Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

SPRAWOZDANIE

PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW

Implementacja i analiza efektywności algorytmu genetycznego

Autor:
Klaudia Marzec #263890

Prowadzący dr inż. Dariusz Banasiak

1 Wstęp teoretyczny

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

Algorytm genetyczny opiera się na mechanizmach ewolucji biologicznej, takich jak selekcja naturalna, krzyżowanie i mutacja. Proces rozpoczyna się od stworzenia początkowej populacji osobników. Każdy z nich 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, a funkcja przystosowania ocenia, jak dobre jest rozwiązanie.

Każda jednostka w populacji jest oceniana pod kątem dostosowania do problemu, a następnie wybierane są jednostki najlepiej dostosowane do środowiska. Te wybrane jednostki są następnie krzyżowane i mutowane, generując potomstwo, które zastępuje starszą populację. Proces ten jest powtarzany przez wiele pokoleń, aż do uzyskania zadowalającego rozwiązania.

W kontekście problemu komiwojażera, jednostki w populacji reprezentują różne trasy przejścia przez miasta, a jakość trasy ocenia się na podstawie długości przebytej drogi. Krzyżowanie i mutacja mają na celu modyfikację tras, eksplorując przestrzeń rozwiązań w poszukiwaniu optymalnego rozwiązania.

Złożoność obliczeniowa algorytmu genetycznego zależy od wielu czynników, takich jak rozmiar populacji, częstość reprodukcji, operatory genetyczne (krzyżowanie, mutacja) oraz sposób oceny dostosowania. Podczas implementacji algorytmu przyjęto takie parametry jak: rozmiar populacji, częstość reprodukcji, częstość krzyżowania, częstość mutacji oraz wybór operatora krzyżowania. Zaimplementowano dwa operatory - operator krzyżowania PMX oraz operator krzyżowania OX. Dodatkowo zastosowano mutację poprzez inwersję, której złożoność obliczeniowa wynosi $O(N)$, gdzie N- liczba miast.

Krzyżowanie PMX (Partially Mapped Crossover) polega na częściowej zamianie genów pomiędzy dwoma rodzicami w celu wygenerowania potomka. Złożoność obliczeniowa dla tego operatora wynosi $O(N)$

Krzyżowanie OX polega na utrzymaniu pewnej sekwencji genów z jednego rodzica, a resztę genów uzupełnianiu z drugiego rodzica, zachowując ich kolejność. Złożoność obliczeniowa dla tego operatora również wynosi $O(N)$, zatem złożoność obliczeniowa dla zaimplementowanego algorytmu to:

$O(reproductionRate * (crossCoeff * O(N) + mutationRate * O(N)))$, gdzie

- reproductionRate- częstość reprodukcji
- crossCoeff- częstość krzyżowania
- mutationRate- częstość mutacji

2 Przykład praktyczny - Opis działania algorytmu krok po kroku

Przyjmijmy przykładową instancję miast o rozmiarze 7. Macierz odległości pomiędzy miastami prezentuje się następująco:

MIASTA

	0	1	2	3	4	5	6
0	0	81	193	659	773	573	984
1	104	0	580	259	508	426	939
2	196	589	0	877	62	861	870
3	644	231	830	0	241	411	997
4	741	546	88	283	0	551	773
5	526	425	861	375	539	0	803
6	995	891	827	995	771	840	0

Przyjęte parametry:

- Rozmiar populacji: 10
- Częstość reprodukcji: 20
- Częstość krzyżowania: 2
- Częstość mutacji: 1

Pierwszym etapem algorytmu jest losowe wygenerowanie populacji miast o zadanym rozmiarze, która prezentuje się następująco:

```
POPULACJA
0->5->2->3->1->6->4->0: 4993
0->5->6->2->4->3->1->0: 2883
0->1->2->3->4->6->5->0: 3918
0->4->1->2->3->6->5->0: 5139
0->6->5->4->1->2->3->0: 5010
0->5->2->4->1->6->3->0: 4620
0->3->6->5->4->1->2->0: 4357
0->1->4->5->3->2->6->0: 4210
0->1->5->3->6->2->4->0: 3509
0->5->4->3->6->2->1->0: 3912
```

Następnie w sposób losowy zostają wylosowane dwie jednostki do krzyżowania:
0->1->4->5->3->2->6->0: 4210
0->3->6->5->4->1->2->0: 4357

Potomek 0->2->6->5->4->1->3->0: 3891 utworzony po krzyżowce zostaje dodany na koniec populacji:

```
POPULACJA PO KRZYZOWCE
0->5->2->3->1->6->4->0: 4993
0->5->6->2->4->3->1->0: 2883
0->1->2->3->4->6->5->0: 3918
0->4->1->2->3->6->5->0: 5139
0->6->5->4->1->2->3->0: 5010
0->5->2->4->1->6->3->0: 4620
0->3->6->5->4->1->2->0: 4357
0->1->4->5->3->2->6->0: 4210
0->1->5->3->6->2->4->0: 3509
0->5->4->3->6->2->1->0: 3912
0->2->6->5->4->1->3->0: 3891
```

W kolejnym kroku następuje drugie krzyżowanie wylosowanych elementów, którymi są:
0->4->1->2->3->6->5->0: 5139
0->5->2->3->1->6->4->0: 4993

Nowy potomek 0->5->2->3->1->6->4->0: 4993 również zostaje dodany na koniec populacji:

```
POPULACJA PO KRZYZOWCE
0->5->2->3->1->6->4->0: 4993
0->5->6->2->4->3->1->0: 2883
0->1->2->3->4->6->5->0: 3918
0->4->1->2->3->6->5->0: 5139
0->6->5->4->1->2->3->0: 5010
0->5->2->4->1->6->3->0: 4620
0->3->6->5->4->1->2->0: 4357
0->1->4->5->3->2->6->0: 4210
0->1->5->3->6->2->4->0: 3509
0->5->4->3->6->2->1->0: 3912
0->2->6->5->4->1->3->0: 3891
0->5->2->3->1->6->4->0: 4993
```

W następnym kroku zostaje losowo wybrana jednostka 0->5->2->4->1->6->3->0: 4620, na której zostanie przeprowadzona mutacja. W jej fekcie powstaje nowa jednostka: 0->5->2->4->6->1->3->0: 4063. Populacja prezentuje się teraz następująco:

```
POPULACJA PO MUTACJI
0->5->2->3->1->6->4->0: 4993
0->5->6->2->4->3->1->0: 2883
0->1->2->3->4->6->5->0: 3918
0->4->1->2->3->6->5->0: 5139
0->6->5->4->1->2->3->0: 5010
0->5->2->4->6->1->3->0: 4063
0->3->6->5->4->1->2->0: 4357
0->1->4->5->3->2->6->0: 4210
0->1->5->3->6->2->4->0: 3509
0->5->4->3->6->2->1->0: 3912
0->2->6->5->4->1->3->0: 3891
0->5->2->3->1->6->4->0: 4993
```

Ostatnim krokiem jest uporządkowanie jednostek w kolejności od najlepszej do najgorszej, a następnie usunięcie najgorszych jednostek tak, aby pozostała populacja o zadanym rozmiarze.

```
POPULACJA
0->5->6->2->4->3->1->0: 2883
0->1->5->3->6->2->4->0: 3509
0->2->6->5->4->1->3->0: 3891
0->5->4->3->6->2->1->0: 3912
0->1->2->3->4->6->5->0: 3918
0->5->2->4->6->1->3->0: 4063
0->1->4->5->3->2->6->0: 4210
0->3->6->5->4->1->2->0: 4357
0->5->2->3->1->6->4->0: 4993
0->5->2->3->1->6->4->0: 4993
```

W tym momencie kończy się pierwsza reprodukcja. Po wykonaniu zadanej ilości reprodukcji algorytm zwraca najlepsze rozwiązanie, którym jest pierwsza jednostka w populacji. W tym przypadku po 20 reprodukcjach algorytm zwrócił rozwiązanie: 0->2->4->6->5->3->1->0: 2578, będące rzeczywistym najlepszym rozwiązaniem.

3 Opis implementacji algorytmu

W zaimplementowanym algorytmie została użyta klasa Graf zawierająca informacje o strukturze grafu oraz klasa Macierz, przechowująca dwuwymiarową tablicę dynamiczną krawędzi w grafie. Algorytm genetyczny jest wywoływany z podziomu klasy Macierz po podaniu przez użytkownika odpowiednich parametrów.

```
int* Macierz::geneticAlgorithm(int populationSize, int reproductionRate, int crossCoefficient, int mutationRate, int start, int choose)
{
    Population population(populationSize, macierz, rozmiar_w, start);
    population.generatePopulation();

    for(int j=0; j<reproductionRate; j++)
    {
        for(int i=0; i<crossCoefficient; i++)
            population.crossover(choose);

        for(int i=0; i<mutationRate; i++)
            population.inversionMutation();

        population.deleteTravels(populationSize);
    }

    int *best = new int[rozmiar_w+1];

    for(int i=0; i<=rozmiar_w; i++)
        best[i] = population.returnBest()[i];

    return best;
}
```

Podczas implementacji została stworzona dodatkowa klasa Population, która zawiera metody odpowiadające za prawidłowe działanie algorytmu, między innymi krzyżowanie, mutację oraz selekcję.

```
class Population
{
private:
    int populationSize, matrixSize, start;
    int** population;
    int **matrix;
    HANDLE hOut1 = GetStdHandle( STD_OUTPUT_HANDLE );

public:

    // konstruktor populacji
    Population(int sizePop, int **matrixx, int matrixSize, int start);

    void generatePopulation();

    int getDistance(int *droga);

    int* generateTravel();

    void display();

    void displayTravel(int index);

    int selection(int selectedIndex);

    void crossover(int choose);

    int* crossoverPMX(int parent1, int parent2, int x1, int x2);

    int* crossoverOX(int parent1, int parent2, int x1, int x2);

    void addtoPopulation(int *travel);

    void inversionMutation();

    void deleteTravels(int popSize);

    int* returnBest();

    ~Population();
};
```

W programie wybrano selekcję turniejową jako wybór jednostek najlepiej przystosowanych do życia. Funkcja jest wywoływana dwa razy w momencie wyboru rodziców w funkcji krzyżowania, dlatego jako parametr podawany jest $selectedIndex$, czyli indeks który został już wcześniej wybrany jako najlepszy w populacji. Służy to zminimalizowaniu czasu wykonywania selekcji. Tworzone są cztery grupy o rozmiarze $populationSize/8$, z których najlepsze jednostki trafiają do kolejnej grupy. Funkcja zwraca najlepszego osobnika z ostatniej finałowej grupy.

```
// NAJLEPSZY Z NAJLEPSZYCH

mini = getDistance(population[group5[0]]);;
winner = group5[0];

for(int j=1; j<4; j++)
{
    int newCost = getDistance(population[group5[j]]);
    if(newCost < mini)
    {
        mini = newCost;
        winner = group5[j];
    }
}

return winner;
```

W funkcji krzyżowania *crossover* najpierw losowanych jest dwóch rodziców za pomocą funkcji *selection*(), a następnie przedział w trasie, który ulegnie modyfikacji. Następnie w zależności od wybranego wcześniej przez użytkownika operatora krzyżowania, zostaje wywołana odpowiednia funkcja. W dalszej części algorytm wybiera lepszego potomka i dodaje go do populacji.

```
    switch(choose)
    {
        case 1:
        {
            child1 = crossoverPMX(candidate1, candidate2, x1, x2);
            child2 = crossoverPMX(candidate2, candidate1, x1, x2);

            break;
        }
        case 2:
        {
            child1 = crossoverOX(candidate1, candidate2, x1, x2);
            child2 = crossoverOX(candidate2, candidate1, x1, x2);

            break;
        }
    }

    int cost1 = getDistance(child1);
    int cost2 = getDistance(child2);

    if(cost1<cost2)
        addtoPopulation(child1);
    else
        addtoPopulation(child2);
}
```

Funkcja *inversionMutation* odpowiedzialna za wykonanie mutacji również wybiera kandydata, na którym zostanie ona przeprowadzona, lecz tym razem w sposób losowy, aby nie wpłynąć negatywnie na efekty krzyżowania. Wylosowany przedział, na którym zostanie przeprowadzona operacja, zostaje skopiowany w odwróconej kolejności do funkcji pomocniczej, a następnie zmieniona zostaje kolejność odwiedzania wierzchołków w wylosowanej jednostce.

4 Plan eksperymentu

W celu sprawdzenia poprawności działania algorytmu, dokonano pomiaru czasu działania w zależności od rozmiaru problemu N. Testy efektywności algorytmu zostały przeprowadzone na losowo wygenerowanych grafach o rozmiarach:.... Każdy pomiar został powtórzony 100 razy.

Generowanie grafu

Każd graf, na którym wykonano pomiary, został losowo wygenerowany za pomocą odpowiedniej funkcji uzupełniającej strukturę grafu losowymi wartościami. Funkcja uzupełnia graf w taki sposób, aby odległości między miastami A->B oraz B->A miały wartości zbliżone do siebie z maksymalną różnicą 50.

```
// Uzupełnienie macierzy

for(int i=0; i<wierzcholki; i++)
{
    for(int j=0; j<wierzcholki; j++)
    {
        if(graf.matrixValue(i, j) == -1)
        {
            // Uzupełnienie

            if(i == j)
            {
                droga = 0;
                droga2 = 0;
            }
            else
            {
                droga = rand() % 1000 + 1;          // Losowanie odleglosci z zakresu <1, 1000>

                // Wylosowanie odleglosci przeciwnej

                // Losowanie znaku + lub -

                int znak = rand() % 2;

                // 0 -
                // 1 +

                // Losowanie rozmicy

                int rozmica = rand() % 51;

                if(znak == 0)
                    droga2 = droga - rozmica;
                else
                    droga2 = droga + rozmica;
            }

            graf.completeMacierz(droga, i, j);

            // Uzupełnienie odbicia lustrzanego

            graf.completeMacierz(droga2, j, i);
        }
    }
}

    graf.pomiary(wierzcholki);
}
```

Pomiar czasu

Do pomiarów czasu została wykonana operacja $QueryPerformanceCounter$- start przed wywołaniem danej funkcji, koniec zaraz po zakończeniu jej działania. Wyniki były zapisywane w pliku tekstowym.

5 Wyniki

Poniższa tabela przedstawia przyjęte podczas pomiarów parametry dla konkretnych rozmiarów instancji. Dodatkowo zostały w niej uwzględnione błędy względne dla dwóch rodzajów krzyżowania zaimplementowanych w projekcie.
Błąd liczony jest ze wzoru: $\frac{wynik - prawidlowyWynik}{prawidlowyWynik} * 100\%$

N	popSize	repRate	crossCoeff	mutRate	Błąd PMX [%]	Błąd OX [%]
8	30	80	10	1	0%	0%
9	50	150	10	2	0.25%	0%
10	100	200	50	2	4%	2%
11	150	300	100	2	2.70%	1.6%
12	200	300	150	2	2%	0.8%
13	200	300	200	2	7%	5.4%
17	50	100	40	2	10%	8%
33	150	300	100	3	55%	34%
38	200	300	100	3	56%	31%
44	250	300	150	4	69%	40%
55	250	300	200	4	197%	55%

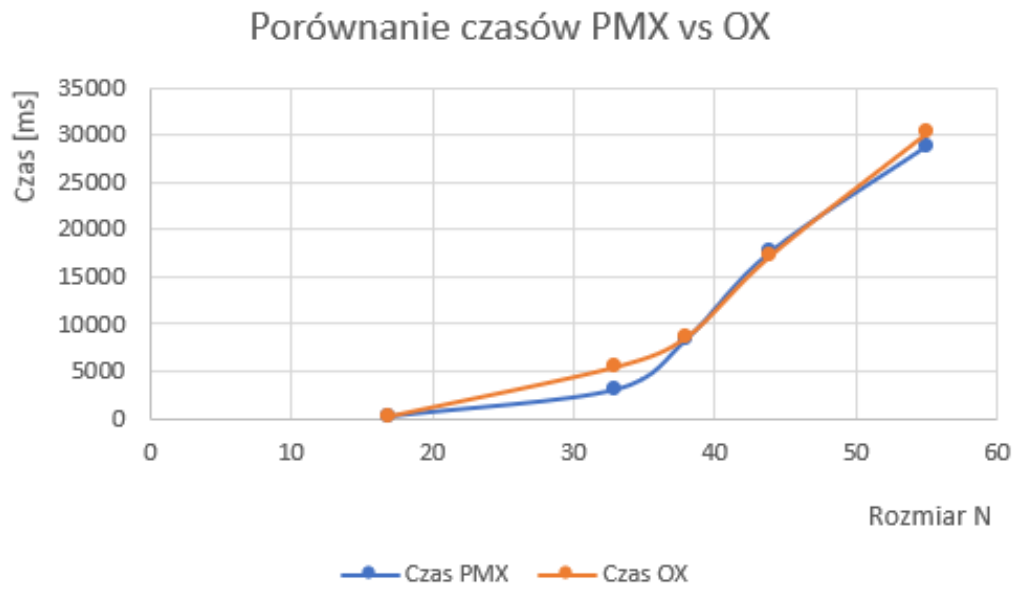
Pomiar czasu działania algorytmu oraz jakość otrzymanego rozwiązania w zależności od rozmiaru problemu N:

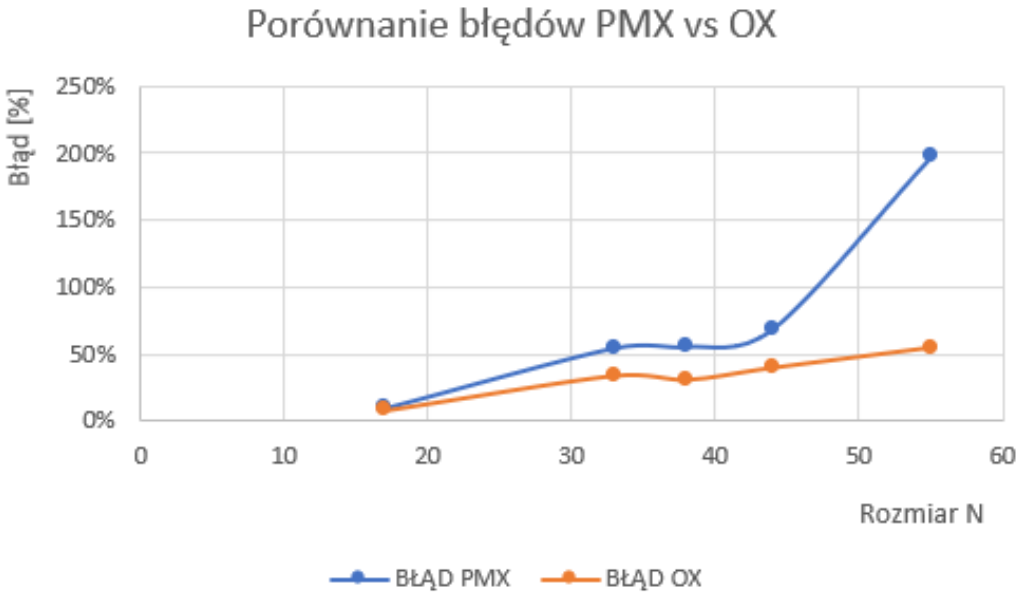
1. Krzyżowanie PMX:

N	t[ms]	BŁĄD
17	200.711	10%
33	3050.92	55%
38	8393.69	56%
44	17703.5	69%
55	28808.7	197%

2. Krzyżowanie OX:

N	t[ms]	BŁĄD
17	140.447	8%
33	5381.65	34%
38	8535.79	31%
44	17267	40%
55	30195	55%



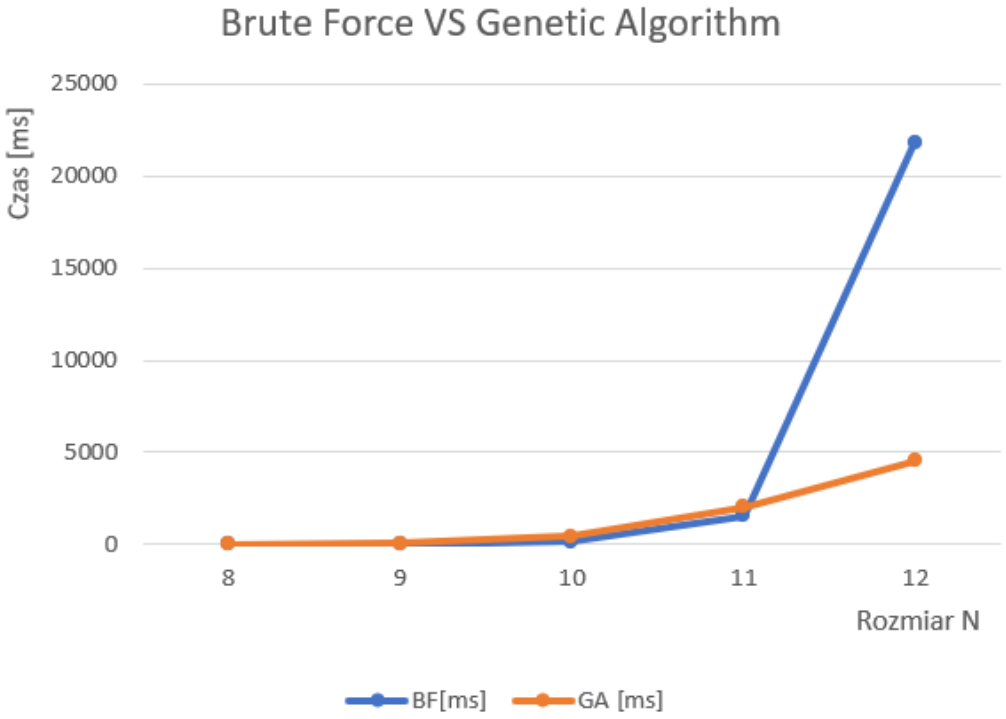


Parametry dobrane podczas porównywania algorytmu genetycznego z algorytmem Brute Force oraz Branch & Bound zostały dobrane w taki sposób, aby znajdować jak najlepszy wynik, dlatego czas trwania algorytmu jest stosunkowo długi.

Porównanie algorytmu genetycznego z algorytmem Brute Force:

Algorytm genetyczny działa o wiele szybciej niż algorytm Brute Force, co można zauważyć już dla instancji o rozmiarze 12. Ponadto algorytm genetyczny znajduje w tym czasie prawidłowe rozwiązanie z bardzo dużą dokładnością, ponieważ błąd wynosi zaledwie około 2% dla instancji o rozmiarze 12. Algorytm genetyczny jest zdecydowanie szybszym algorytmem niż Brute Force, pozwalającym uzyskać prawidłowy wynik w o wiele krótszym czasie.

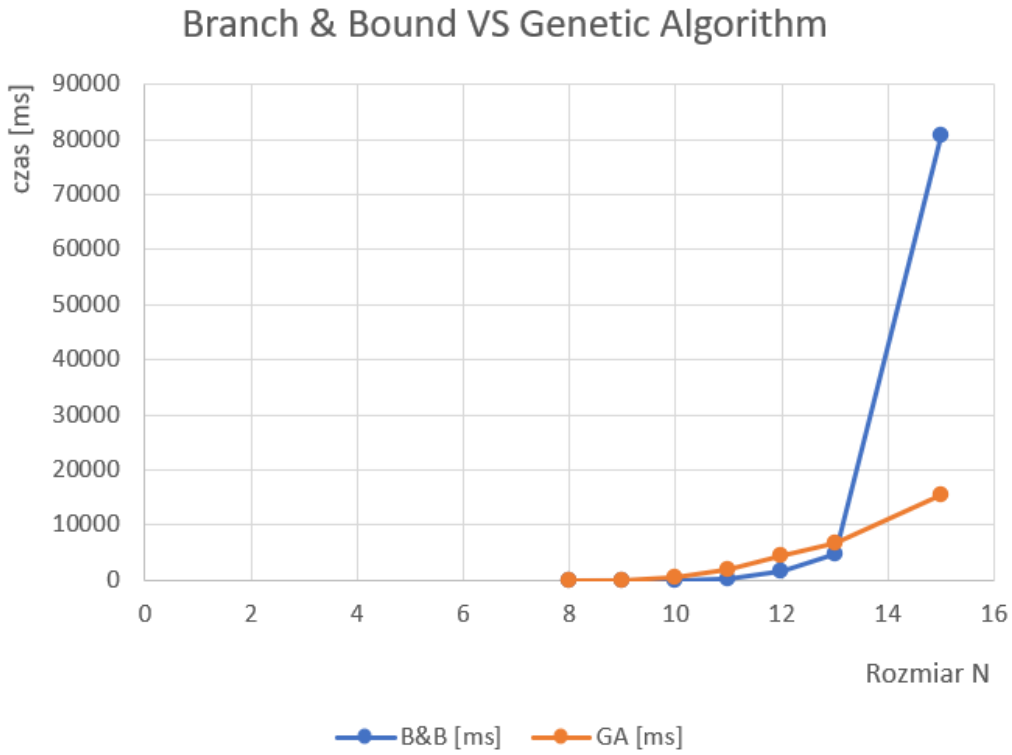
N	Brute Force [ms]	GA [ms]
8	1.48	9.51
9	15	38.9
10	142.23	431.49
11	1546	2051.55
12	21811	4553.51



Porównanie algorytmu genetycznego z algorytmem Branch & Bound:

Dla mniejszych instancji algorytm Branch & Bound sprawdza się o wiele lepiej, ponieważ zwraca prawidłowy wynik w bardzo krótkim czasie. Aby algorytm genetyczny był w stanie zwrócić jak najlepsze rozwiązanie, potrzebuje zdecydowanie więcej czasu. Jednakże im większa instancja, tym ta różnica jest coraz mniejsza. Ponadto dla instancji o rozmiarze 15 czas trwania algorytmu Branch & Bound staje się dłuższy od algorytmu genetycznego. Łatwo można wysnuć wniosek, że dla większych instancji algorytm B&B nie będzie tak dobrym wyborem jak algorytm genetyczny ze względu na zbyt długi czas obliczeń.

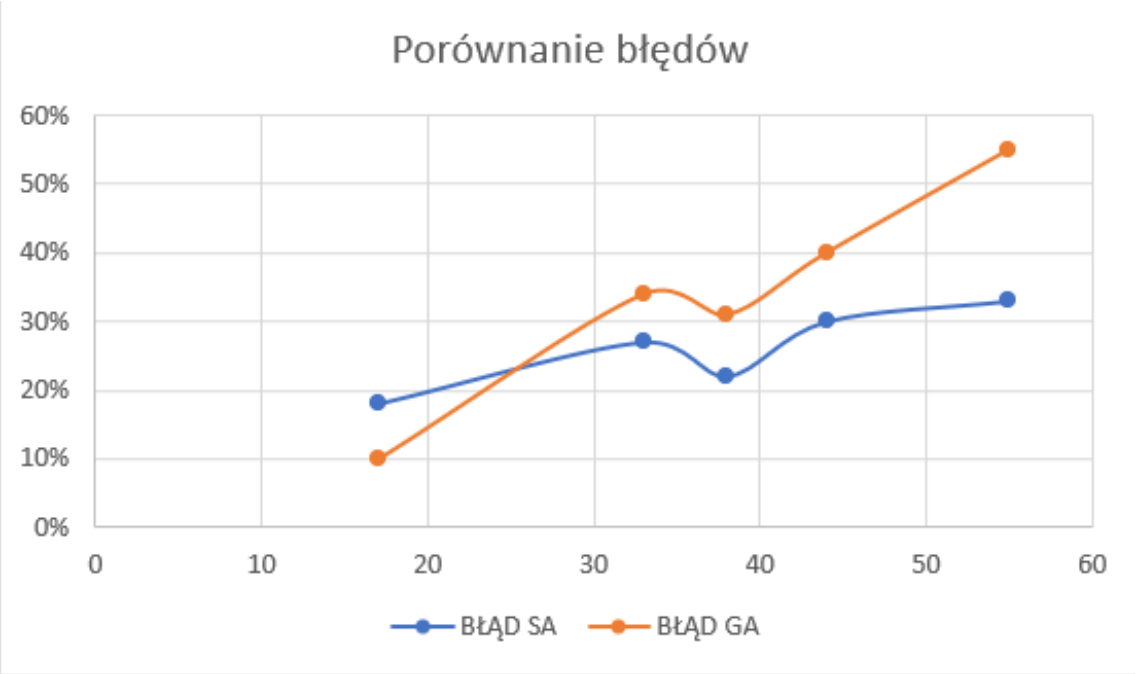
N	B&B [ms]	GA [ms]
8	4.24	9.51
9	5.8	38.9
10	26.52	431.49
11	160.18	2051.55
12	1528.48	4553.51
13	4843.32	6687.72
15	80682	15449

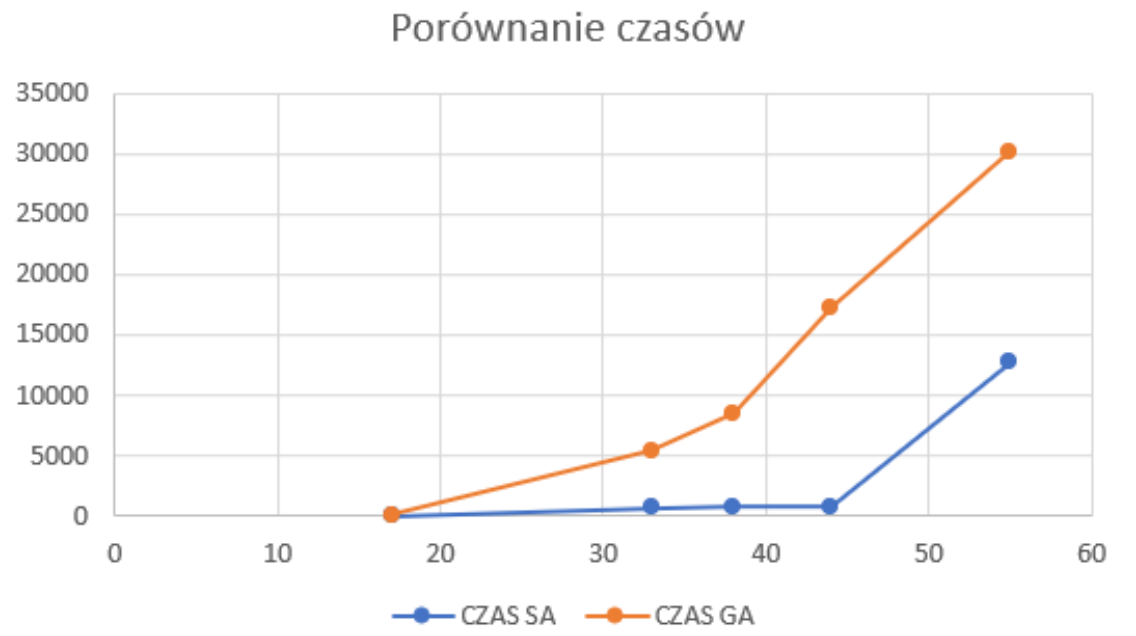


Porównanie algorytmu genetycznego z algorytmem symulowanego wyżarzania:

Do porównania algorytmów wybrano krzyżowanie OX, ponieważ uzyskuje ono o wiele lepsze wyniki w podobnym czasie niż krzyżowanie PMX. Poniższa tabela przedstawia średni błąd względny uzyskanych wyników oraz średni czas obliczeń. W tym wypadku można łatwo zauważyć, że algorytm genetyczny nie jest takim dobrym rozwiązaniem jak algorytm symulowanego wyżarzania. Zdarza się, że błąd jest mniejszy, jednakże czas wykonywania obliczeń jest zdecydowanie dłuższy niż dla algorytmu symulowanego wyżarzania.

	SA		GA	
N	BŁĄD	CZAS [ms]	BŁĄD	CZAS [ms]
17	18%	0.442	10%	140.45
33	27%	682.9	34%	5381
38	22%	824.23	31%	8535
44	30%	845.71	40%	17267
55	33%	12739	55%	30194





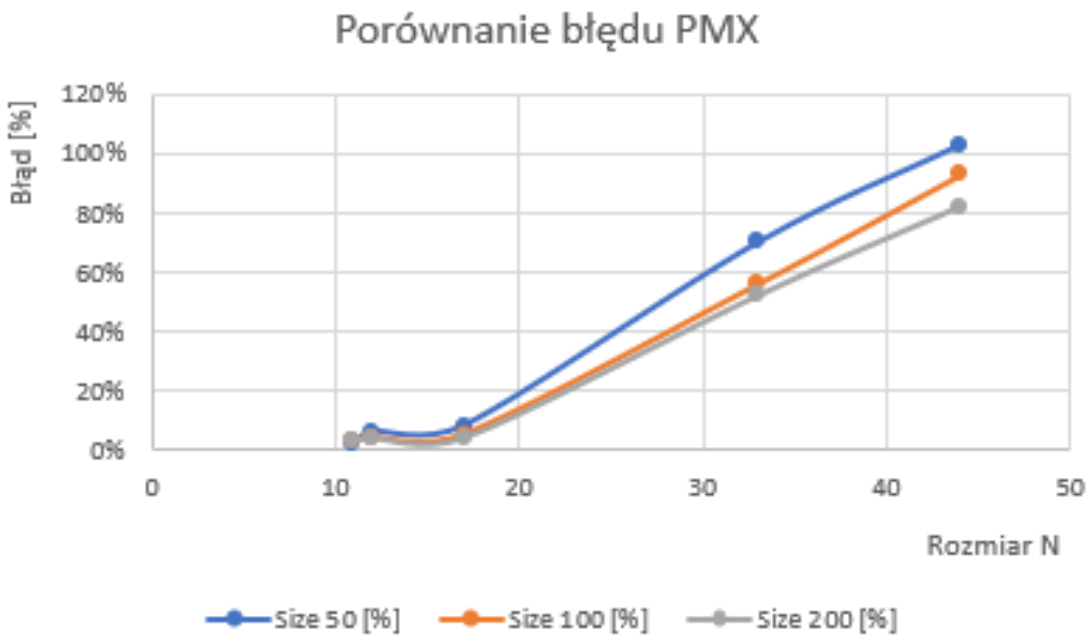
Badanie wpływu wielkości populacji na uzyskane wyniki dla trzech różnych wartości, stałe wartości współczynnika krzyżowania i mutacji

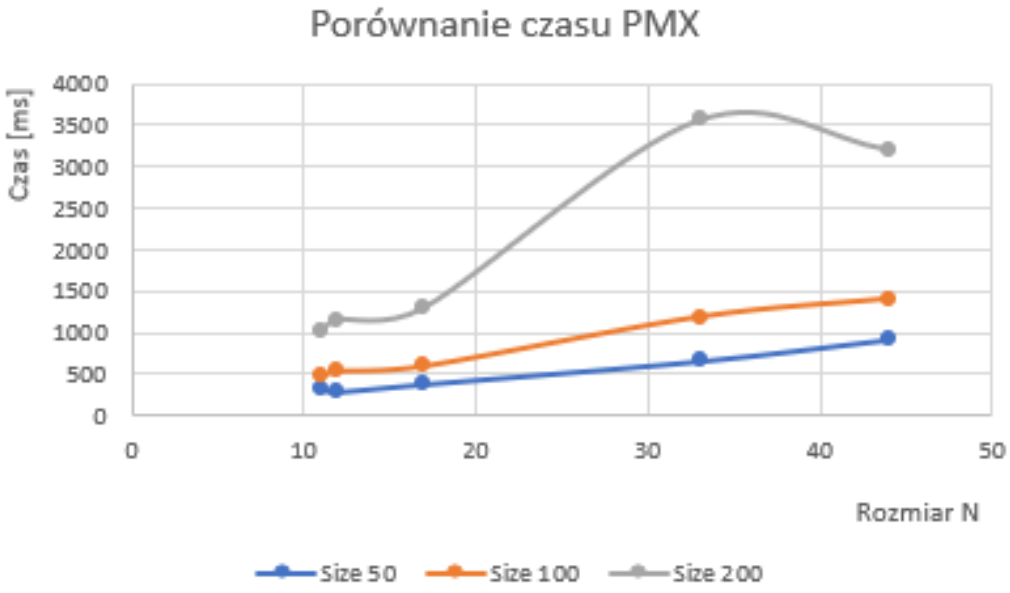
Przyjęte stałe wartości podczas badania wpływu wielkości populacji na uzyskane wyniki:

- współczynnik reprodukcji = 100
- współczynnik krzyżowania = 100
- współczynnik mutacji = 2

Krzyżowanie PMX:

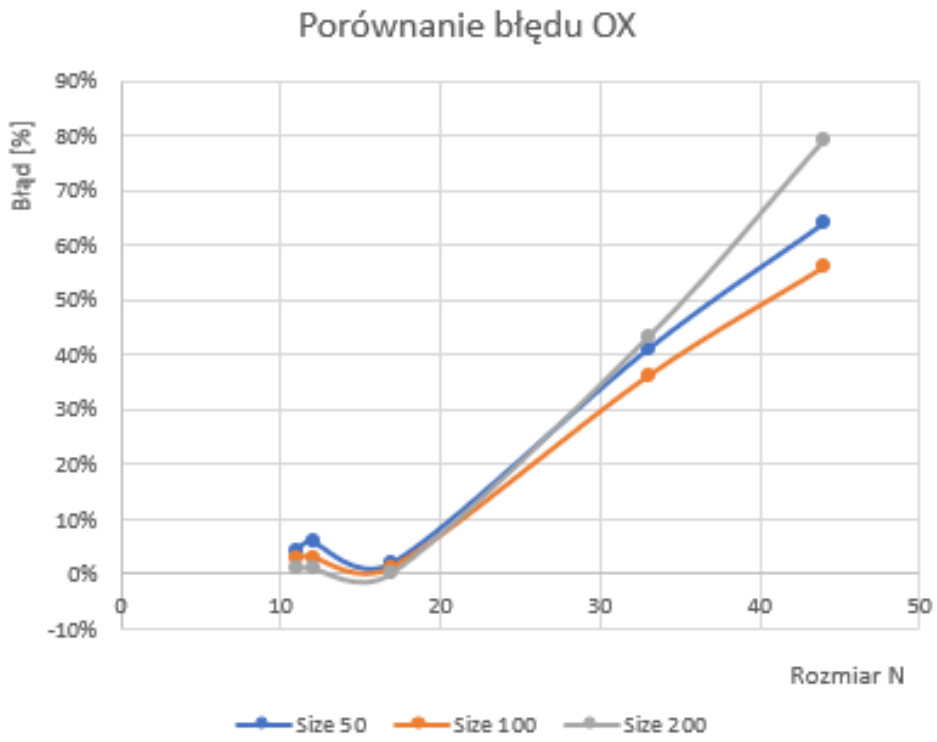
	popSize = 50		popSize = 100		popSize = 200	
N	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]
11	316	2%	474	3%	1001	3%
12	283	6%	542	4%	1158	4%
17	375	8%	603	5%	1297	4%
33	656	70%	1190	56%	3567	52%
44	918	103%	1409	93%	3213	82%

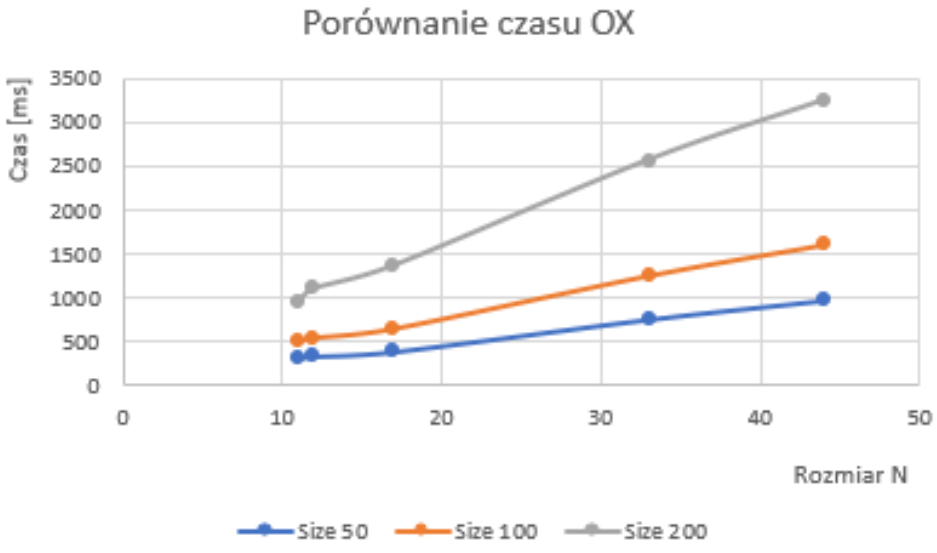




Krzyżowanie OX:

	popSize = 50		popSize = 100		popSize = 200	
N	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]
11	306	4%	490	3%	931	1%
12	333	6%	540	3%	1103	1%
17	385	2%	642	1%	1363	0%
33	754	41%	1248	36%	2562	43%
44	971	64%	1602	56%	3241	79%



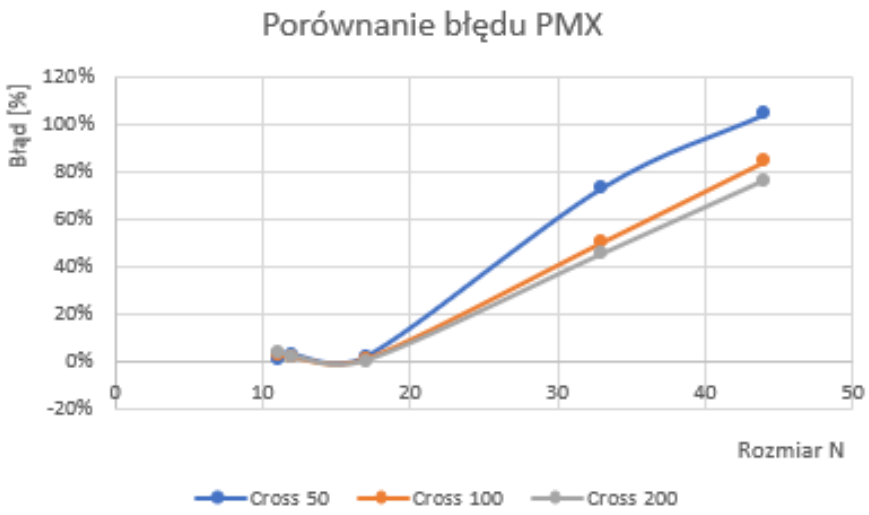


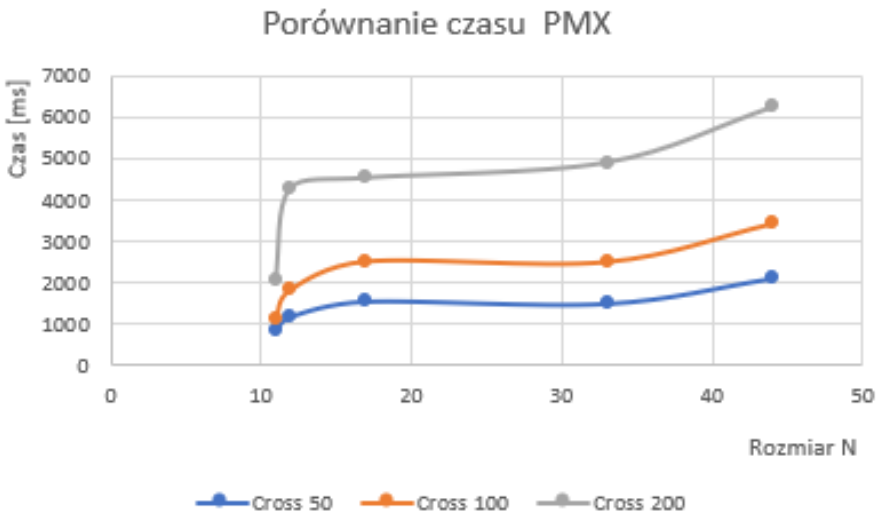
Badanie wpływu współczynnika krzyżowania przy ustalonej wielkości populacji i ustlonym współczynniku mutacji

- rozmiar populacji = 200
- współczynnik reprodukcji = 100
- współczynnik mutacji = 2

Krzyżowanie PMX:

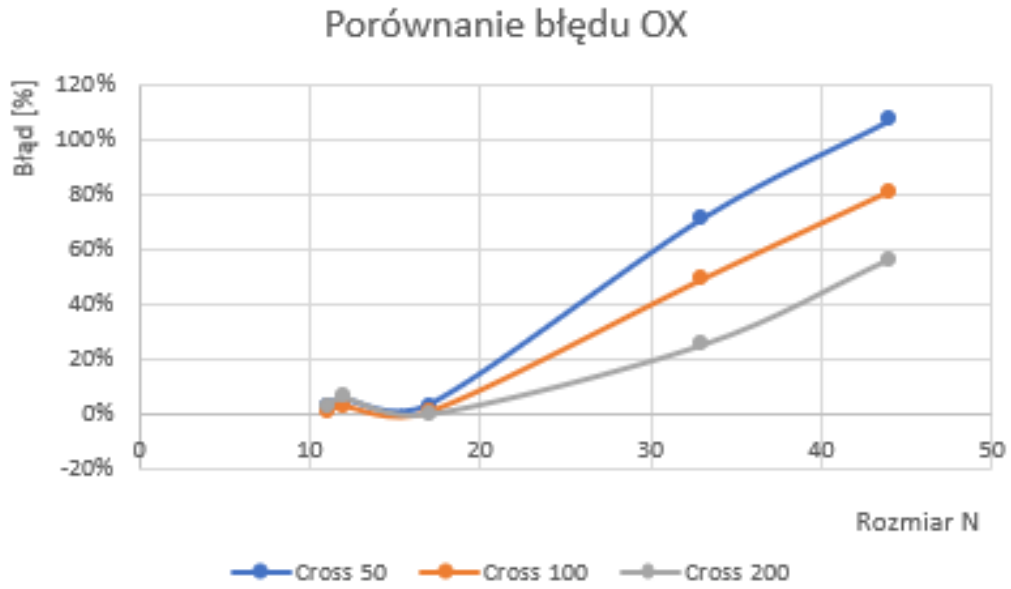
	crossCoeff = 50		crossCoeff = 100		crossCoeff = 200	
N	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]
11	819	1%	1114	3%	2033	4%
12	1168	3%	1837	2%	4288	2%
17	1539	2%	2523	1%	4536	0%
33	1485	73%	2509	50%	4904	45%
44	2091	104%	3422	84%	6238	76%

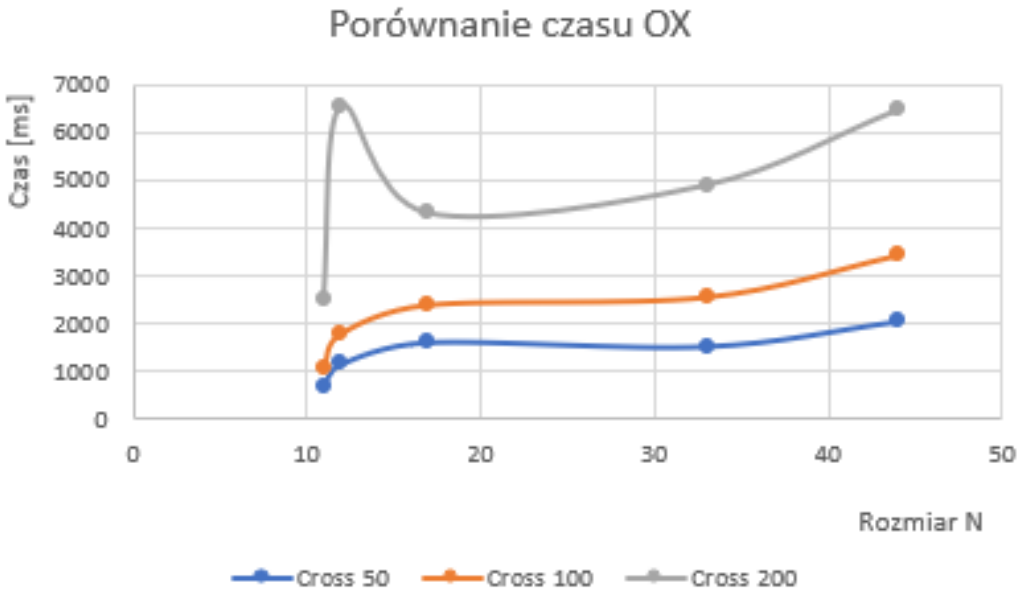




Krzyżowanie OX:

	crossCoeff = 50		crossCoeff = 100		crossCoeff = 200	
N	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]	t[ms]	BŁĄD [%]
11	661	3%	1042	1%	2522	3%
12	1144	5%	1783	3%	6547	6%
17	1603	3%	2394	1%	4311	0%
33	1518	71%	2556	49%	4892	25%
44	2034	107%	3430	81%	6464	56%





6 Wnioski

Na podstawie wykonanych pomiarów można stwierdzić, że czas wykonywania obliczeń operatorów krzyżowania jest porównywalny. Różnica między nimi jest praktycznie niezauważalna, co można wywnioskować z wykresu. Ponadto krzyżowanie OX uzyskuje o wiele lepsze wyniki niż krzyżowania PMX, ponieważ błąd względny jest zdecydowanie niższy, co zwiększa prawdopodobieństwo uzyskania prawidłowego rozwiązania. Dla obu operatorów krzyżowania czas wykonywania obliczeń- niezależnie od ustawionych parametrówrośnie. Wynika to z faktu, że złożoność obliczeniowa zarówno dla krzyżowania PMX, jak i krzyżowania OX zależy od rozmiaru instancji N. Wraz z jej wzrostem, rośnie również błąd względny otrzymanego wyniku.

Z wykresów dotyczących badania wpływu wielkości populacji na uzyskane wyniki wniosek zarówno dla krzyżowania PMX, jak i OX jest taki sam. Z łatwością można zauważyć, że im większa populacja, tym błąd obliczeń jest coraz mniejszy. Dla im większych instancji, różnica ta jest bardzo mocno zauważalna, ponieważ wynosi nawet 10 p.p. Kosztem lepszych wyników jest wydłużający się czas trwania obliczeń, jednak nie przekracza on optymalnego czasu wykonywania oblczeń.

Badanie wpływu współczynnika krzyżowania przy ustalonej wielkości populacji oraz współczyniku mutacji przyniosło podobny wynik jak wcześniej. W tym wypadku również zarówno dla krzyżowania PMX, jak i OX łatwo można zauważyć, że przy większym współczynniku krzyżowania, błąd obliczeń maleje. Oczywiście idzie za tym zwiększony czas wykonywania obliczeń, jednakże mieści się on w normie optymalnego czasu działania algorytmu.

Algorytm gnetyczny jest o wiele lepszym wyborem niż Brute Force czy Branch & Bound, ponieważ jest w stanie uzyskać optymalne rozwiązanie w zdecydowanie krótszym czasie. Jednakże algorytm symulowanego wyżarzania jest lepszym wyborem niż algorytm genetyczny. Jest on szybszy i dokładniejszy dla większych instancji.

Algorytm symulowanego wyżarzania jest bardziej efektywny w poszukiwaniu globalnego minimum. Jest to spowodowane tym, że obniża temperaturę stopniowo, co pozwala mu na akceptowanie gorszych rozwiązań, które mogą być bliższe globalnemu minimum. Jest także mniej podatny na utknięcie w lokalnych minimach. Jest to spowodowane tym, że akceptuje gorsze rozwiązania z pewnym prawdopodobieństwem, co pozwala mu na opuszczenie lokalnego minimum. Te czynnika wpływają na to, że jest on lepszym algorytmem niż algorytm genetyczny.

7 Kod źródłowy

https://github.com/Klaudiaamarzec/Genetic-Algorithm

8 Bibliograia

- https://www.kompikownia.pl/index.php/2020/07/29/problem-komiwojazera-rozwiazywany-algorytmem-genetycznym/
- http://algorytmy.ency.pl/tutorial/problem_komiwojazera_algorytm_genetyczny
- https://www.aragorn.wi.pb.edu.pl/ wkwedlo/EA5.pdf