Sprawozdanie - Projektowanie efektywnych algorytmów

Maksim Birszel nr indeksu 241353

12 stycznia 2020

Zadanie 3 - Implementacja i analiza efektywności algorytmu genetycznego (ewolucyjnego) dla wybranego problemu optymalizacji - Algorytmu Genetycznego

Prowadzący:
Dr. Inż
Zbigniew Buchalski

Termin zajęć:
Wtorek 9:15

1 Wstęp teoretyczny

Problem komiwojażera (ang. travelling salesman problem, TSP) – zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.
Nazwa pochodzi od typowej ilustracji problemu, przedstawiającej go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić, oraz odległość / cena podróży / czas podróży pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej / najtańszej / najszybszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w określonym punkcie. Jest on zaliczany to klasy problemów NP-trudnych.
Problem dzielimy na symetryczny oraz asymetryczny.
Symetryczny - odległości z miasta A do B są równe odległością z B do A
Asymetryczny - odległości z A do B różnią się od odległości z B do A

2 Algorytm Genetyczny

Algorytm genetyczny – rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania najlepszych rozwiązań. Sposób działania algorytmów genetycznych nieprzypadkowo przypomina zjawisko ewolucji biologicznej, ponieważ ich twórca John Henry Holland właśnie z biologii czerpał inspiracje do swoich prac. Obecnie zalicza się go do grupy algorytmów ewolucyjnych.

3 Podstawowe pojęcia

Problem definiuje środowisko, w którym istnieje pewna populacja osobników. Każdy z osobników ma przypisany pewien zbiór informacji stanowiących jego genotyp, a będących podstawą do utworzenia fenotypu. Fenotyp to zbiór cech podlegających ocenie funkcji przystosowania modelującej środowisko. Innymi słowy - genotyp opisuje proponowane rozwiązanie problemu, a funkcja przystosowania ocenia, jak dobre jest to rozwiązanie. Genotyp składa się z chromosomów, gdzie zakodowany jest fenotyp i ewentualnie pewne informacje pomocnicze dla algorytmu genetycznego. Chromosom składa się z genów.

- Funkcja oceny - Funkcja oceny to miara jakości dowolnego osobnika (fenotypu, rozwiązania) w populacji. Dla każdego osobnika jest ona

obliczana na podstawie pewnego modelu rozwiązywanego problemu. Załóżmy dla przykładu, że chcemy zaprojektować obwód elektryczny o pewnej charakterystyce. Funkcja oceny będzie premiowała rozwiązania najbardziej zbliżonej do tej charakterystyki, zbudowane z najmniejszej liczby elementów. W procesie selekcji faworyzowane będą najlepiej przystosowane osobniki. One staną się "rodzicami" dla populacji.

- Operator krzyżowania - ma za zadanie łączyć w różnych kombinacjach cechy pochodzące z różnych osobników populacji, zaś operator mutacji ma za zadanie zwiększać różnorodność tych osobników. O przynależności dowolnego algorytmu do klasy algorytmów genetycznych decyduje głównie zastosowanie operatora krzyżowania i praca z całymi populacjami osobników (idea łączenia w przypadkowy sposób genotypów nieprzypadkowo wybranych osobników). Równie ważny jest operator mutacji. Jeśli krzyżowanie traktować jako sposób eksploatacji przestrzeni rozwiązań, to mutacja jest sposobem na jej eksplorację. Może się jednak zdarzyć, że dla niektórych zagadnień jej zastosowanie nie jest krytyczne.

- Krzyżowanie - polega na połączeniu niektórych (wybierane losowo) genotypów w jeden. Kojarzenie ma sprawić, że potomek dwóch osobników rodzicielskich ma zespół cech, który jest kombinacją ich cech (może się zdarzyć, że tych najlepszych).

- Mutacja - wprowadza do genotypu losowe zmiany. Jej zadaniem jest wprowadzanie różnorodności w populacji, czyli zapobieganie (przynajmniej częściowe) przedwczesnej zbieżności algorytmu. Mutacja zachodzi z pewnym przyjętym prawdopodobieństwem - zazwyczaj rzędu 1

4 Pseudokod algorytmu Genetycznego

Działanie algorytmu przebiega następująco:

- Losowana jest pewna populacja początkowa.

- Populacja poddawana jest ocenie (selekcja). Najlepiej przystosowane osobniki biorą udział w procesie reprodukcji.

- Genotypy wybranych osobników poddawane są operatorom ewolucyjnym:

- są ze sobą kojarzone poprzez złączanie genotypów rodziców (krzyżowanie),

- przeprowadzana jest mutacja, czyli wprowadzenie drobnych losowych zmian.

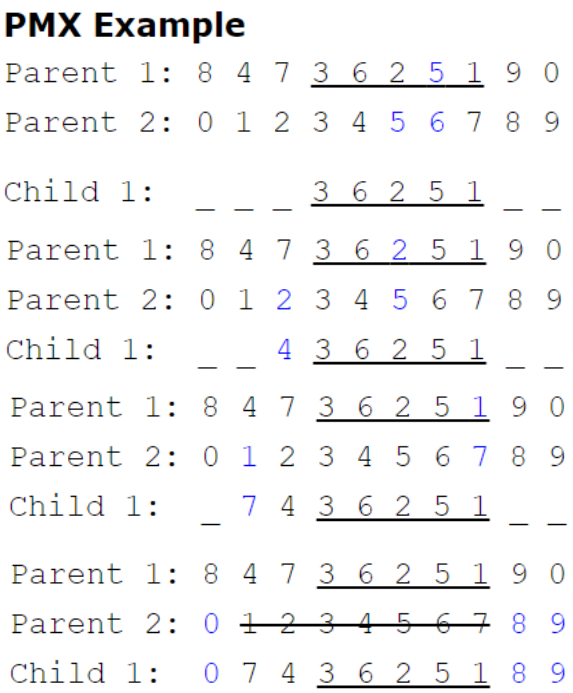
- Rodzi się drugie (kolejne) pokolenie. Aby utrzymać stałą liczbę osobników w populacji te najlepsze (według funkcji oceniającej fenotyp) są powielane, a najsłabsze usuwane. Jeżeli nie znaleziono dostatecznie dobrego rozwiązania, algorytm powraca do kroku drugiego. W przeciwnym wypadku wybieramy najlepszego osobnika z populacji - jego genotyp to uzyskany wynik.

Działanie algorytmu genetycznego obejmuje kilka zagadnień potrzebnych do ustalenia:

- ustalenie genomu jako reprezentanta wyniku
- ustalenie funkcji przystosowania/dopasowania
- ustalenie operatorów przeszukiwania

5 Sposoby krzyżowania

5.1 Krzyżowanie PMX



Kod zastosowanego krzyżowania:

```
do {
        begin = rand() % size;
```

```
    end = rand() % size;
} while ((0 >= (end - begin)) || !begin || !(end - (size - 1)));

for (int i = begin; i <= end; i++) {
    desc1[i] = parent1[i];
    desc2[i] = parent2[i];
    map1[parent1[i]] = parent2[i];
    map2[parent2[i]] = parent1[i];
}

for (int i = 0; i < size; i++) {
    if (desc1[i] == -1) {
        if (!isInPath(parent2[i], begin, end, desc1))
            desc1[i] = parent2[i];
        else {
            temp = parent2[i];

            do {
                temp = map1[temp];
            } while (isInPath(temp, begin, end, desc1));

            desc1[i] = temp;
        }
    }
}

for (int i = 0; i < size; i++) {
    if (desc2[i] == -1) {
        if (!isInPath(parent1[i], begin, end, desc2))
            desc2[i] = parent1[i];
        else {
            temp = parent1[i];

            do {
                temp = map2[temp];
            } while (isInPath(temp, begin, end, desc2));

            desc2[i] = temp;
        }
    }
}
```

5.2 Krzyżowanie przypisujące (Order)

Parent 1: 8 4 7 3 6 2 5 1 9 0

Parent 2: 0 1 2 3 4 5 6 7 8 9

Child 1: 0 4 7 3 6 2 5 1 8 9

Kod zastosowanego krzyżowania:

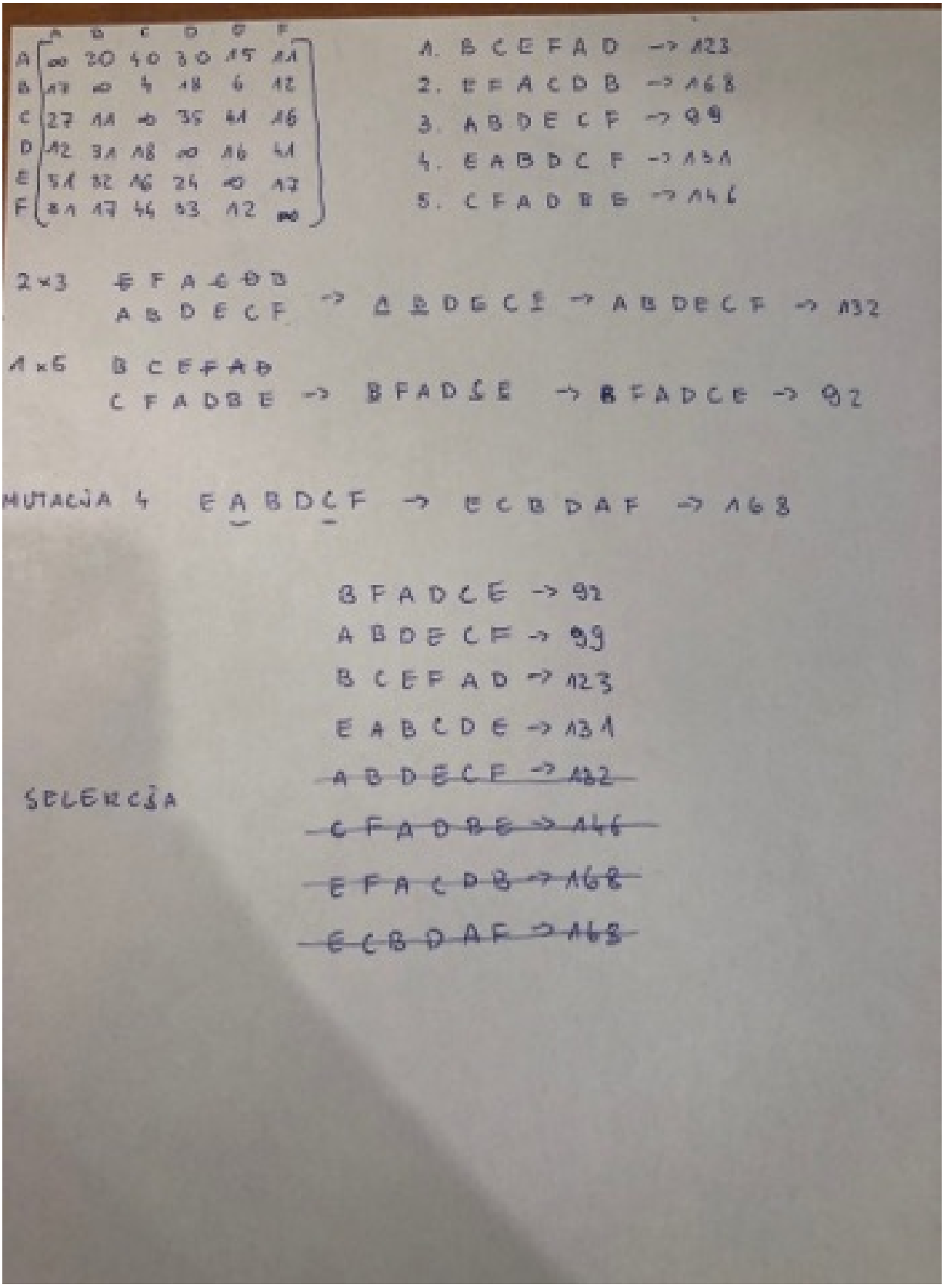
```
do {
        begin = rand() % size;
        end = rand() % size;
    } while ((0 >= (end - begin)) || !begin || !(end - (size - 1)));

    for (int i = begin; i <= end; i++) {
        desc1[i] = parent1[i];
        desc2[i] = parent2[i];
    }

    itr1 = desc1.begin() + end + 1;
    itr2 = parent2.begin() + end + 1;

    while (itr1 != desc1.begin() + begin) {
        if (!(isInPath(*itr2, begin, end, desc1))) {
            *itr1 = *itr2;

            if (itr1 == desc1.end() - 1)
                itr1 = desc1.begin();
            else
                itr1++;

            if (itr2 == parent2.end() - 1)
                itr2 = parent2.begin();
            else
                itr2++;
        }
        else {
            if (itr2 == parent2.end() - 1)
                itr2 = parent2.begin();
            else
                itr2++;
        }
    }

    itr1 = desc2.begin() + end + 1;
    itr2 = parent1.begin() + end + 1;

    while (itr1 != desc2.begin() + begin) {
```

```
        if (!(isInPath(*itr2, begin, end, desc2))) {
            *itr1 = *itr2;

            if (itr1 == desc2.end() - 1)
                itr1 = desc2.begin();
            else
                itr1++;

            if (itr2 == parent1.end() - 1)
                itr2 = parent1.begin();
            else
                itr2++;
        }
        else {
            if (itr2 == parent1.end() - 1)
                itr2 = parent1.begin();
            else
                itr2++;
        }
    }
```

6 Przykład praktyczny

W przykładzie praktycznym mamy pokazaną pierwszą iterację algorytmu dla 6 miastowej instancji problemu.
W pierwszym kroku losujemy populacje, w naszym przykładzie jest to 5 elementów.
W następnym kroku dokonujemy krzyżowania elementów z populacji. Są one dobierane losowo.
Ścieżkę pozostałą mutujemy.
Następnie wypisujemy wszystkie ścieżki, łącznie z tymi nowo powstałymi podczas procesu krzyżowania i sortujemy je rosnąco.
Teraz odbywa się selekcja.
Usuwamy najgorszą połowę populacji i przechodzimy do kolejnego obrotu algorytmu.

W taki sposób kontynuujemy wykonywanie algorytmu, aż do otrzymania zadowalającego wyniku.


A B C D E F
A ∞ 20 40 30 15 21
B 17 ∞ 4 18 6 12
C 27 14 ∞ 35 41 16
D 42 34 18 ∞ 16 41
E 31 32 16 24 ∞ 17
F 21 17 44 43 12 ∞
1. B C E F A D → 123
2. E F A C D B → 168
3. A B D E C F → 99
4. E A B D C F → 134
5. C F A D B E → 146
2×3 E F A C D B
A B D E C F → A E D B C F → A B D E C F → 132
1×5 B C E F A D
C F A D B E → B F A D E B → B F A D C E → 92
MUTACJA 4 E A B D C F → E C B D A F → 168
B F A D C E → 92
A B D E C F → 99
B C E F A D → 123
E A B C D E → 131
A B D E C F → 132
SELEKCJA
C F A D B E → 146
E F A C D B → 168
E C B D A F → 168

7 Implementacja algorytmu

Funkcja obliczająca koszt aktualnej ścieżki:

```
for (int i = 0; i < size; i++)
        permutation.push_back(i);

    for (int i = 0; i < populationSize; i++) {
        std::random_shuffle(permutation.begin(), permutation.end());
        population.push_back(permutation);
    }

    start = std::clock();

    // Kolejne iteracje algorytmu
    while (((std::clock() - start) / (CLOCKS_PER_SEC)) < stop) {
        fitness.clear();

        // Ocena jakości osobników
        for (auto& itr : population)
            fitness.push_back(calculatePath(itr));

        // Tworzenie nowej populacji na drodze selekcji
        for (int j = 0; j < populationSize; j++) {
            result = INT32_MAX;

            // Wybór najlepszego osobnika
            for (int k = 0; k < tournamentSize; k++) {
                index = rand() % populationSize;

                if (fitness[index] < result) {
                    result = fitness[index];
                    permutation.clear();
                    permutation = population[index];
                }
            }
            nextPopulation.push_back(permutation);
        }

        // Wymiana pokoleń
        for (auto& itr : population)
            itr.clear();

        population.clear();
        population = nextPopulation;
```

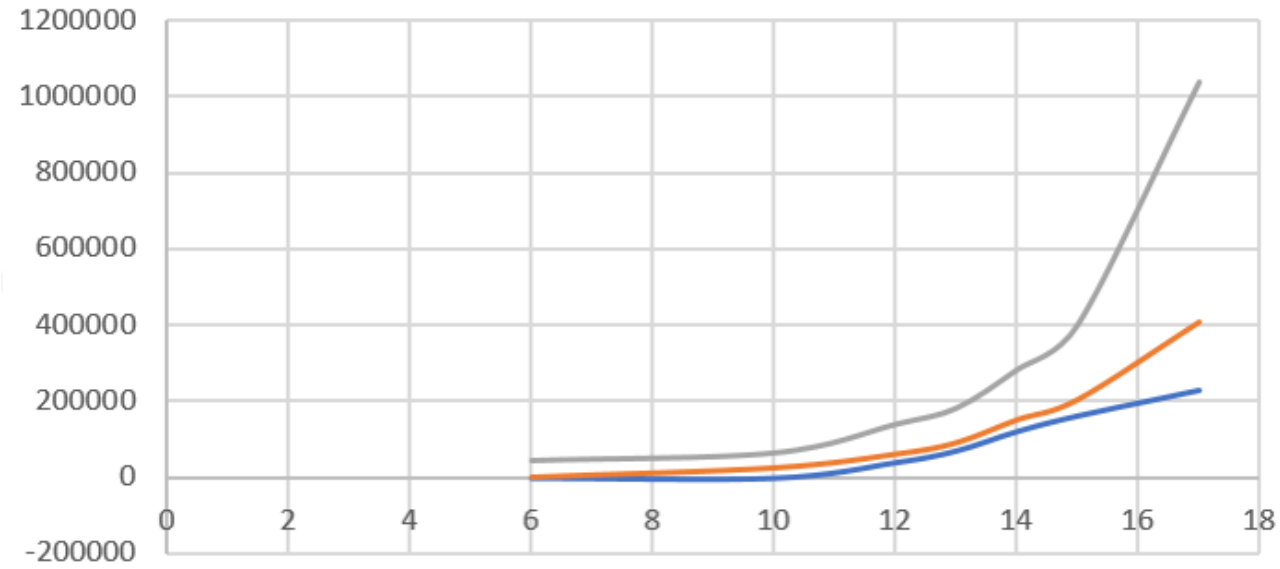
```
        for (auto& itr : nextPopulation)
            itr.clear();

        nextPopulation.clear();

        // Krzyżowanie osobników
        for (int j = 0; j < (int)(crossRate * (float)populationSize); j += 2) {
            do {
                p1 = rand() % populationSize;
                p2 = rand() % populationSize;
            } while (!(p1 - p2));

            if (crossing)
                orderedCrossover(population.at(j), population.at(j + 1));
            else
                PMXCrossover(population.at(j), population.at(j + 1));
        }

        // Mutacje osobników
        for (int j = 0; j < (int)(mutationRate * (float)populationSize); j++) {
            do {
                p1 = rand() % size;
                p2 = rand() % size;
            } while (!(p1 - p2));

            std::swap(population.at(j)[p1], population.at(j)[p2]);
        }
    }

    result = *(std::min_element(fitness.begin(), fitness.end()));
```

8 Plan eksperymentu

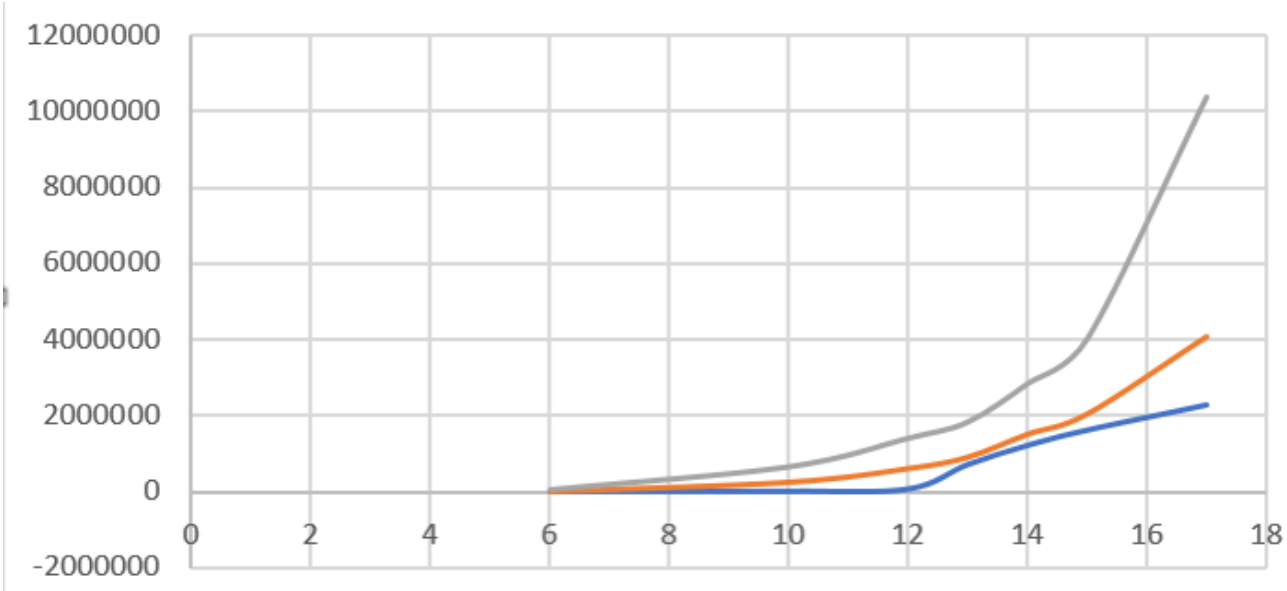
Dane oraz rozmiary testowanych danych pochodziły ze strony Pana Jarosława Mierzwy. Podobnie użyta metoda pomiaru czasu - funkcja QueryPerformanceCounter.

9 Wyniki eksperymentu

Wpływ wielkości populacji na wyniki dla trzech różnych wartości. Przyjęto współczynnik krzyżowania 0.8 oraz współczynnik mutacji 0.01.
Poniższy wykres przedstawia czas obliczania średniej najkrótszej ścieżki w zależność od populacji: niebieski - populacja 5, pomarańczowy - populacja 10, szary - populacja 20. Przyjęte krzyżowanie metodą Order.



Poniższy wykres przedstawia takie same wartości, ale dla krzyżowania metodą PMX.



10 Wnioski

Z racji tego, że algorytm Genetyczny jest algorytmem heurystycznym nie daje nam gwarancji uzyskania optymalnego rozwiązania. Każdorazowe wykonanie go może dać różne rozwiązanie od poprzednich, jednak wszystkie podczas testów oscylowały wokół tego minimalnego. Z tego powodu możemy uznać ten algorytm za bardzo efektywny. Jego czas nie rośnie przy większych instancjach z tego powodu, że przeglądamy tylko ustaloną populację. Jedyną różnicą jest długość każdej instancji. Najwięcej czasu w algorytmie, pochłania operacja krzyżowania PMX, jest ona dość skomplikowana ze względu na kilkukrotne przeglądanie elementów instancji i konieczność tworzenia tymczasowych miejsc ich przechowywania. Poza tym algorytm działa bardzo szybko i jest efektywniejszy od algorytmu Tabu Search.