Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 3

Implementacja i analiza efektywności algorytmu genetycznego dla problemu komiwojażera

Piątek, 11:15

Bartosz Szymczak
252 734

prowadzący
Dr inż. Jarosław Mierzwa

Informatyka Techniczna
Wydział Informatyki i Telekomunikacji

22 stycznia 2022

Spis treści

1 Wstęp teoretyczny

1.1 Podstawowe założenia

Podczas realizacji zadania należy przyjąć następujące założenia:

- używane struktury danych powinny być alokowane dynamicznie,
- program powinien umożliwić wczytanie danych testowych z plików: ftv47.atsp, ftv170.atsp, rgb403.atsp ze strony http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/atsp/,
- program musi umożliwiać wprowadzenia kryterium stopu jako czasu wykonania podawanego w sekundach,
- implementacje algorytmów należy dokonać zgodnie z obiektowym paradygmatem programowania,
- kod źródłowy powinien być komentowany.

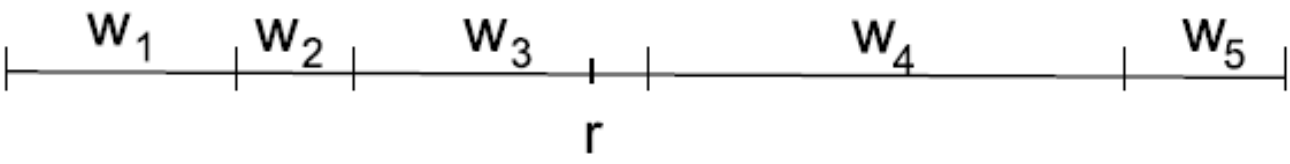
1.2 Opis algorytmu genetycznego

1.2.1 Idea algorytmu genetycznego

Algorytmy genetyczne odwzorowują naturalne procesy ewolucyjne, zachodzące w czasie, których celem jest maksymalne dopasowanie osobników do istniejących warunków życia. Są one procedurami przeszukiwania, opartymi na mechanizmach doboru naturalnego i dziedziczenia. Korzystają z ewolucyjnej zasady, przeżycia osobników najlepiej przystosowanych. W rozwiązywaniu zadań algorytm ten stosuje zasady ewolucji i dziedziczności, posługuje się populacją potencjalnych rozwiązań, zawiera proces selekcji, oparty na dopasowaniu osobników i pewne operatory genetyczne. Każde rozwiązanie ocenia się na podstawie miary jego dopasowania (współczynnik fitness). Nową populację tworzy się przez selekcję osobników najlepiej dopasowanych.

1.2.2 Metoda selekcji

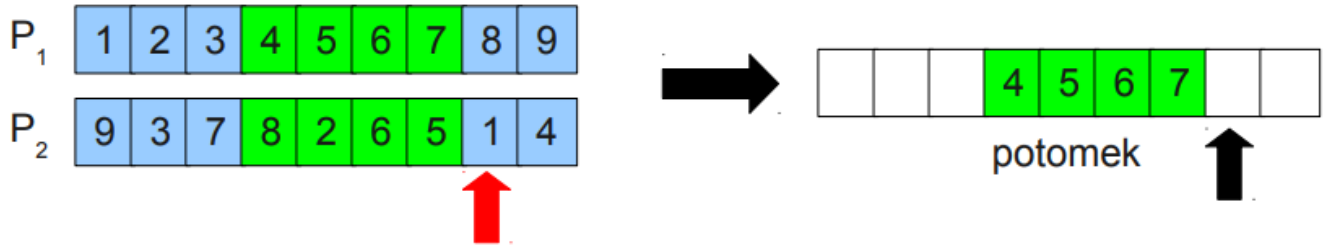
Zastosowano prosty wariant metody ruletkowej. W myśl strategii, że lepsze rozwiązanie posiada większą miarę dopasowania, następuje losowanie rozwiązań reprezentowanych przez wspomniane miary. W zaimplementowanym algorytmie wartości miary dopasowania zostają znormalizowane, w celu losowania wartości z zakresu od 0 do 1.



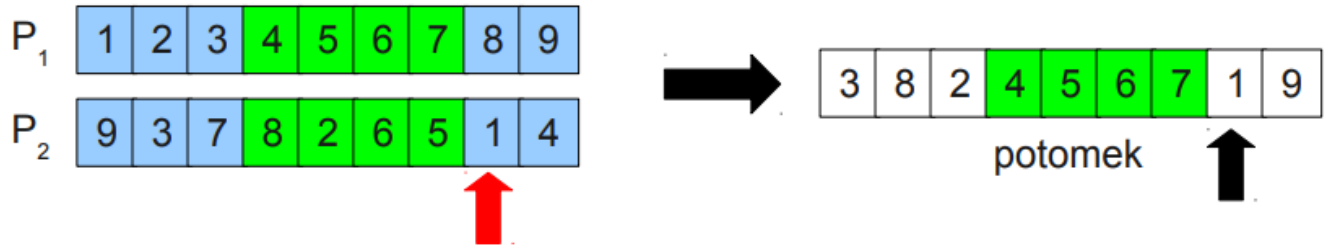
Rysunek 1: Metoda ruletki (r - wylosowana wartość, w - miara dopasowania elementu populacji)

1.2.3 Metoda krzyżowania

Zastosowano metodę Order Crossover (OX). W krzyżowaniu biorą udział dwa osobniki rodzicielskie, a rezultatem krzyżowania jest jeden potomek. Główna idea metody polega na wylosowaniu dwóch indeksów, określających początek i koniec części pierwszego rodzica. Następnie część pierwszego rodzica zostaje przekopiowana na takie samo miejsce do potomka. Dalej kopiowane są wartości z drugiego rodzica do potomka, o ile dane wartości nie występują w skopiowanej z pierwszego rodzica części.



Rysunek 2: Pierwszy etap OX - kopiowanie części pierwszego rodzica do potomka



Rysunek 3: Drugi etap OX - wypełnienie potomka niepowtórzonymi indeksami rodzica drugiego

1.2.4 Metoda mutacji

Zastosowano metody swap oraz insert. Mutacja elementu populacji jest wykonywana dla każdego elementu na losowych, różnych indeksach, z pewnym prawdopodobieństwem, za pomocą wybranej metody.

Metoda swap zamienia miejscami miasta o podanych różnych indeksach w podanej ścieżce.

Rysunek 4: Przykładowy przebieg metody swap

Metoda insert usuwa element z pierwszego podanego indeksu i wpisuje go na drugi podany indeks w podanej ścieżce.

Rysunek 5: Przykładowy przebieg metody insert

2 Implementacja algorytmu

2.1 Klasa GeneticAlghoritm

Klasa GeneticAlghoritm zawiera metody odpowiedzialne za przeprowadzenie algorytmu genetycznego na podanych danych. Posiada strukturę elementów populacji, przechowującą jego ścieżkę, koszt i fitness. Polami klasy są:

- vector<vector<int>> temp - Macierz miast i tras pomiędzy nimi, nad którą będą przeprowadzane obliczenia,
- vector<PopulationElement> population - Wektor elementów populacji,
- PopulationElement best - Zmienna przechowującą element populacji, zawierający najkrótszą, wyznaczoną przez algorytm ścieżkę - globalne rozwiązanie,
- int mutMethod - Zmienna przechowująca sposób mutacji,
- int size - Zmienna przechowująca ilość miast w macierzy,
- int populationSize - Zmienna przechowująca rozmiar populacji,
- float stopCriterion - Zmienna przechowująca kryterium stopu,
- float crossoverRate - Zmienna przechowująca współczynnik krzyżowania,
- float mutationRate - Zmienna przechowująca współczynnik mutacji,

Metodami klasy są:

- void doGA() - Metoda główna, odpowiedzialna za przeprowadzenie algorytmu genetycznego, z uwzględnieniem podanych w konstruktorze danych. Początkowo wywołuje metodę makePopulation. Następnie tworzy kopię populacji początkowej, która będzie aktualizowana metodą nextGeneration. Jej główna pętla trwa przez czas ograniczony współczynnikiem stopu i kolejno wywołuje metody countElements, nextGeneration, po czym zamienia populację początkową z jej kopią. Na koniec metoda wypisuje najlepsze rozwiązanie,
- vector<int> pickPath() - Metoda pomocnicza, zwraca ścieżkę z populacji na podstawie metody ruletkowej. Polega na wylosowaniu liczby od 0 do 1, po czym losuje wartość przetrwania elementu populacji, poprzez odejmowanie miar dopasowania od wylosowanej liczby i jednoczesnym zwiększaniu rozpatrywanego indeksu (będzie indeksem wyznaczonego elementu populacji),
- vector<int> crossOver(vector<int> pathA, vector<int>pathB) - Metoda pomocnicza, zwraca zmodyfikowana genetycznie ścieżkę, używając algorytmu Order crossover (OX, Davis, 1985). Ścieżka zostaje stworzona przy użyciu ścieżek podanych w argumencie. Początkowo wylosowane zostają różne indeksy, odpowiadające przedziałowi pierwszego rodzica do przekopiowania. Dalej dany fragment zostaje skopiowany do potomka. Następnie ustawia się indeksy ścieżki potomnej i drugiego rodzica. W głównej pętli, trwającej do momentu wypełnienia ścieżki potomnej, odbywają się kolejne czynności. Pierwsza z nich to sprawdzenie skopiowanej części pod kątem występowania wartości, wskazywanej przez indeks drugiego rodzica. Jeśli wartość nie została znaleziona, to zostaje wpisana do potomka w miejsce indeksu ścieżki potomnej, po tym indeksy są aktualizowane,

- void makePopulation(int n) - Metoda pomocnicza tworząca populacje o podanej wielkości. Tworzy wektor, wypełnia go kolejnymi indeksami miast, modyfikuje kolejność w sposób losowy oraz dodaje ją do populacji. Modyfikacja i dodanie odbywa się n razy.
- void countElements() - Metoda pomocnicza, uzupełnia wartości elementów populacji oraz aktualizuje globalne rozwiązanie. Dla każdego elementu populacji, obliczany jest koszt ścieżki oraz współczynnik przystosowania, który jest równy odwrotności kosztu ścieżki, podniesionego do potęgi. Dalej oblicza się sumę miar przystosowania. Potem porównywane są koszta obecnie rozpatrywanej ścieżki i globalnie najlepszej, jeśli ta pierwsza jest mniejsza to staje się nowym rozwiązaniem. Na koniec następuje normalizacja współczynników przystosowania.
- void nextGeneration(vector<PopulationElement>& newPopulation) - Metoda pomocnicza, modyfikuje podaną w argumencie populacje, używając krzyżowania oraz mutacji. Dla każdego elementu populacji, wybierane są dwie ścieżki przy użyciu metody pickPath, po czym zostają skrzyżowane metodą crossOver oraz zmutowane metodą mutate, jeśli oba współczynniki są większe niż wylosowana wartość,
- void mutate(vector<int>& path) - Metoda pomocnicza, dokonuje mutacji metodą swap lub insert wylosowanych indeksów ścieżki podanej w argumencie,
- void swap(vector<int>& path, int town1, int town2) - Metoda pomocnicza, zamienia miejscami miasta o podanych indeksach w podanej sciezce,
- void insert(vector<int>& path,int indexToDelete, int indexToPut) - Metoda pomocnicza, zmienia miejsce miasta o indeksie indexToDelete, na miejsce indexToPut, w scieżce path,
- int countPath(vector<int> path) - Zmienna pomocnicza zwracająca koszt podanej w argumencie ścieżki,

3 Eksperymenty

3.1 Plan eksperymentu

- Testom poddano opisany wyżej algorytm,
- Pomiary wykonywane były dla problemów o rozmiarze 47, 170 i 403 miast dla dwóch metod mutacji i różnych współczynników krzyżowania i rozmiarów populacji,
- Pomiary czasu zostały wykonane przy użyciu biblioteki std::chrono

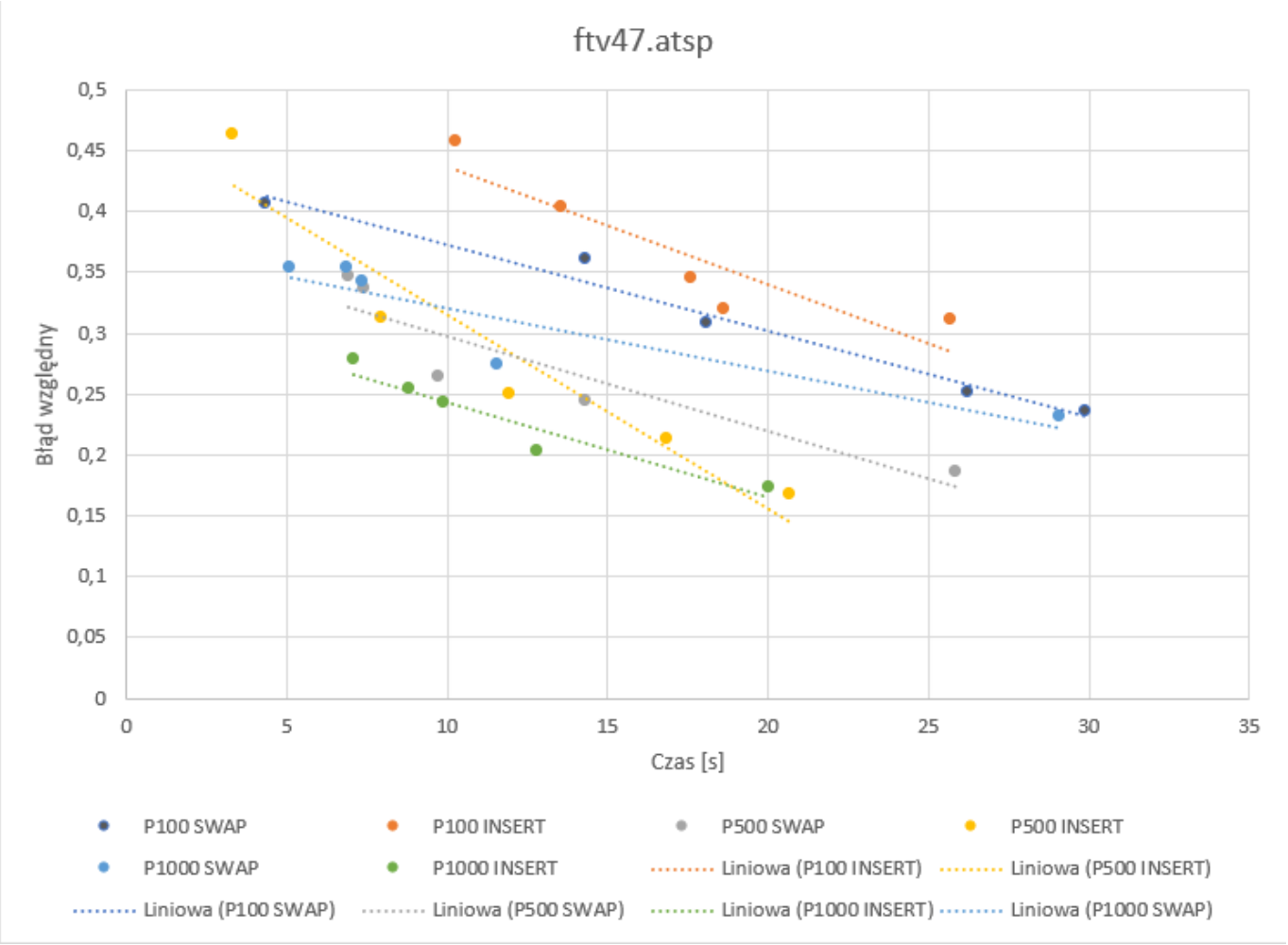
Podczas testu przyjęto parametry:

- Rozmiar populacji = 100/500/1000,
- Współczynnik mutacji = 0.01,
- Współczynnik krzyżowania = 0.5/0.7/0.8/0.9,
- Kryterium stopu = 30 sekund,
- Metoda mutacji = 0/1.

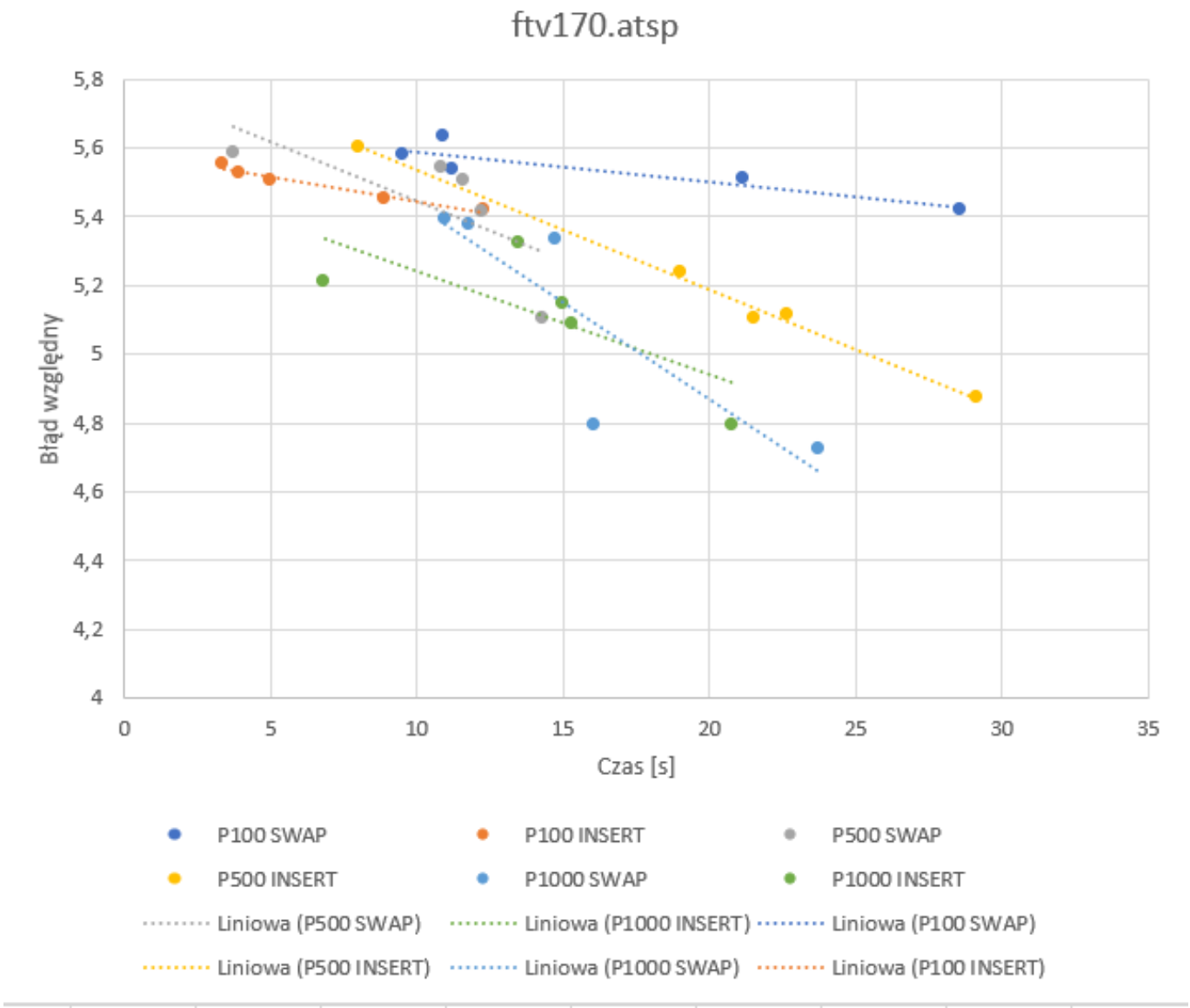
3.2 Zmienny rozmiar populacji

Rozmiar	Populacja	Mutacja	Czas	Błąd
47	100	SWAP	18,6 s	0,308
47	100	INSERT	17,1 s	0,356
47	500	SWAP	12,86 s	0,289
47	500	INSERT	12,16 s	0,355
47	1000	SWAP	12,08 s	0,262
47	1000	INSERT	11,72 s	0,252
170	100	SWAP	16,28 s	5,53
170	100	INSERT	6,72 s	5,49
170	500	SWAP	10,53 s	5,44
170	500	INSERT	20.09 s	5,28
170	1000	SWAP	15,47 s	5,12
170	1000	INSERT	14,28 s	5,18
403	100	SWAP	20,25 s	1,742
403	100	INSERT	12,66 s	1,712
403	500	SWAP	12,82 s	1,72
403	500	INSERT	15,57 s	1,74
403	1000	SWAP	13,12 s	1,746
403	1000	INSERT	14,71 s	1,746

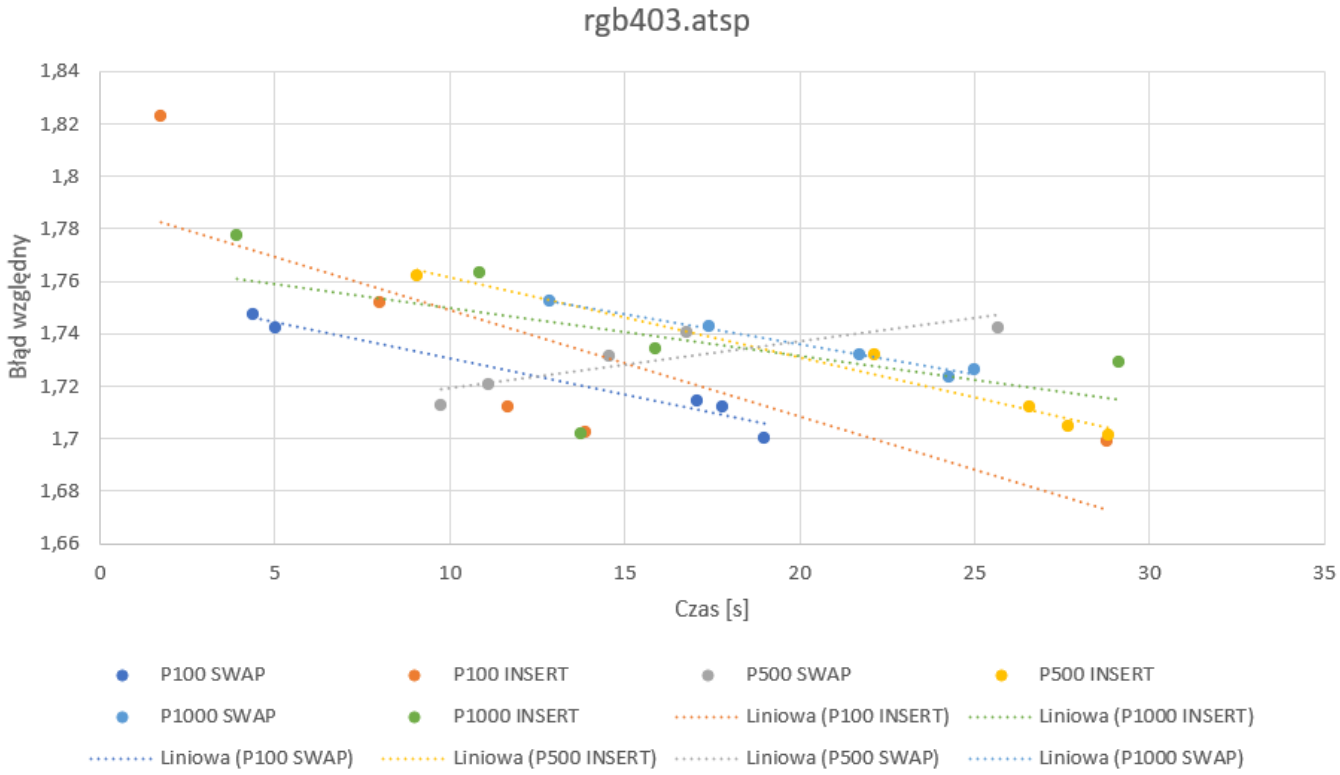
Tabela 1: Średnie pomiary dla różnych rozmiarów populacji



Rysunek 6: Pomiary dla różnych rozmiarów populacji dla pliku ftv47.atsp



Rysunek 7: Pomiary dla różnych rozmiarów populacji dla pliku ftv170.atsp

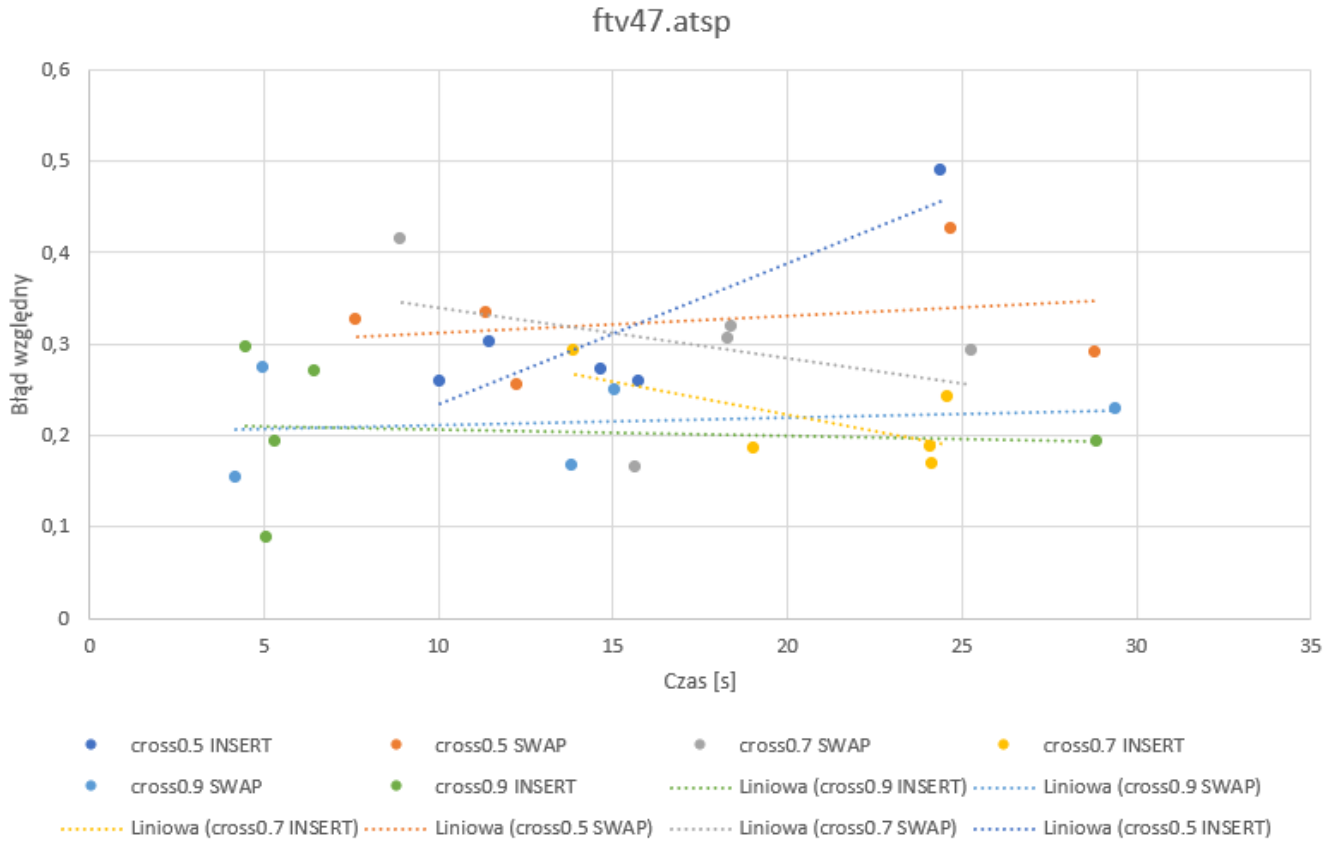


Rysunek 8: Pomiary dla różnych rozmiarów populacji dla pliku rbg403.atsp

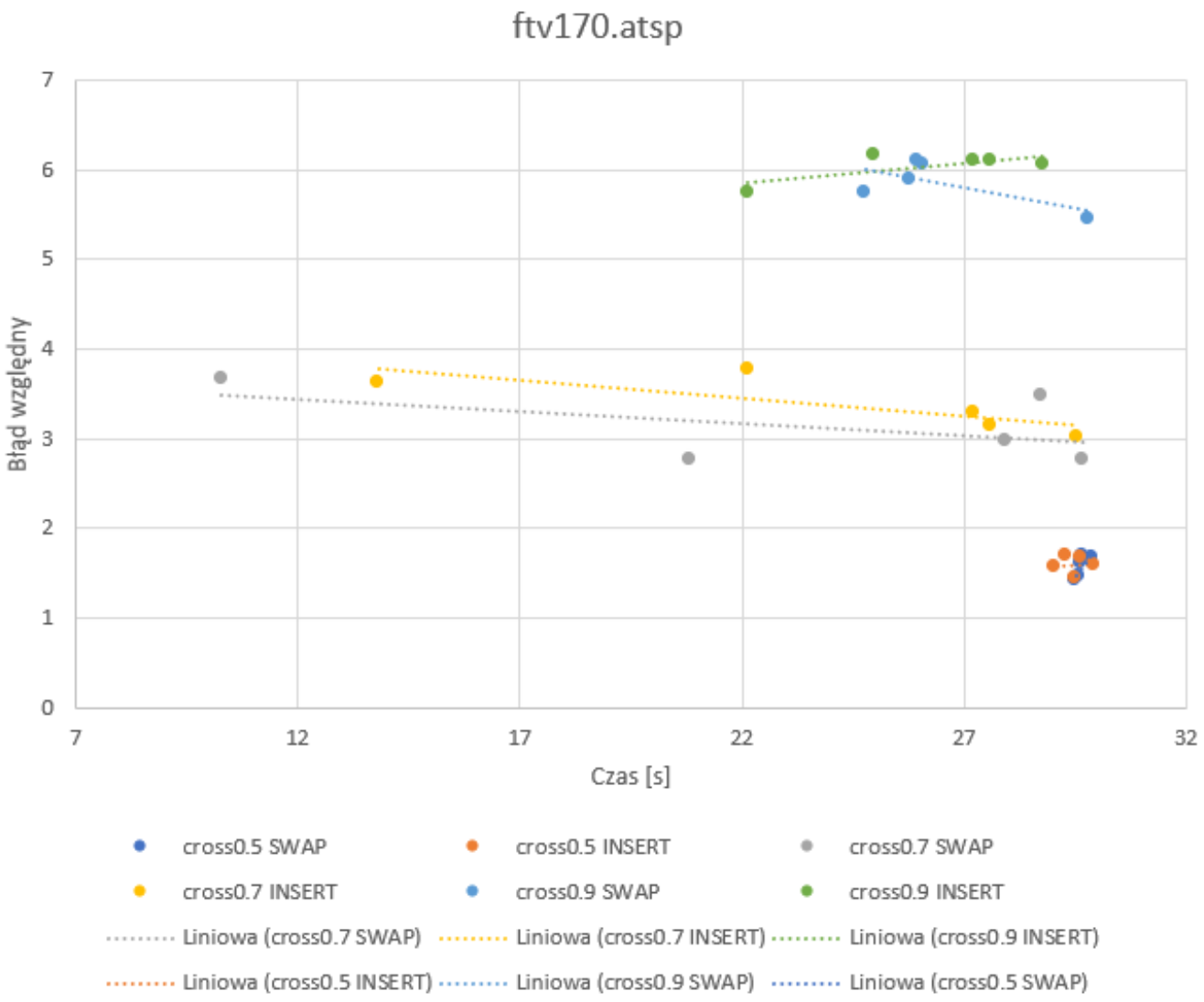
3.3 Zmienny współczynnik krzyżowania

Rozmiar	Mutacja	Wsp. Krzyż.	Czas	Błąd
47	SWAP	0.5	16,9 s	0,325
47	INSERT	0.5	15,28 s	0,315
47	SWAP	0.7	17,32 s	0,298
47	INSERT	0.7	21,17 s	0,214
47	SWAP	0.9	13,08 s	0,214
47	INSERT	0.9	10,06 s	0,207
170	SWAP	0.5	29,66 s	1,57
170	INSERT	0.5	29,48 s	1,58
170	SWAP	0.7	23,5 s	3,12
170	INSERT	0.7	24,06 s	3,36
170	SWAP	0.9	26,46 s	5,84
170	INSERT	0.9	26,14 s	6,02
403	SWAP	0.5	19,3 s	1,31
403	INSERT	0.5	23,38 s	1,23
403	SWAP	0.7	15,7 s	1,71
403	INSERT	0.7	11,76 s	1,72
403	SWAP	0.9	13,86 s	1,76
403	INSERT	0.9	17,77 s	1,757

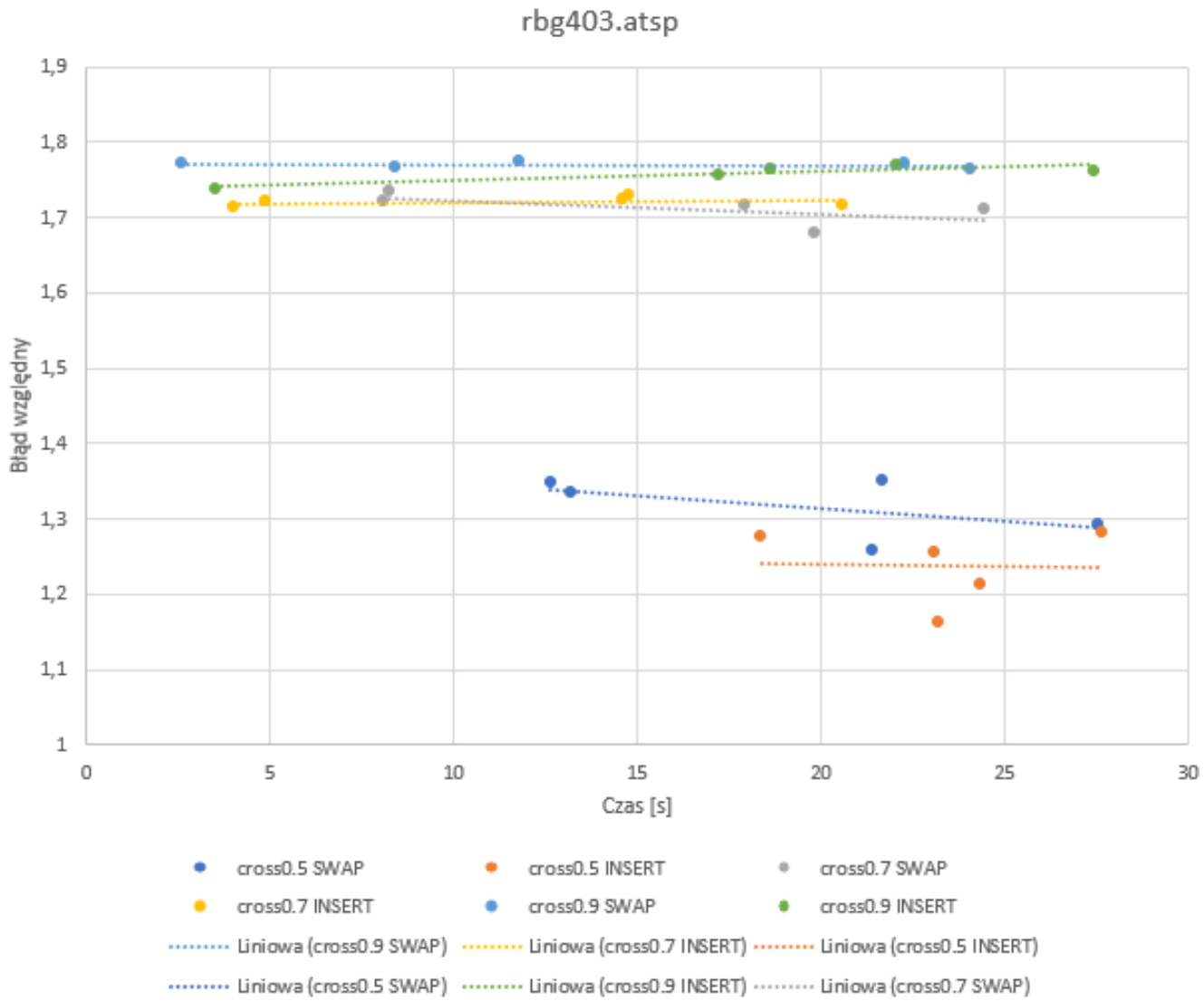
Tabela 2: Średnie pomiary dla różnych współczynników krzyżowania



Rysunek 9: Pomiary dla różnych współczynników krzyżowania dla pliku ftv47.atsp



Rysunek 10: Pomiary dla różnych współczynników krzyżowania dla pliku ftv170.atsp



Rysunek 11: Pomiary dla różnych współczynników krzyżowania dla pliku rbg403.atsp

3.4 Porównanie z Tabu Search

Rozmiar	Optymalnie	Znalezione	Czas	Błąd
47	1772	2207	2 ms	0,245485
170	2755	3771	20 ms	0,368784
403	2465	2866	5 s	0,162677

Tabela 3: Najlepsze wyniki algorytmu Tabu Search

Rozmiar	Optymalnie	Znalezione	Czas	Błąd
47	1772	1930	5,12 s	0,08671
170	2755	6673	29,52 s	1,42214
403	2465	5333	23,19 s	1,16349

Tabela 4: Najlepsze wyniki algorytmu genetycznego

4 Wnioski

Pierwsza grupa wykresów przedstawia wpływ wielkości populacji. Z tabeli 1 wynika, że najlepszym rozmiarem populacji dla większości przypadków jest największa wartość, czyli 1000. Stąd wniosek, że wielkość populacji ma wpływ na otrzymane wyniki, jednak nie jest to znaczący wpływ, różnice są kilkuprocentowe.

Następnie przetestowano wpływ współczynnika krzyżowania. Testy dały ciekawe wyniki. Dla problemu o wartości 47 miast, współczynnik nie miał istotnego znaczenia na wynik. Z testów wynika, że im większy współczynnik krzyżowania, tym lepszy rezultat otrzymamy. Dla problemu o wartości 170 miast współczynnik znacznie wpływał na rezultat. W tym przypadku mniejszy współczynnik przekładał się na lepsze wyniki algorytmu. Dla problemu o wartości 403 miast sytuacja wyglądała podobnie, mimo że różnice nie były tak duże. Współczynnik krzyżowania zdaje się mieć znaczny wpływ na wyniki algorytmu genetycznego, zwłaszcza na te o dużych rozmiarach problemu.

W większości przypadków metoda mutacji SWAP przekładała się na lepsze wyniki niż INSERT.

Na koniec porównano najlepsze wyniki Tabu Search i algorytmu genetycznego na podanych plikach. Sposób Tabu zwraca wyniki, które są bliższe optymalnych i szybsze niż te określone przez sposób genetyczny.

Literatura

[1] http://aragorn.pb.bialystok.pl/∼wkwedlo/EA5.pdf

[2] https://www.youtube.com/c/TheCodingTrain

[3] Roulette-wheel selection via stochastic acceptance, Adam Lipowski and Dorota Lipowska, Faculty of Physics, Adam Mickiewicz University, Poznań, Poland, Faculty of Modern Languages and Literature, Adam Mickiewicz University, Poznań, Poland, 6 Dec 2011

[4] Wykłady dr inż. Tomasza Kapłona z kursu Projektowania Efektywnych Algorytmów