

Politechnika Wrocławска  
Wydział Elektroniki

# Projektowanie Efektywnych Algorytmów

## Projekt 3

Autor: Wojciech Wójcik 235621

Termin: wt. 18:55  
Prowadzący: dr inż. Jarosław Rudy

20 stycznia 2019

## **1. Wstęp**

Zadanie projektowe polegało na zaimplementowaniu oraz dokonaniu analizy efektywności algorytmu genetycznego dla problemów:

- Komiwojażera (ang. *Traveling Salesman Problem- TSP*),
- Szeregowania zadań (ang. *Scheduling Problem- SP*)

W ramach projektu zajęto się problemem komiwojażera.

Zaimplementowanym w tym projekcie algorytmem jest *Algorytm Genetyczny*. Sposób działania algorytmu jest analogią do procesu biologicznej *ewolucji darwinowskiej*. Ewolucja ta, rozpatrywana jest na poziomie zbiorów osobników – *populacji*. Podlegają one oddziaływaniu ze środowiskiem naturalnym. Osobniki najlepiej do niego przystosowane mają największą szansę na przeżycie i wydanie potomstwa, które odziedziczy cechy rodziców. Oprócz cech poprawiających przystosowanie, u potomstwa mogą występować losowe zmiany, nazywane *mutacjami*, które mogą prowadzić do zmian ich przystosowania. Powyższe spostrzeżenia prowadzą do wniosku, że z czasem populacja będzie średnio rzecz biorąc coraz lepiej przystosowana. Jest to w pewnym stopniu proces optymalizacji miary przystosowania.

W celu wykorzystania procesu ewolucji w rozwiązywaniu problemów optymalizacyjnych, na podstawie wiedzy o problemie musi zostać skonstruowana *funkcja przystosowania* mająca na celu ocenę każdego osobnika pod względem jego jakości. *Osobnikiem* nazywany jest pewien zbiór cech (genów) i ich wartości, opisujących bezpośrednio rozwiązanie z *przestrzeni rozwiązań*. Obliczenie wartości funkcji przystosowania dla każdego osobnika pozwala na wybór najlepszych. Zostaną oni poddani reprodukcji, a ich dzieci stworzą nową populację. Przy rozmnażaniu ważna jest *selekcja* odpowiednich osobników do reprodukcji oraz sposób dziedziczenia cech rodziców przez potomstwo (*krzyżowanie*). Rodzice lub potomstwo poddawane jest również losowym mutacjom, co ma zapewnić różnorodność w populacji. Każdą nowo uformowaną populację nazywa się *generacją*.

## **2. Opis programu**

W zaimplementowanym algorytmie, poszukującym rozwiązanie problemu komiwojażera funkcją przystosowania jest funkcja wyliczająca koszty ścieżek wytworzonych przez algorytm. Im rozwiązanie ma mniejszy koszt, tym przystosowanie jest lepsze.

Kolejnym elementem jest sposób selekcji osobników do puli rodzicielskiej. Jak wcześniej napisano, ogólna zasada wskazuje, że większą szansę na bycie rodzicem mają jednostki o lepszym przystosowaniu. W algorytmie zostało to uzyskane poprzez wykorzystanie selekcji turniejowej. Polega ona na wyborze z populacji, do grupy turniejowej N osobników, gdzie N jest parametrem algorytmu. Spośród grupy turniejowej wybierany jest najlepszy uczestnik turnieju i wstawiany do zbioru przyszłych rodziców. Selekcja jest powtarzana aż do uzyskania puli rodziców równej rozmiarowi populacji.

Następnym krokiem jest przeprowadzenie krzyżowania, którego wynikiem będą 2 nowe osobniki zasilające kolejną populację. Krzyżowanie wykonywane jest z prawdopodobieństwem będącym parametrem algorytmu. Losowo wybrane pary, które nie będą się krzyżować, zostają skopiowane do nowej populacji bez zmian, pary będące krzyżowane generują 2 potomków którzy zasilią zbiór nowego pokolenia. W projekcie zaimplementowano łącznie dwa operatory krzyżowania działające na ścieżkowej reprezentacji rozwiązań. Pierwszy z nich to Partially-Mapped Crossover (PMX). Polega on na wyborze dwóch losowych pozycji w ścieżce. W ten sposób sekwencja zostaje podzielona na 3 części: lewą, środkową oraz prawą.

$$\begin{aligned} r1 &= (1 \ 2 \ | \ 3 \ 4 \ 5 \ 6 \ | \ 7 \ 8 \ 9) \\ r2 &= (5 \ 3 \ | \ 6 \ 7 \ 8 \ 1 \ | \ 2 \ 9 \ 4) \end{aligned}$$

Następnie tworzona jest mapa zmian wierzchołków w częściach środkowych ścieżek.

$$(6<=>3), (7<=>4), (8<=>5), (1<=>6)$$

Potem do fragmentu środkowego dziecka nr 1 wpisywana jest odpowiadająca część z rodzica nr 2 i odwrotnie.

$$\begin{aligned} d1 &= (\_ \_ \ | \ 6 \ 7 \ 8 \ 1 \ | \ \_ \ \_) \\ d2 &= (\_ \_ \ | \ 3 \ 4 \ 5 \ 6 \ | \ \_ \ \_) \end{aligned}$$

Do części lewej oraz prawej dziecka 1 wpisywane są wierzchołki z odpowiadających pozycji rodzica 1, tylko gdy nie tworzą konfliktu z częścią środkową.

$$\begin{aligned} d1 &= (\_ \ 2 \ | \ 6 \ 7 \ 8 \ 1 \ | \ \_ \ 9) \\ d2 &= (\_ \_ \ | \ 3 \ 4 \ 5 \ 6 \ | \ 2 \ 9 \ \_) \end{aligned}$$

Wierzchołki wchodzące w konflikt uzupełniane są z wykorzystaniem wcześniej sporządzonego mapowania wierzchołków, zachowując poprawny format ścieżki.

$$\begin{aligned} d1 &= (3 \ 2 \ | \ 6 \ 7 \ 8 \ 1 \ | \ 4 \ 5 \ 9) \\ d2 &= (8 \ 1 \ | \ 3 \ 4 \ 5 \ 6 \ | \ 2 \ 9 \ 7) \end{aligned}$$

W ten sposób uzyskuje się 2 osobniki będące potomkami rodziców r1 i r2.

Drugim sposobem jest zastosowanie operatora Non-Wrapping Order Crossover (NWOX). Na początku dzieci są dokładnymi kopiami rodziców.

$$\begin{aligned} r1 &= (1 \ 2 \ | \ 3 \ 4 \ 5 \ 6 \ | \ 7 \ 8 \ 9) \\ r2 &= (5 \ 3 \ | \ 6 \ 7 \ 8 \ 1 \ | \ 2 \ 9 \ 4) \end{aligned}$$

Tak jak w przypadku PMX, dokonuje się podziału sekwencji na 3 części. U dziecka 1 usuwane są wierzchołki wchodzące w konflikt z wierzchołkami ze środka dziecka 2 (rodzica 2) i odwrotnie.

$$\begin{aligned} d1 &= (\_ \ 2 \ | \ 3 \ 4 \ 5 \ \_ \ | \ \_ \ 9) \\ d2 &= (\_ \_ \ | \ \_ \ 7 \ 8 \ 1 \ | \ 2 \ 9 \ \_) \end{aligned}$$

Przy usuwaniu wierzchołków powstają „dziury”, należy przemieścić je wszystkie w obszar środkowy zamieniając ze sobą miejscami wartości na sąsiadujących pozycjach.

$$\begin{aligned} d1 &= (2 \ 3 \ | \ _ \ _ \ _ \ | \ 4 \ 5 \ 9) \\ d2 &= (7 \ 8 \ | \ _ \ _ \ _ \ | \ 1 \ 2 \ 9) \end{aligned}$$

Gdy wszystkie „dziury” znajdują się we fragmentach środkowych następuje skopiowanie tych fragmentów z rodzica 1 do dziecka 2 oraz z rodzica 2 do dziecka 1.

$$\begin{aligned} d1 &= (2 \ 3 \ | \ 6 \ 7 \ 8 \ 1 \ | \ 4 \ 5 \ 9) \\ d2 &= (7 \ 8 \ | \ 3 \ 4 \ 5 \ 6 \ | \ 1 \ 2 \ 9) \end{aligned}$$

Po stworzeniu nowej populacji, na każdym osobniku dokonać się może z pewnym prawdopodobieństwem mutacja.

Na potrzeby pisania programu zdefiniowano następujące metody:

- **Insert**

Wybierane są losowo dwie pozycje na danej ścieżce, z jednej pobiera się wierzchołek i wstawia na drugą

$$\text{Przykład: } (0,1,2,3,4,5,6,7,8,9) \Rightarrow (0,1,7,2,3,4,5,6,8,9)$$

^                    ^

- **Invert**

Wybierane są losowo dwie pozycje na danej ścieżce, tworząc podścieżkę. W następnym kroku odwracana jest kolejność tej podścieżki

$$\text{Przykład: } (0,1,2,3,4,5,6,7,8,9) \Rightarrow (0,1,7,6,5,4,3,2,8,9)$$

^                    ^

Po wykonaniu powyższych kroków, wyliczane są koszty rozwiązań z nowej populacji i porównywane z najlepszym do tej pory znalezionym rozwiązaniem, jeśli której okaże się lepsze, staje się ono wtedy rozwiązaniem najlepszym.

Procedura powtarzana jest zadaną liczbę iteracji lub przez określony czas. Generację 0 stanowią osobniki wygenerowane losowo.

### 3. Specyfikacja

W projekcie zaimplementowano kilka podstawowych klas, reprezentujących instancję problemu oraz jego wyniki.

Klasa *Graph* jest bezpośrednią reprezentacją instancji problemu w pamięci komputera, zawiera ona macierz odległości między wierzchołkami, przechowuje rozmiar problemu i nazwę pliku, z którego pochodzi. Możliwy jest odczyt instancji z pliku .txt, wypisanie zawartości macierzy oraz stworzenie losowego grafu o zadanych parametrach.

*Path* to klasa przechowująca tworzone w algorytmach ścieżki oraz ich koszty. Ciąg wierzchołków jest zapamiętywany w kontenerze *vector*. Umożliwione są akcje wyświetlenia, dodania i usunięcia elementów ścieżki. Usuwanie i dodawanie odbywa się na końcu wektora.

Obiekt klasy *Time* służy do pomiaru oraz przechowywania czasów wykonywania się algorytmów. Posiada trzy odrębne pola określające czas trwania algorytmu w sekundach, milisekundach i mikrosekundach. Metody Start oraz Stop służą do pomiaru czasu. Każde wywołanie funkcji Start rozpoczyna nowy pomiar, możliwe jest wielokrotne wywoływanie funkcji Stop w celu doliczania do wyniku kolejnych czasów.

Wszystkie parametry algorytmów, które miały nie być zapominane po zakończeniu algorytmu lub zmianie danych są przechowywane w obiekcie klasy *Global*. Są to: maksymalna liczba generacji algorytmu, maksymalny czas trwania algorytmu, operatory oraz współczynniki krzyżowania i mutacji, rozmiar turnieju, wielkość populacji. Klasa udostępnia funkcje zmiany oraz wyświetlania parametrów.

Algorytm został zaimplementowany w osobnych klasach, których uogólnieniem jest klasa *Algorithm*. Są w niej wyspecyfikowane pola przechowujące dotyczącą liczbę iteracji algorytmu. Ponadto, znajdują się tam trzy ścieżki, parametry globalne, reprezentacje problemu i miernik czasu. Konstruktor przyjmuje utworzone na zewnątrz obiekty z parametrami oraz reprezentacją problemu, w ten sposób zapewnione zostaje pamiętanie instancji oraz ustalonych wartości. Dodatkowo klasa posiada metody do wyliczania kosztów ścieżek, generowania pierwszych rozwiązań (losowe i ustalone jako rosnący ciąg wierzchołków) i wyświetlenia informacji o uzyskanych rezultatach.

Klasa *Genetic* zawiera implementacje algorytmu genetycznego. Dla zachowania porządku i ograniczenia odwołań do obiektu klasy *Global*, klasa *Genetic* przechowuje lokalne kopie kluczowych parametrów, jak np. rozmiar populacji czy wielkość turnieju. Ponadto, istnieje zmienna przechowująca średni koszt aktualnej generacji oraz zbiory ścieżek reprezentujące populacje lub jej części np. tablica ścieżek Population, Tournament, Offspring czy Parents. W klasie zostały zdefiniowane funkcje opisane w poprzednim rozdziale, są to między innymi: Solve(), TournamentSelection(), Mutation(), GetNewCosts(Path \*\* Set).

Klasę Menu stanowi zbiór funkcji realizujących interakcję z użytkownikiem. Są tam funkcje głównego menu, testów ręcznych, ustawień grafu oraz testów efektywności. W klasie istnieją również pola na instancję problemu (Graph) oraz parametry globalne (Global).

## 4. Plan eksperymentów

Badanie efektywności zaimplementowanego algorytmu polegało na zbadaniu zależności wielkości populacji na otrzymywane wyniki, sprawdzaniu wpływu prawdopodobieństw krzyżowania i mutacji, jak i również samych operatorów na rezultaty.

Algorytm testowano dla trzech istotnie różnych problemów (instancji), tych samych co dla algorytmów SA i TS. Uzyskane wyniki konfrontowano z najlepszym znany dla tego problemu rozwiązaniem oraz z czasami wykonywania się tych algorytmów.

Mając za punkt odniesienia najlepsze znane rozwiązania, obliczano błędy względne otrzymanych wyników. Przykładowe wyniki algorytmów przedstawiano na wykresach ukazujących przebieg błędu w funkcji czasu wykonywania się algorytmu. Na wykresach błędu względnego prezentowano przebiegi o możliwie zbalansowanym stosunku czasów wykonania do uzyskiwanych przy danych parametrach wartościach błędu.

Testy przeprowadzono stu-krotnie w celu wyciągnięcia średniej. Po poznaniu średniej dla danej konfiguracji parametrów, wywoływano algorytm do momentu aż nie zostanie uzyskany wynik możliwie jak najbliższy średniemu. Przebieg takiego przypadku był umieszczany na wykresie, a wynik zapisywany w tabeli. Dodatkowo, korzystając z wiedzy nabytej po właściwych testach,

sprawdzono kilka innych rozkładów parametrów. Najlepsze uzyskane wyniki umieszczone w odpowiedniej tabeli i sporządzono osobne wykresy.

Instancje problemu komiwojażera na których testowano zaimplementowane algorytmy to:

- Ftv38 (tsp\_39)
- Ftv70 (tsp\_71)
- Ftv170 (tsp\_171)

## 5. Wyniki pomiarów

Rezultaty eksperymentów zaprezentowano w poniższych tabelach.

		Wyniki Średnie - manipulacja wielkością populacji									
		INSERT					INVERT				
		N = 50		N = 300		N = 1000		N = 50		N = 300	
PMX	M_p=0.01	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
	TSP_39	1000.01	15.69%	1000.60	10.85%	1001.45	10.13%	1000.03	62.48%	1000.24	59.15%
	TSP_71	1500.05	59.95%	1500.11	61.13%	1501.31	61.44%	1500.16	142.97%	1500.08	109.74%
NWOWX	C_p=0.8	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
	TSP_39	5000.01	184.83%	5000.38	182.98%	5003.20	166.97%	5000.15	355.25%	5000.75	345.84%
	TSP_71	5000.18	160.80%	5000.31	152.56%	4999.35	153.58%	5000.05	329.69%	5000.03	315.72%
NWOWX	T_size=5	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
	TSP_39	1000.04	30.92%	1000.31	21.57%	1001.63	16.73%	1000.06	67.39%	1000.06	47.06%
	TSP_71	1500.05	58.41%	1500.05	46.87%	1502.18	46.67%	1500.10	152.87%	1500.31	139.03%
NWOWX	N = 1000	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
	TSP_39	1000.49	40.52%	1000.06	47.06%	1000.06	47.06%	1000.06	47.06%	1000.49	40.52%
	TSP_71	1500.29	89.18%	1500.31	139.03%	1500.31	139.03%	1500.31	139.03%	1500.29	89.18%
NWOWX	TSP_171	4998.49	359.38%	4998.06	301.67%	4998.06	301.67%	4998.06	301.67%	4998.06	301.67%

		Wyniki Średnie - manipulacja współczynnikiem mutacji													
		INSERT					INVERT								
		N=300		M_p=0.01		M_p=0.05		M_p=0.1		M_p=0.01		M_p=0.05		M_p=0.1	
PMX	C_p=0.8	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%	T [ms]	%
	PMX	TSP_39	292.98	26.21%	308.06	17.97%	301.50	19.87%	306.33	57.71%	289.84	52.88%	303.79	54.38%	
	NWOWX	TSP_71	533.45	63.54%	936.02	40.62%	580.85	41.54%	744.75	126.87%	1035.02	119.90%	881.29	118.97%	
NWOWX	T_size=5	TSP_171	3184.28	152.16%	3091.57	113.65%	2742.09	101.27%	3229.11	328.64%	3139.05	285.26%	3139.05	285.26%	

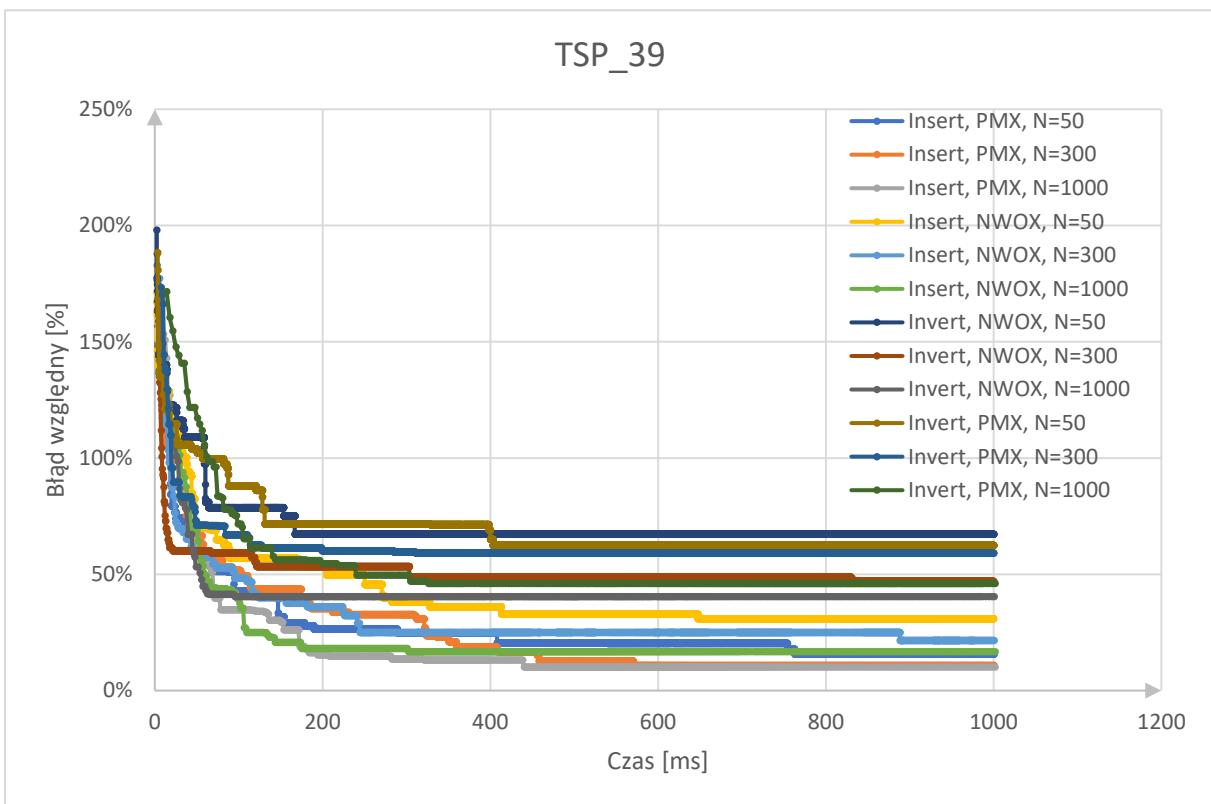
		Wyniki Średnie - manipulacja współczynnikiem krzyżowania														
		PMX					NWOWX									
		N=300		PMX		NWOWX		PMX		NWOWX						
PMX	M_p=0.01	C_p=0.01	T [ms]	%	C_p=0.05	T [ms]	%	C_p=0.1	T [ms]	%	C_p=0.01	T [ms]	%	C_p=0.05	T [ms]	%
	PMX	TSP_39	605.69	26.86%	605.69	27.39%	605.69	27.06%	605.69	25.88%	605.69	24.64%	605.69	24.38%		
	NWOWX	TSP_71	863.35	69.44%	863.35	68.72%	863.35	69.28%	863.35	65.95%	863.35	64.56%	863.35	61.79%		
NWOWX	T_size=5	TSP_171	3139.05	170.74%	3139.05	166.72%	3139.05	177.89%	3139.05	161.56%	3139.05	155.61%	3139.05	154.52%		

m_p=0.2	c_p=0.7	Wyniki minimalne		
NWOX	INSERT	%	T [ms]	GEN
N=300	TSP_39	3.79%	2326.68	3603.00
T_size=5	TSP_71	17.49%	5143.646	8986.00
	TSP_171	69.66%	10000.06	8830.00

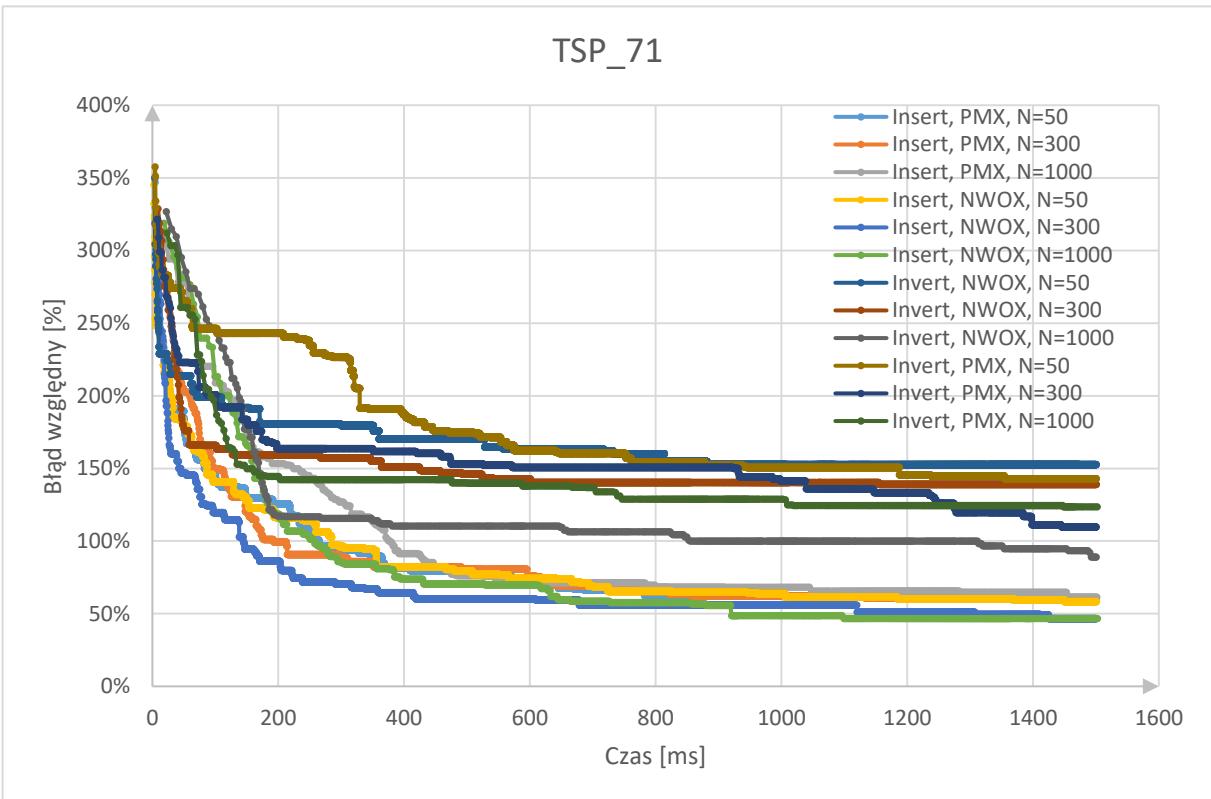
Gdzie: M\_p – prawdopodobieństwo mutacji, C\_p – prawdopodobieństwo krzyżowania, T\_size – rozmiar turnieju, N – wielkość populacji

Uwaga: ciemne pola w lewym górnym rogu tabel określają parametry ustalone dla wszystkich danych z tabeli

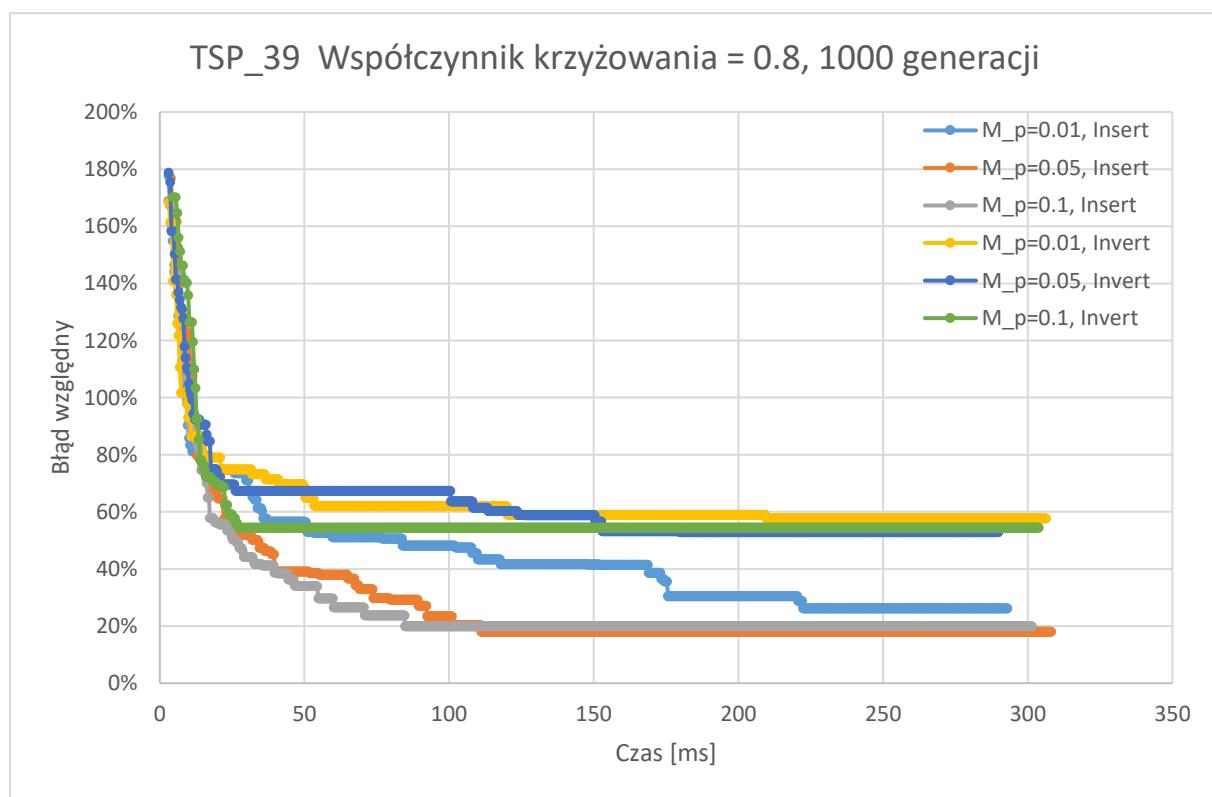
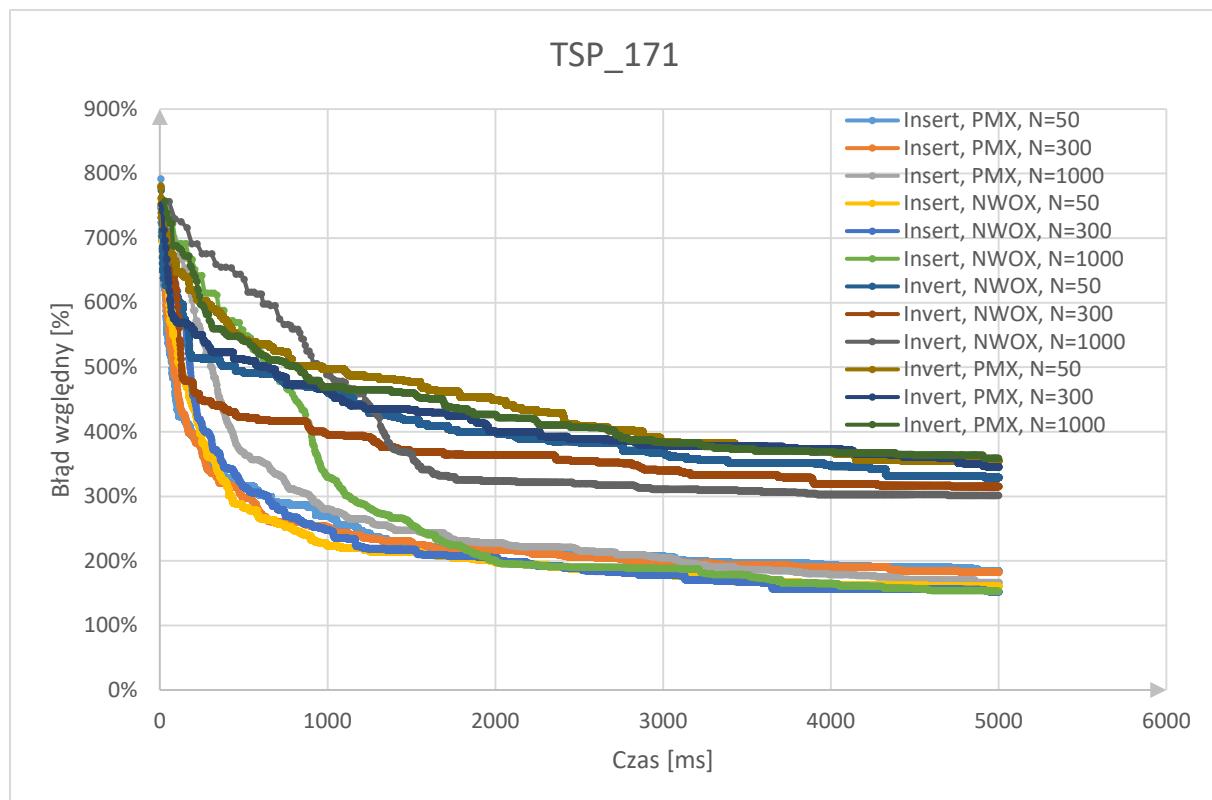
Na kolejnych stronach przedstawiono wykresy porównujące ze sobą wyniki poszczególnych algorytmów.



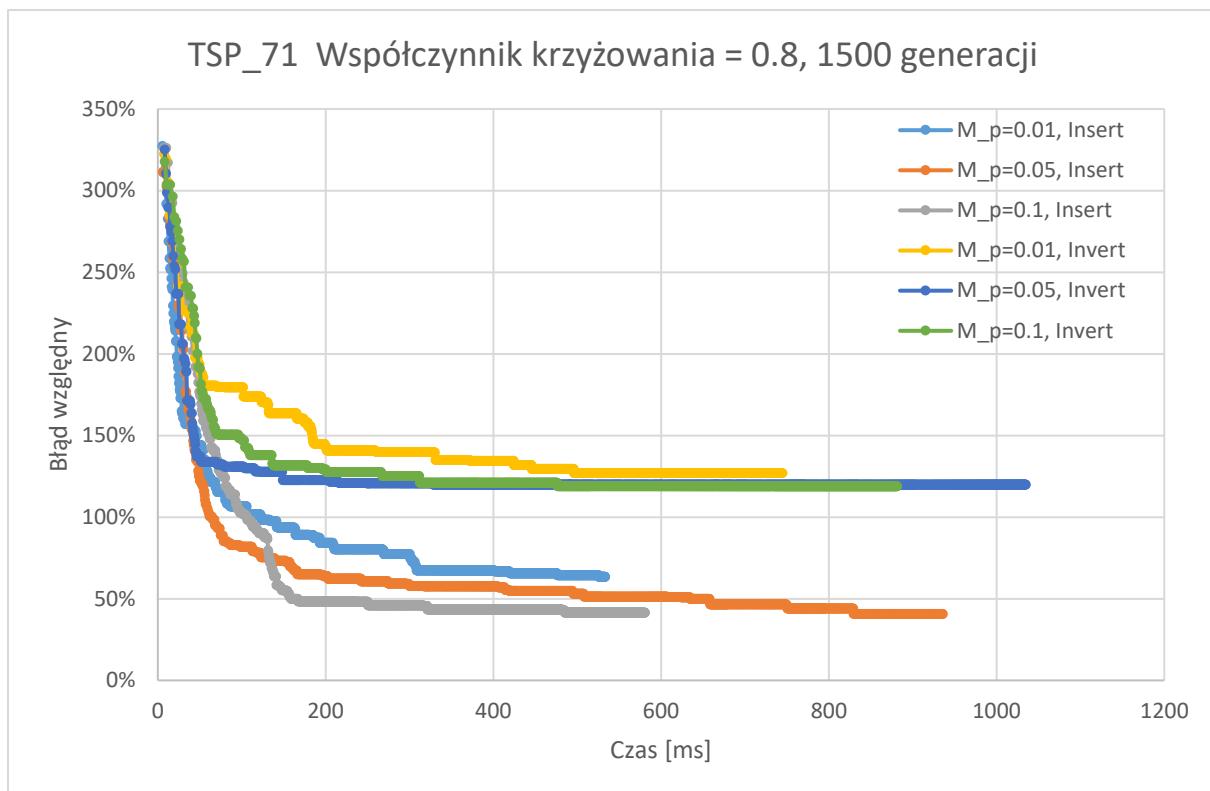
Rysunek 1. Manipulacja wielkością populacji



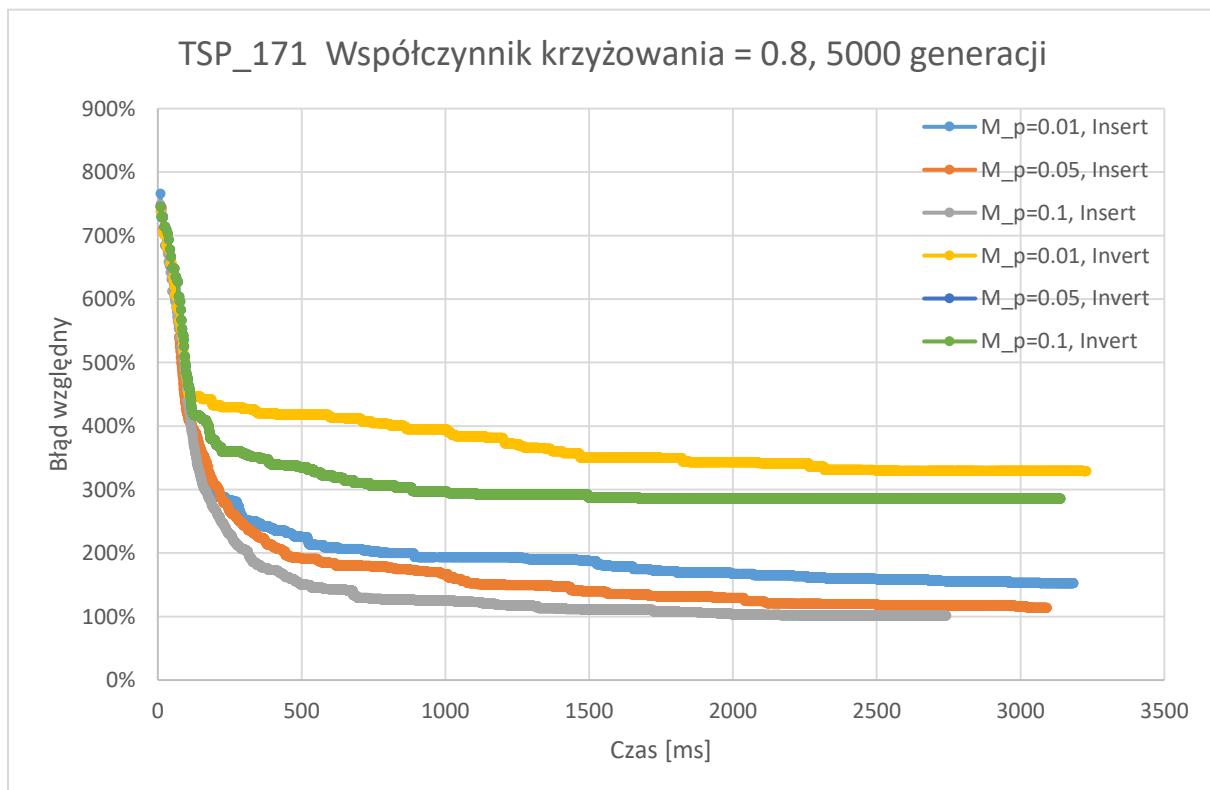
Rysunek 2. Manipulacja wielkością populacji



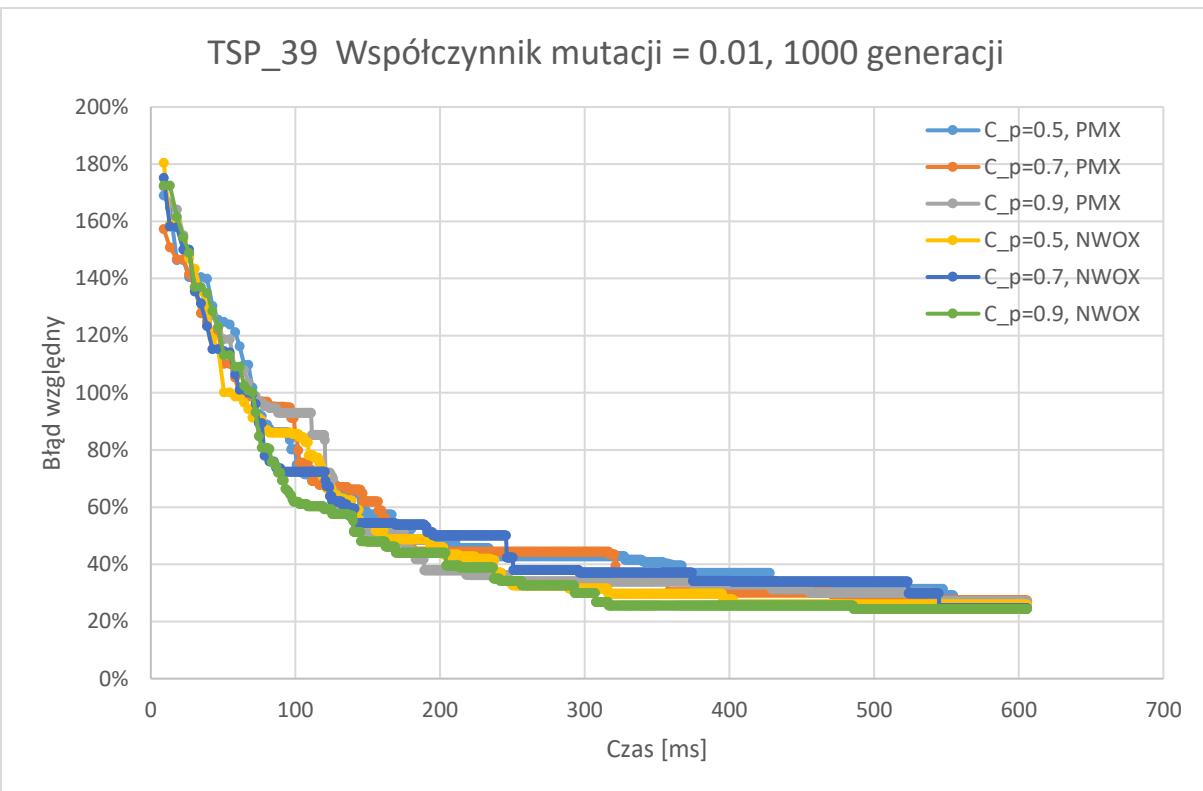
Rysunek 3. Manipulacja współczynnikiem mutacji



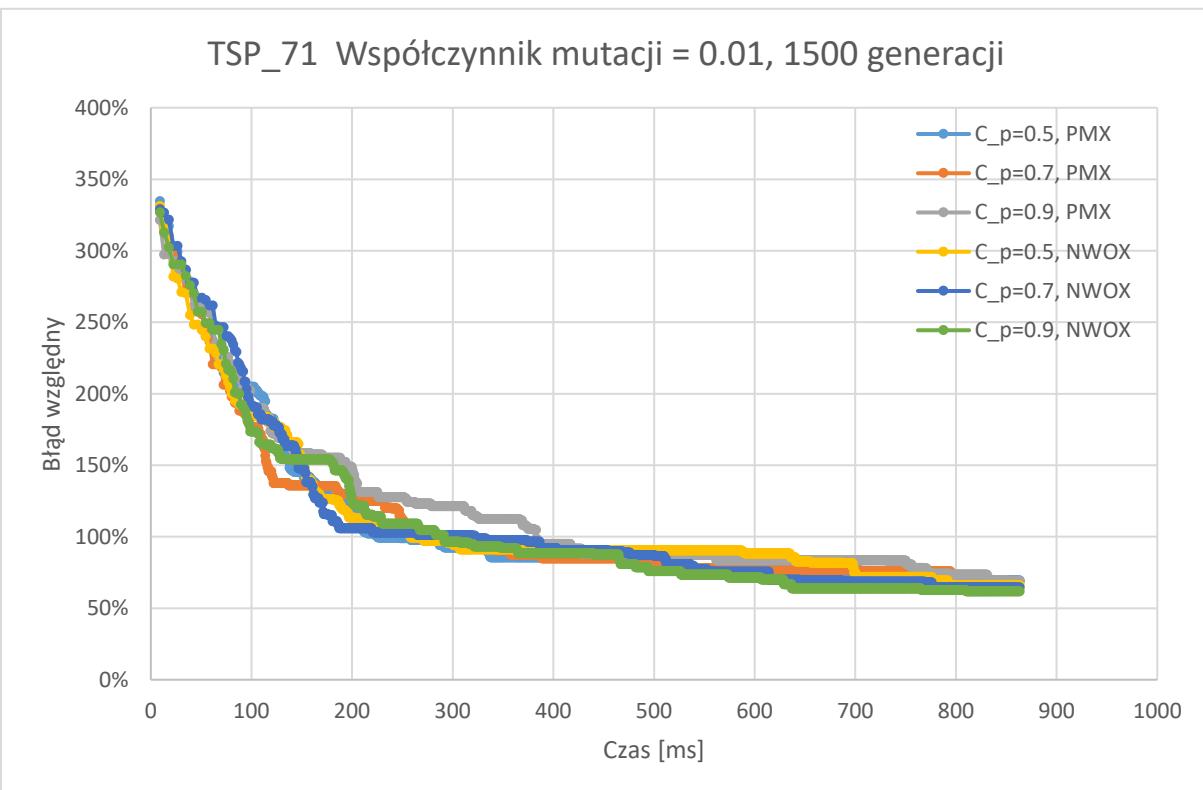
Rysunek 5. Manipulacja współczynnikiem mutacji



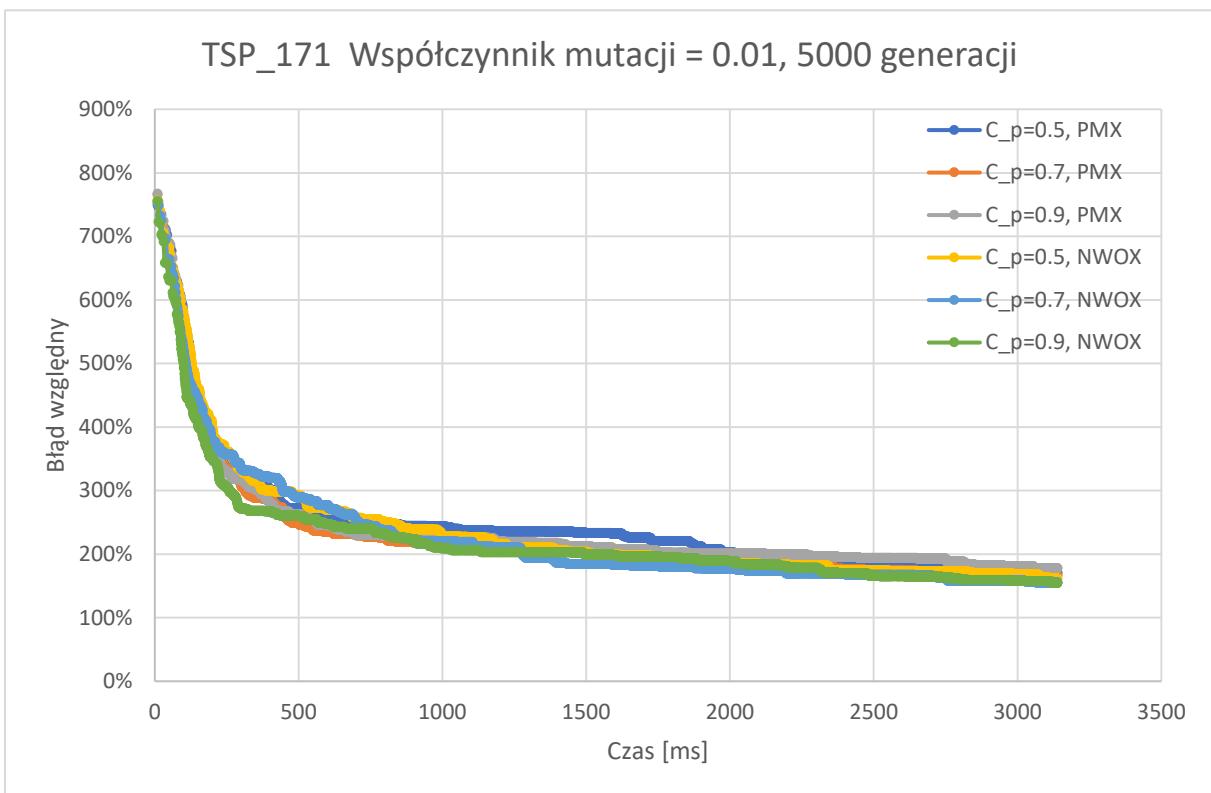
Rysunek 6. Manipulacja współczynnikiem mutacji



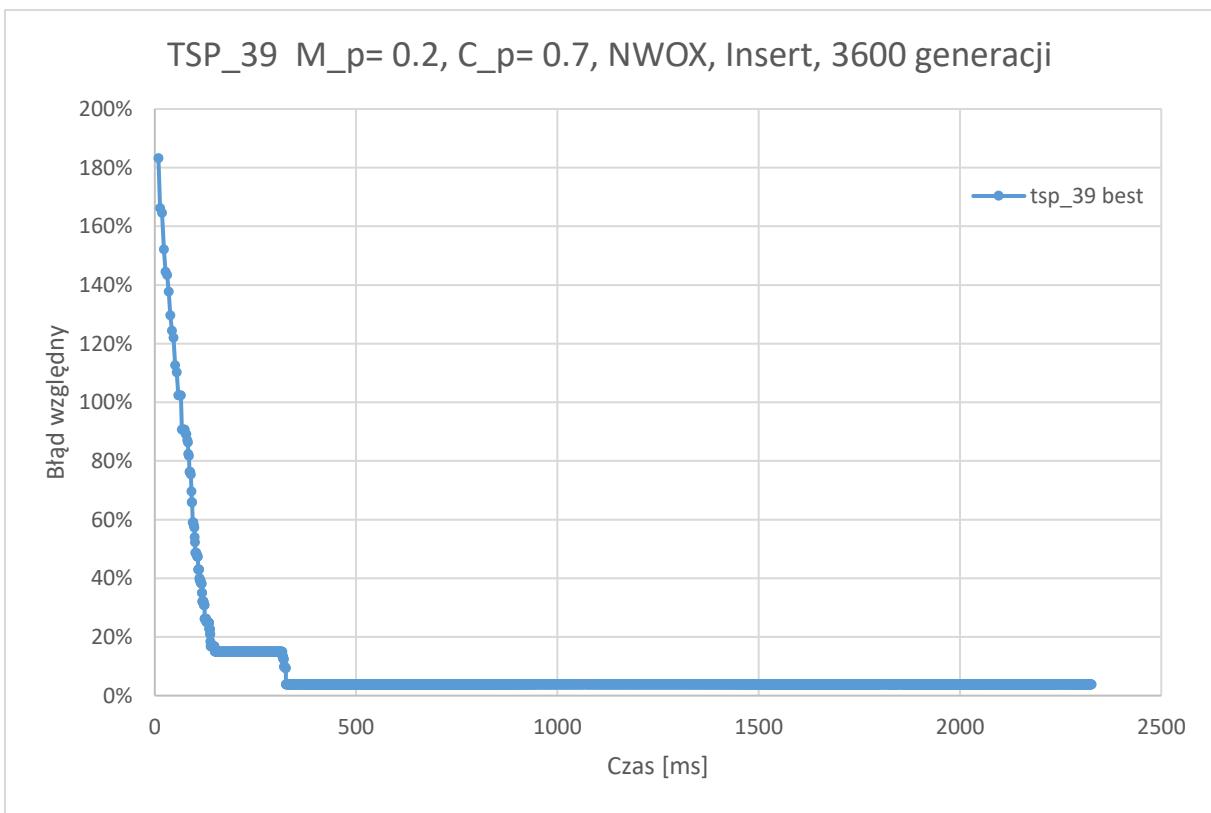
Rysunek 7. Manipulacja współczynnikiem krzyżowania



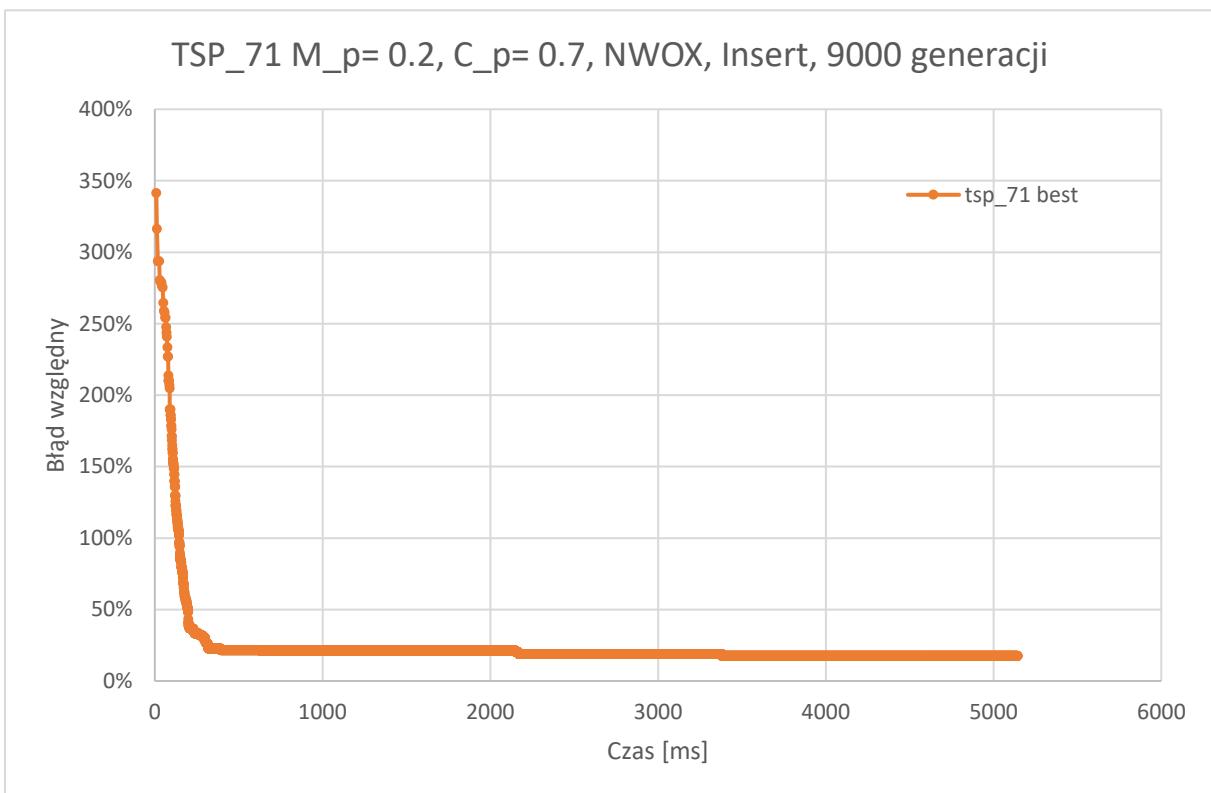
Rysunek 8. Manipulacja współczynnikiem krzyżowania



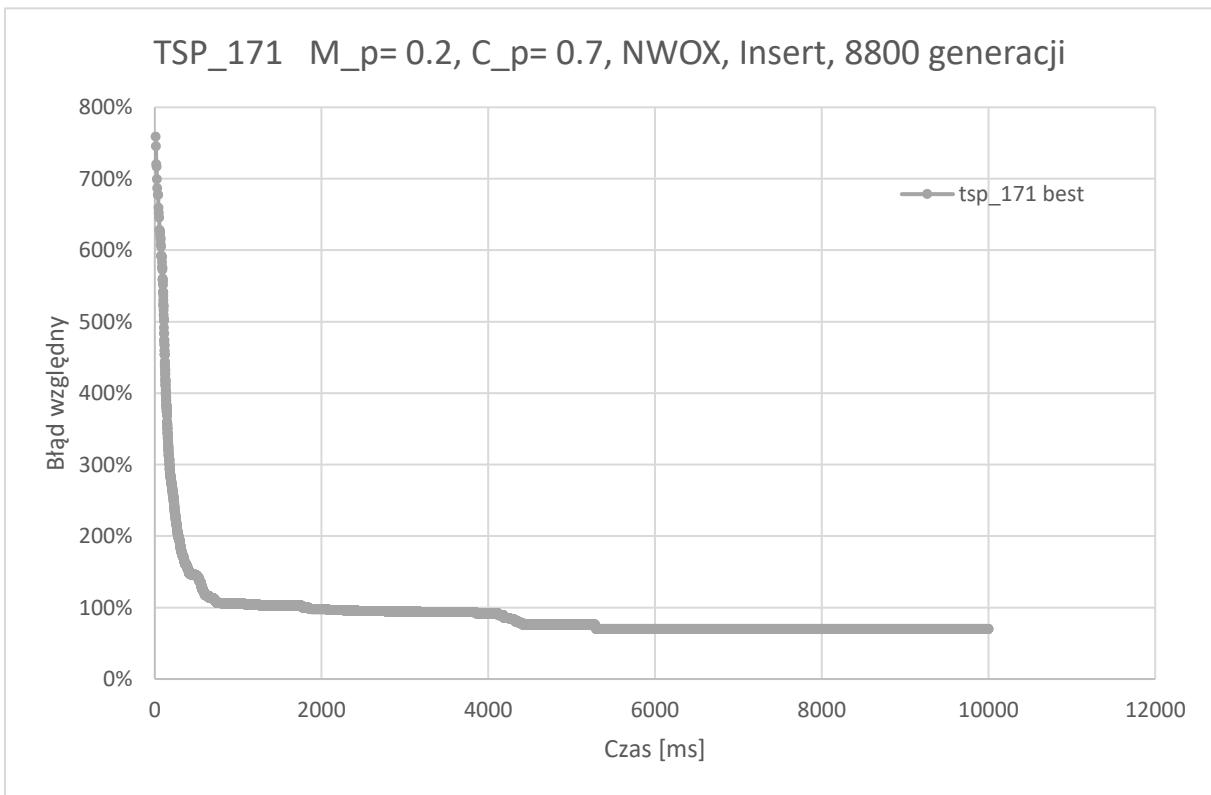
Rysunek 10. Manipulacja współczynnikiem krzyżowania



Rysunek 9. Najlepszy uzyskany wynik



Rysunek 12. Najlepszy uzyskany wynik



Rysunek 11. Najlepszy uzyskany wynik

## 6. Wnioski

Zaimplementowany algorytm daje w wyniku trasy z kosztami z błędami na akceptowalnym poziomie, w dobrym czasie.

Analizując wyniki, łatwo zauważyc, że algorytm genetyczny ma skłonności do utykania w okolicach minimów lokalnych. Jest to spowodowane najprawdopodobniej przedwczesną zbieżnością, zbyt dużym podobieństwem do siebie osobników w populacji na pewnym etapie. W takim wypadku pomóc może jedynie sprzyjająca mutacja, która występuje zwykle rzadko. Algorytm można by wzbogacić o lepszą strategię selekcji, z implementacją elity lub elity lokalnej.

Dla testowanych wielkości populacji można wywnioskować, że im większa populacja tym lepsze wyniki. Odbija się to jednak znaczco na długości obliczeń. Operatory krzyżowania okazały się mieć podobne właściwości. Uzyskane przy użyciu obu metod wyniki były porównywalne. Jednak tym operatorem, który pozwolił na uzyskanie najlepszych ścieżek, był NWOX. Jego istotnymi cechami jest zachowanie względnej kolejności wierzchołków w ścieżkach oraz mniej wymagająca implementacja. Operator mutacji Insert okazał się zdecydowanie lepszy od operatora Invert. Każda testowana konfiguracja z mutacją typu invert osiągała wyniki dwukrotnie lub więcej, gorsze. Największy wpływ na jakość wyników miało prawdopodobieństwo zajścia mutacji. W każdym przypadku testowym, obejmującym zmianę tego parametru, zaobserwowano polepszenie wyników wraz ze zwiększaniem prawdopodobieństwa mutacji. Oznacza to, że algorytm często mutujący, ma możliwość efektywniejszego przeszukiwania wielu obszarów przestrzeni rozwiązań, ponieważ losowe mutacje pozwalają na nieznaczne, lecz istotne zmiany jednostek. Algorytm może wydostać się wtedy z minimum lokalnego i szukać dalej. Trzeba jednak mieć na uwadze, że zbyt duże prawdopodobieństwo mutacji spowoduje dużą losowość w algorytmie, co upodobni go do *random search*. Manipulowanie współczynnikiem krzyżowania nie dawało zauważalnych zmian w charakterze rozwiązań. Niezależnie od użytego operatora i prawdopodobieństwa, wyniki poszczególnych testów były do siebie bardzo podobne.

Przetestowane algorytmy charakteryzuje się dużą elastycznością. Dzięki wielu parametrom i zastosowaniu odpowiednich warunków końcowych i strategii można osiągać wyniki o zadowalającej dokładności, szczególnie dla instancji o rozmiarach poniżej 100. Algorytm genetyczny, dzięki operatorowi mutacji oraz obecności populacji jest w stanie przeszukiwać wiele rejonów przestrzeni rozwiązań. Niezwykle pomysłowa metoda, wzorowana na naturalnym procesie biologicznym okazuje się być dobrym rozwiązaniem w poszukiwaniu algorytmów rozwiązujących problem komiwojażera.

## **7. Literatura**

- Vincent A. Cicirello, Non-Wrapping Order Crossover: An Order Preserving
- JORGE MAGALHÃES-MENDES, A Comparative Study of Crossover Operators for Genetic Algorithms to Solve the Job Shop Scheduling Problem
- Otman ABDOUN, Jaafar ABOUCHABA KA, Chakir TAJANI, Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem
- ABDOUN Otman, ABOUCHABA KA Jaafar, A Comparative Study of Adaptive Crossover Operators for Genetic Algorithms to Resolve the Traveling Salesman Problem
- S.T. Wierzchoń, Sztuczne systemy immunologiczne. Teoria I zastosowania
- [http://www.zio.iiar.pwr.wroc.pl/pea/w9\\_ga\\_tsp.pdf](http://www.zio.iiar.pwr.wroc.pl/pea/w9_ga_tsp.pdf)
- <http://kolos.math.uni.lodz.pl/~archive/Sztuczna%20inteligencja/6%20Klasyczny%20algorytm%20genetyczny%20cz1.pdf>