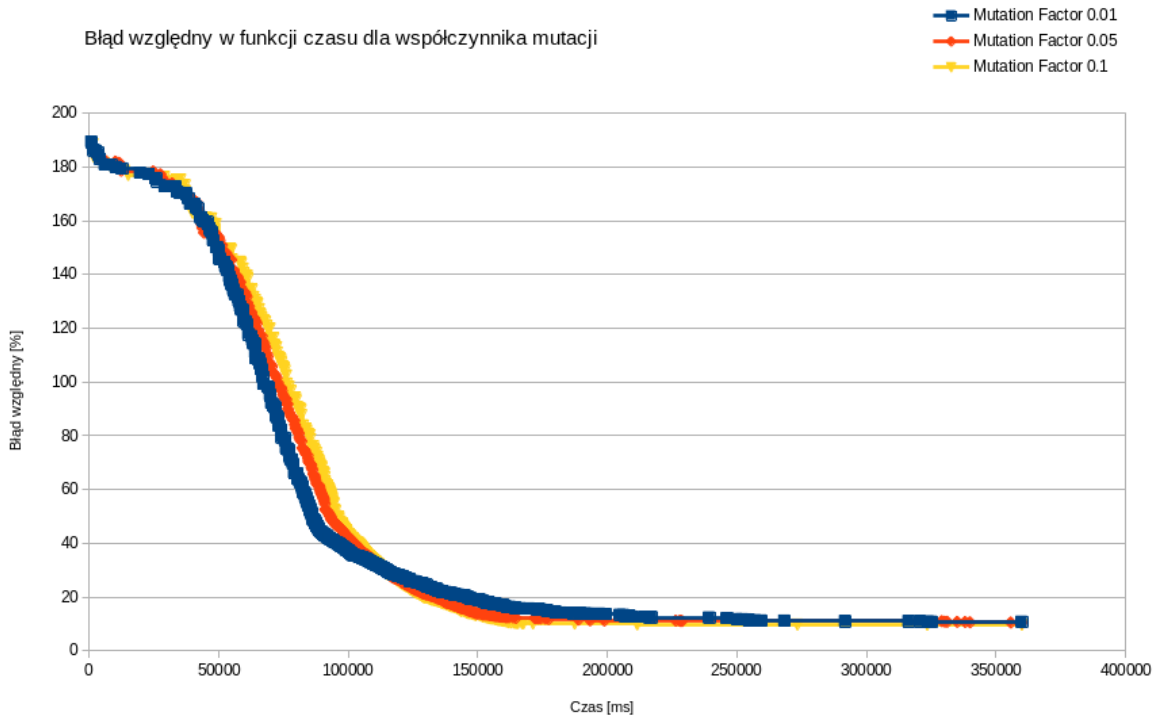


Rysunek 12: Wykres błędu względnego w funkcji czasu dla różnych współczynników krzyżowania

Na podstawie wykresu możemy zauważyć wpływ współczynnika krzyżowania na błąd względny. Rezultat końcowy jest bardzo podobny dla każdego wykresu, zmienia się natomiast jego nachylenie w znajdowaniu rozwiązania. Zauważyć można, że mniejsza wartość współczynnika krzyżowania - co

za tym idzie - mniejsza zmiana genu pomiędzy populacjami, powoduje przyśpieszenie znajdowania optimum na początku działania algorytmu i następnie im większa wartość tym ten proces jest wolniejszy. Pomimo podobnych wyników w przejściach działania algorytmu, najlepszy wynik osiągneła metoda z wyważonym współczynnikiem krzyżowania - a więc kolejno 0.7 - 2714, 0.9 - 2735, 0.5 - 2774. Pomimo subtelnych różnic w wynikach działania algorytmu, można wysnuć wniosek, że wyważenie współczynnika krzyżowania najlepiej wpływa na finalny wynik algorytmu.

3.4.2 Współczynnik mutacji



Rysunek 13: Wykres błędu względnego w funkcji czasu dla różnych współczynników mutacji

Podobnie jak dla współczynnika krzyżowania, tak dla współczynnika mutacji możemy zauważyć różnice w przebiegach wykresu. Różnice te są jednak dużo bardziej subtelne co pokazuje, że w przypadku problemu TSP, współczynnik mutacji może nie mieć dużego wpływu na wyniki działania algorytmu, w tym wypadku na podstawie wyników końcowych można stwierdzić, że wysoki współczynnik mutacji, przyczynił się do polepszenia wyniku końcowego 0.01 - 2731, 0.05 - 2724, 0.1 - 2703. Z wykresu można również odczytać, że podobnie jak w przypadku współczynnika krzyżowania, większa wartość współczynnika mutacji - a co za tym idzie, większa losowość w kolejnych populacjach, przyczynia się do zwolnienia tempa znajdowania lepszych rozwiązań.

4 Wnioski

Parametry podczas badania algorytmu genetycznego zostały z góry narzucone, pomimo tego w algorytmie nadal występuje wiele zmiennych dobranych metodami empirycznymi które mogą zoptymalizować / pogorszyć działanie algorytmu, sama metoda selekcji, oraz liczba zakwalifikowanych do turnieju chromosomów może znacznie wpłynąć na wyniki uzyskane przez algorytm. Na podstawie wyników badań metod krzyżowania oraz mutacji możemy orzec najlepszych kandydatów dla problemu TSP, jednak w tym wypadku słowo najlepsza oznaczać będzie najbardziej uniwersalna metoda, ponieważ dla różnych rodzajów problemów - różnych plików z danymi - różne metody (kombinacje metod) okazywały się najlepsze w działaniu.

Dla badań wpływu współczynników możemy orzec natomiast subtelne różnice w działaniach algorytmów, głównie skupiające się na szybkości znajdowania lepszego rozwiązania - z wyników możemy wysnuć wniosek że mniejsza losowość w generacji kolejnych populacji przyczynia się do wzrostu prędkości znajdowania optymalnego rozwiązania. Jednak należy brać pod uwagę, że metoda selekcji bierze w turnieju tylko dwa chromosomy co samo w sobie wprowadza dużą losowość do algorytmu, dlatego zmniejszenie entropii w późniejszych krokach może mieć optymalny wpływ na znajdowanie rozwiązań.

Porównując najlepsze wyniki działania algorytmów weźmiemy pod uwagę plik ftv47.atsp, najlepszy wynik został osiągnięty w obydwóch projektach w przypadku właśnie tego pliku. W projekcie 2 algorytm Tabu Search z metodą przeszukiwania - insercja losowego miasta, osiągnął we wszystkich przejściach algorytmu optimum dla tego pliku, jak i również bardzo dobre wyniki dla pozostałych plików, co jednocześnie stawia go jako bardzo optymalne podejście do rozwiązywania problemu TSP i bezsprzecznego zwycięzce w przeprowadzonych eksperymentach, algorytm genetyczny w najlepszym przypadku dla pliku ftv47.atsp z metodą krzyżowania Ordered Crossover oraz metodą mutacji Swap Mutation osiągnął błąd względny na poziomie 10%, był to jednocześnie najmniejszy błąd względny w przeprowadzonych badaniach. Z pewnością jednak, dalsza optymalizacja parametrów algorytmu genetycznego była by w stanie znacznie usprawnić działanie tego algorytmu.

Bibliografia

[1] ChatGPT.

[2] Tutorials With Gary. 2020. URL: https://www.youtube.com/watch?v=WTWYRHc_t7o.

[3] Tutorials With Gary. 2020. URL: https://www.youtube.com/watch?v=pJ45IaGzI1c.

[4] Geeks for geeks. 2023. URL: https://www.geeksforgeeks.org/tournament-selection-ga/.

[5] Wojciech Kwedlo. 2011. URL: https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf.

[6] S Prayudani i in. "Analysis Effect of Tournament Selection on Genetic Algorithm Performance in Traveling Salesman Problem (TSP)". W: *Journal of Physics: Conference Series* 1566.1 (czer. 2020), s. 012131. DOI: 10.1088/1742-6596/1566/1/012131. URL: https://dx.doi.org/10.1088/1742-6596/1566/1/012131.

[7] dr inż. Witold Beluch - Politechnika Śląska. 2012. URL: http://www.imio.polsl.pl/Dopobrania/Cw%20MH%2007%20(TSP).pdf.