Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie efektywnych algorytmów

Projekt nr 4 - Implementacja i analiza efektywności algorytmu genetycznego dla problemu komiwojażera

semestr zimowy 2023/2024

Autor:
Eryk Mika 264451

Prowadzący:
Dr inż. Marcin Łopuszyński

Spis treści

1. Wstęp teoretyczny[1]

Zgodnie z informacjami przedstawionymi w poprzednich sprawozdaniach, problem komiwojażera (*TSP*) jest problemem trudnym pod względem obliczeniowym. W tym opracowaniu zostanie omówione rozwiązanie tego problemu z wykorzystaniem algorytmu genetycznego. Jest to rodzaj algorytmu ewolucyjnego - jest wzorowany na biologicznej ewolucji oraz stosowany jest do optymalizacji oraz planowania.

Algorytm genetyczny jest heurystyką[2]. Symuluje on proces naturalnej selekcji poprzez ocenę adaptacji poszczególnych jednostek, eliminację słabszych osobników oraz krzyżowanie tych o największym przystosowaniu – w ten sposób powstają nowe osobniki w populacji. Każdy osobnik reprezentuje określony sposób rozwiązania problemu, który wyznacza dany *chromosom*. Osobniki oceniane są według pewnego kryterium – *funkcji oceny* – która, w przypadku problemu *TSP*, może być rozumiana jako funkcja przyporządkowująca koszt do danej trasy. Istotnym elementem algorytmu jest także mutacja, która polega na zmianie pewnych elementów rozwiązania według pewnego wzorca z określonym prawdopodobieństwem. Efektem tego procesu jest populacja jednostek, z których wybierane są te o najwyższym stopniu przystosowania. Zbiór informacji całej populacji określa się jako *genotyp*.

1.1. Metody krzyżowania

Krzyżowanie, realizowane poprzez *operator krzyżowania*, polega na kombinacji cech różnych osobników z populacji, co prowadzi do powstania nowych rozwiązań. Krzyżowanie zachodzi z pewnym ustalonym prawdopodobieństwem.

W zaimplementowanym i omawianym algorytmie zastosowano operator krzyżowania ***PMX*** (ang. *partially matched crossover*) – krzyżowanie z częściowym odwzorowaniem. W algorytmie realizującym ten operator wybierane są dwa punkty podziału, które wyznaczają tzw. sekcję dopasowania (ang. *matching section*). W ten sposób definiowane są punkty, które wyznaczają sposób transpozycji (zmianę miejsc) elementów danego rozwiązania[3] – szczegółowy opis algorytmu zawarty jest w opisie implementacji.

[1] https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy_genetyczne.pdf
[2] https://pl.wikipedia.org/wiki/Algorytm_genetyczny
[3] https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf

1.2. Metody mutacji[4]

Mutacja polega na wprowadzaniu losowych zmian do genotypu populacji. Ma to na celu zwiększenie różnorodności generowanych rozwiązań. Mutacja zachodzi z pewnym ustalonym prawdopodobieństwem, które z reguły jest niewielkie (≤1%), co ma na celu zachowanie równowagi pomiędzy przeszukiwaniem lokalnym (wokół pewnej grupy rozwiązań) oraz zwiększaniem przeszukiwanej przestrzeni rozwiązań[5].

W prezentowanym projekcie zastosowano dwa operatory mutacji: ***inverse*** oraz ***scramble***.

Operator *inverse* polega na odwróceniu kolejności elementów rozwiązania pomiędzy dwoma przyjętymi punktami w chromosomie (Rysunek 1.1).

Rysunek 1.1 Przykład zastosowania operatora inverse

Źródło: opracowanie własne

Operator *scramble* polega na losowym przestawianiu wybranych elementów z genotypu (Rysunek 1.2).

Rysunek 1.2 Przykład zastosowania operatora scramble

Źródło: opracowanie własne

1.3. Metoda selekcji

Selekcja w algorytmie genetycznym polega na wybieraniu osobników z populacji, które przejdą do następnego pokolenia (iteracji algorytmu). Możliwa jest realizacja selekcji na wiele sposobów. Do najważniejszych należą między innymi tzw. metoda ruletki oraz **metoda rankingowa[6]**, która została zaimplementowana i użyta w przedstawionym projekcie.

Metoda rankingowa w zaimplementowanej postaci polega na posortowaniu osobników w populacji rosnąco według przyjętej funkcji oceny – kosztu danej trasy. Następnie, zakładając, że do następnego pokolenia przechodzi *n* najlepszych osobników, pozostałe (gorsze) są usuwane (przykład przedstawia Rysunek 1.3).

Rysunek 1.3 Przykład zastosowania metody rankingowej. Populacja jest przedstawiona w postaci tablicy kosztów osobników.

[**55, 62, 74, 80, 91**, 100, 120, 182] → [55, 62, 74, 80, 91] *n = 5*

Źródło: opracowanie własne

[4] https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf
[5] https://www.baeldung.com/cs/genetic-algorithms-crossover-probability-and-mutation-probability
[6] https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)

1.4. Populacja początkowa

Populacja początkowa jest grupą osobników – rozwiązań, od których zaczyna swoje działanie algorytm. Wielkość tej populacji jest różna i zazwyczaj zależy ona od specyfiki rozwiązywanego problemu[7]. Często stosuje się wygenerowanie całości populacji w sposób losowy, jednakże spotyka się także podejście z wykorzystaniem „ziarna", które stanowią osobniki, o których wstępnie wiadomo, że mogą być obiecujące – są wygenerowane, na przykład, za pomocą metody zachłannej[8]. „Ziarno" to stanowi pewną część osobników populacji początkowej oprócz osobników wygenerowanych losowo. Ten sposób został wykorzystany w tym projekcie.

2. Opis implementacji algorytmów

W celu analizy efektywności omawianych algorytmów został napisany program w języku C++ z wykorzystaniem obiektowego paradygmatu programowania. Najistotniejszymi komponentami aplikacji są klasy *Graph* oraz *Route*, których pola (struktury danych) oraz metody są odpowiedzialne za realizację algorytmu. Wiele istotnych kwestii związanych z implementacją zostało wyjaśnionych w komentarzach w plikach źródłowych.

2.1. Klasa *Graph*

Klasa Graph jest główną klasą programu, która jest odpowiedzialna za przechowywanie struktury i metod grafu, na którym wykonywane są badane algorytmy. Pola prywatne klasy – dwuwymiarowa tablica *std::vector matrix* oraz *size* są użyte do przechowywania długości krawędzi w postaci macierzy kwadratowej – kosztów - *matrix* stopnia *size*. Oba pola przechowują liczby stałoprzecinkowe typu *int*. W macierzy komórka o współrzędnych *i*, *j* zawiera odległość pomiędzy wierzchołkami *i i j*.

Zaimplementowano konstruktor wczytujący instancję z pliku tekstowego, przeładowany operator przypisania oraz metodę wypisującą graf (macierz) na ekran. Została zaimplementowana metoda *calculateRouteCost()*, która służy do obliczania kosztu danej trasy komiwojażera w grafie – poprzez iterowanie po krawędziach w ścieżce i dodanie ich długości do kosztu, który jest przypisywany do odpowiedniego pola w obiekcie klasy *Route*.

2.2. Klasa *Graph* – metoda *generateInitialSolution()*

Metoda ta użyta do wyznaczenia rozwiązania początkowego w sposób **zachłanny**. Rozwiązanie to wykorzystane jest do tworzenia „ziarna" populacji początkowej.

[7] https://en.wikipedia.org/wiki/Genetic_algorithm
[8] https://medium.datadriveninvestor.com/population-initialization-in-genetic-algorithms-ddb037da6773

Algorytm:

1. Utworzenie listy *visited*, zainicjowanej odwiedzeniem korzenia (wierzchołka 0).
2. Rozpoczyna się pętla, która wykonuje się *size-1* razy, ponieważ trasa musi odwiedzić wszystkie wierzchołki oprócz korzenia.
3. Dla aktualnego wierzchołka *curIndex* na trasie, znajdowany jest najbliższy nieodwiedzony wierzchołek *dst* z najmniejszą wagą krawędzi. Wartość minimalnej wagi przechowywana jest w zmiennej *dstMin*.
4. Znaleziony wierzchołek *dst* jest dodany do listy visited i do wynikowej trasy *res* na odpowiedniej pozycji.
5. Po zakończeniu pętli, waga krawędzi powrotnej do korzenia, czyli od ostatnio odwiedzonego wierzchołka do korzenia (wierzchołek 0), jest dodawana do kosztu. Wygenerowana trasa jest zwracana z metody.

2.3. Klasa *Graph* – metoda *solveGA()*

Metoda ta użyta jest do rozwiązywania problemu komiwojażera za pomocą algorytmu genetycznego.

W metodzie tworzony jest wektor *population* przechowujący populację tras. Tworzona jest populacja początkowa. 10% osobników w tej populacji stanowią rozwiązanie uzyskane metodą zachłanną oraz osobniki pochodzące z tego rozwiązania poprzez wywoływania operatora *swap()* (zamiana elementów miejscami). Uzyskana populacja jest sortowana (Rysunek 2.1).

Rysunek 2.1 Algorytm tworzenia populacji początkowej

```
/*
   10% osobników w początkowej populacji wywodzi się z rozwiązania zachłannego
   razem z bazowym rozwiązaniem wygenerowanym zachłannie
*/
while(population.size() < (unsigned)(0.1 * initialPopulation))
{
/* Wygenerowanie osobników pochodzących z rozwiązania zachłannego (greedy) - losowe
przestawianie elementów trasy */
   Route r = greedy;
   r.procedureSwap(rand() % routeElements, rand() % routeElements);
   r.procedureSwap(rand() % routeElements, rand() % routeElements);
   calculateRouteCost(r);
   population.emplace_back(r);
}

// Reszta osobników jest wygenerowana losowo
while(population.size() < initialPopulation)
{
   Route r(routeElements);
   r.randomize();
   calculateRouteCost(r);
   population.emplace_back(r);
}

// Sortowanie populacji wg kosztów tras - podejście rankingowe
std::sort(population.begin(), population.end());
```

Źródło: opracowanie własne

Następnie, po inicjalizacji zmiennych związanych z obsługą pomiaru czasu, rozpoczyna się główna pętla algorytmu (Rysunek 2.2).

Rysunek 2.2 Główna pętla algorytmu genetycznego w postaci pseudokodu

```
// Główna pętla algorytmu
   while(warunek stopu)
   {
      // Rozmiar populacji w obecnej iteracji
      unsigned populationSize = population.size();

      for(unsigned i=0; i<populationSize; i++)
      {

         // Mutacja
         if( (double)rand() / (double)RAND_MAX < mutationFactor )
         {
            // Mutacja typu scramble
            if( mutationChoice )
            {
               // Wybierana jest losowa liczba elementów trasy do poprzestawiania
               population[i].mutateScramble( rand() % routeElements );
            }
            // Mutacja typu inverse
            else
            {
               // Odwracanie trasy pomiędzy indeksami idx1, idx2
               int idx1 = 0, idx2 = 0;

               // Uniknięcie braku efektu mutacji
               while(idx1==idx2)
               {
                  idx1 = rand() % routeElements;
                  idx2 = rand() % routeElements;
               }
               population[i].mutateInverse( idx1, idx2 );
            }
            calculateRouteCost(population[i]);
         }

         // Krzyżowanie
         if( (double)rand() / (double)RAND_MAX < crossoverFactor )
         {
            // Wybranie drugiego osobnika - rozwiązania - do krzyżowania
            unsigned secIndex = rand()% populationSize;
            if(secIndex == i) secIndex = ( secIndex + 1 ) % populationSize;
            Route offspring = population[i].crossoverPMX( population[secIndex] );
            calculateRouteCost(offspring);
            population.emplace_back(offspring);
         }
      }

      std::sort(population.begin(), population.end());

      // Jeżeli znaleziono nowe najlepsze rozwiązanie
      if( population[0].getCost() < bestSolution )
      {
         bestSolution = population[0].getCost();
         pomiarczasu;
      }

      /*
       Selekcja rodziców do nastepnej iteracji
       Wybierane jest 'eliteSize' najbardziej obiecujących (najlepszych) rozwiązań
       */
      if( population.size() > initialPopulation )
      {
         population.erase( population.begin() + eliteSize, population.end() );
      }

      // Sprawdzenie warunku stopu
      if( czas przekroczony)
      {
          break;
      }
   }
```

Źródło: opracowanie własne

W każdej iteracji algorytmu zachodzą operacje mutacji oraz krzyżowania – z ustalonym prawdopodobieństwem odpowiednio *mutationFactor* oraz *crossoverFactor*. W przypadku mutacji możliwe jest wybranie jednego z dwóch operatorów. W przypadku operatora *scramble* losowana jest liczba elementów do przestawiania, natomiast w przypadku operatora *inverse* losowane są dwa indeksy, które wyznaczają odwracany fragment rozwiązania. Po zakończeniu tych operacji populacja jest sortowania i, jeżeli znaleziono nowe najlepsze rozwiązanie, jest ono zapisywane (*bestSolution*). Jednocześnie dokonywany jest pomiar czasu jego znalezienia. Następuje selekcja osobników zgodnie z podejściem rankingowym – zostawiane jest *eliteSize* najlepszych osobników w populacji. Parametr *eliteSize* określony jest jako 50% liczebności populacji początkowej. Na końcu sprawdzany jest warunek stopu jako przekroczony czas – pomiar ten jest wykonywany co 10. iterację pętli w celu zapobieżenia nadmiernego wpływu pomiaru czasu na czas wykonywania właściwego algorytmu. Efekty działania algorytmu są zwracane z metody jako para *(czas znalezienia najlepszej trasy, koszt najlepszej trasy)*.

2.4. Klasa *Route*

Klasa ta jest użyta do reprezentowania ścieżki – trasy komiwojażera bez pierwszego i ostatniego przystanku na trasie, który jest przyjęty jako 0. Oznacza to, że przykładowo ciąg wierzchołków 0-1-2-3-0 jest w tej klasie reprezentowany jako ciąg 1-2-3. Ciąg wierzchołków jest przechowywany w postaci tablicy *std::vector<int> route*.

Zostały zaimplementowane następujące komponenty klasy:

- Konstruktor *Route(int n)* - tworzy obiekt trasy o rozmiarze *n*. Inicjalizuje wektor *route* o zadanej wielkości,
- Metoda *randomize()* generuje losową permutację trasy, reprezentującą trasę komiwojażera bez pierwszego i ostatniego przystanku na trasie (0),
- Metoda *toString()* zwraca tekstową reprezentację trasy, gdzie kolejne liczby są oddzielone spacją.
- Operator przypisania *operator=* przypisuje zawartość jednej trasy do drugiej.
- Operator dostępu do elementu *operator[]* umożliwia odczyt i modyfikację elementów trasy.
- Operator porównania *operator==* porównuje dwie trasy i zwraca *true*, jeśli są identyczne.
- Metoda *procedureSwap()* wykonuje operację swap (zamiana miejscami) dla dwóch wierzchołków na trasie.
- Metoda *procedureInverse()* wykonuje operację odwracania kolejności elementów trasy między dwoma wskazanymi indeksami.
- Metoda *getSize()* zwraca rozmiar trasy.

- Pomocnicza metoda *swap()* zamienia miejscami dwa elementy trasy na podstawie ich indeksów.

W klasie tej zostały zaimplementowane operatory mutacji oraz krzyżowania.

W przypadku operatorów mutacji, operator *inverse* odwraca kolejność elementów w danym fragmencie rozwiązania poprzez iteracyjne zamienianie elementów z pozycji *i* i *j*. Dla operatora scramble losowane są indeksy elementów które będą przestawiane (*chosenIndices*) – dodawane są te indeksy, które nie zostały jeszcze wybrane – nieoznaczone w trasie *markedAsVisited* jako 0. Następnie wybrane wcześniej elementy są losowo przestawiane za pomocą metody pomocniczej *swap()*.

Rysunek 2.3 Implementacja operatorów mutacji w klasie Route

```
// Mutacja typu inverse - iteracyjne odwracanie kolejności elementów
void Route::mutateInverse(unsigned i, unsigned j)
{
   if(i > j)
   {
      int temp = i;
      i = j;
      j = temp;
   }

   while(i < j) swap(i++, j--);

}

// Mutacja typu scramble - losowe przestawianie k elementów trasy
void Route::mutateScramble(unsigned k)
{
   if(k > route.size()) return;
   // Kopia obecnej trasy sluzaca do oznaczania wybranych elementów/indeksów - jako 0
   Route markedAsVisited = *this;
   // Wybrane indeksy do przestawiania
   std::vector<int> chosenIndices = {};

   // Wybierane są k elementy trasy, które będą losowo przestawiane
   while(chosenIndices.size() < k)
   {
      int chosenIndex = rand() % ( route.size() );
      if( markedAsVisited[chosenIndex] != 0 )
      {
         chosenIndices.push_back(chosenIndex);
         markedAsVisited[chosenIndex] = 0;
      }
   }

   // Wczesniej wybrane elementy trasy sa losowo przestawiane - wywołanie własnej metody pomocniczej
   for(unsigned i=0; i<chosenIndices.size(); i++)
      swap( chosenIndices[i], chosenIndices[ rand() % chosenIndices.size()] );
}
```

Źródło: opracowanie własne

W klasie zaimplementowano operator krzyżowania *PMX*. Algorytm zaczyna się od losowego wyboru dwóch indeksów (*a* i *b*), które określają segment poddawany krzyżowaniu. Zapewnione jest, że *a* jest mniejsze niż *b*. Następnie potomek jest tworzony, inicjalizując go trasą drugiego rodzica (*sec*). W kolejnym kroku następuje kopiowanie wybranego segmentu z pierwszego rodzica (*this*) do potomka. Dla każdego elementu w tym segmencie, kopiowany jest do potomka. Po skopiowaniu segmentu identyfikowane są pary elementów, które nie zostały skopiowane w tym segmencie z drugiego rodzica. Tworzona jest lista par (*i*, *j*), gdzie *i* to element z drugiego rodzica, a *j* to odpowiadający mu element z pierwszego rodzica. Następnie następuje iteracja przez te pary i elementy potomka są umieszczane w miejscach określonych przez te pary. W przypadku konfliktu, gdy element, który ma być umieszczony, już istnieje w potomku, dokonywane są odpowiednie zamiany. Na koniec zwracana jest trasa potomka, która zawiera połączenie cech obu rodziców z uwzględnieniem krzyżowania *PMX*.

Rysunek 2.4 Algorytm operatora PMX

```
// Zamieniamy jeżeli taka konieczność, aby a <= b
   if(a > b) {
      unsigned temp = a;
      a = b;
      b = temp;
   }
   // Wynikowy potomek
   Route offspring = sec;
   // Skopiowanie wybranego segmentu z pierwszego rodzica (*this)
   for(unsigned i=a; i<=b; i++)
   {
      offspring[i] = this->route[i];
   }

   std::vector<std::pair<int, int>> pairs = {};

   // Wyznaczenie par (i, j); elementy, które nie zostały skopiowane w analogicznym segmencie z drugiego rodzica
   for(unsigned i=a; i<=b; i++)
   {
      int secRouteElement = sec.route[i];
      bool isFound = false;
      for(unsigned k=a; k<=b; k++) {
         if(secRouteElement == this->route[k]) {
            isFound = true;
            break;
         }
      }
      if(isFound) continue;
      pairs.push_back(std::pair<int, int>(secRouteElement, this->route[i]));
   }

   // Umieszczanie elementów z par (i, j)
   for(unsigned p=0; p<pairs.size(); p++)
   {
      int i = pairs[p].first;
      int j = pairs[p].second;
      int destIndex = -1;

      unsigned it = 0;
      while( it < routeSize ) {
         // Czy wewąatrz skopiowanego fragmentu
         bool withinCopiedSegment = ( it >= a && it <= b );
         if( sec.route[it] == j && withinCopiedSegment )
         {
            j = this->route[it];
            it = 0;
            continue;
         } else if ( sec.route[it] == j ){
            destIndex = it;
            break;
         }
         it++;
      }
      // Umieszczenie i na pozycji zajmowanej przez j
      offspring[destIndex] = i;
   }
   return offspring;
```

Źródło: opracowanie własne

3. Sposób przeprowadzenia badania

W celu realizacji badania – eksperymentu zostały wykorzystane omówione wcześniej klasy. Podobnie jak w przypadku realizacji wcześniejszych projektów, w celu oceny efektywności badanych algorytmów został wykorzystany pomiar czasu przy wykorzystaniu funkcjonalności biblioteki <*chrono*> i zawartej w niej klasy *steady_clock*, która reprezentuje zegar monotoniczny, dla którego gwarantowane jest, że różnica czasu (przykładowo przekonwertowanego do milisekund – tak jak w programie) będzie większa od zera dla dwóch momentów czasu badania, z których drugi występuje później niż pierwszy. Po wykonaniu metody realizującej algorytm genetyczny wypisywany jest czas uzyskania wyniku w milisekundach oraz koszt ścieżki.

Do badań zostały wykorzystane problemy wskazane w wytycznych dot. projektu. Są one reprezentowane przez pliki:

- *ftv47.atsp* (1776),
- *ftv170.atsp* (2755) ,
- *rgb403.atsp* (2465).

W nawiasach zostały podane najlepsze znane rozwiązania – koszty tras – dla tych problemów[9].

Przeanalizowano rezultaty działania algorytmu dla dwóch opisanych wcześniej operatorów mutacji oraz jednego operatora przypisania. Badania zrealizowano w pierwszej kolejności dla współczynnika mutacji 0,01 oraz współczynnika krzyżowania 0,8 oraz następujących wielkości populacji $\mathbf{10^3}$, $\mathbf{10^4}$ oraz $\mathbf{10^5}$.

Następnie ustalono, że najlepsze rezultaty uzyskiwane są dla największej liczebności populacji (10^5), w związku z czym przeprowadzono kolejne badania dla tej wartości – przeanalizowano wpływ współczynnika mutacji na wyniki (wartości **0,02**, **0,5**, **0,1**) dla obu operatorów mutacji.

Ustalono warunek stopu – limit czasu wykonywania algorytmu na: 2 minuty dla problemu *ftv47.atsp*, 4 minuty dla problemu *ftv170.atsp* oraz 6 minut dla problemu *rbg403.atsp* – identyczny jak w poprzednim projekcie, co umożliwia porównanie działania algorytmów.

Sprawdzono ostateczne wartości błędu oraz wartości uzyskiwane w funkcji czasu dla pojedynczego uruchomienia algorytmu przy zadanych parametrach. Zebrano wyniki za pomocą tabel oraz wykresów.

4. Wyniki przeprowadzonego badania

4.1. Wszystkie wyniki przedstawione tabelarycznie

Wyniki uruchomień algorytmu dla poszczególnych problemów z ustalonymi parametrami zostały zestawione w tabelach. W ostatnich wierszach zawarto wartości średnie.

[9] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ATSP.html

Tabela 1

Problem ftv47.atsp
Mutacja: inverse Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
8.16e+04	2287	28%
4.43e+03	2234	25%
6.83e+04	2287	28%
2.85e+04	2287	28%
1.28e+04	2241	26%
7.45e+03	2287	28%
1.56e+04	2241	26%
1.04e+04	2234	25%
4.82e+04	2287	28%
9.17e+03	2241	26%
2.87e+04	2262	26%

Źródło: opracowanie własne

Tabela 2

Problem ftv47.atsp
Mutacja: inverse Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
4.93e+03	2218	24%
1.54e+03	2241	26%
1.20e+03	2217	24%
1.22e+03	2289	28%
1.09e+03	2234	25%
8.95e+02	2241	26%
1.26e+04	2181	22%
5.87e+04	2265	27%
1.25e+03	2234	25%
4.69e+04	2202	23%
1.30e+04	2232	25%

Źródło: opracowanie własne

Tabela 3

Problem ftv47.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.28e+04	2202	23%
2.03e+04	2176	22%
9.74e+04	2164	21%
1.10e+04	2217	24%
1.55e+04	2218	24%
3.34e+04	2173	22%
1.12e+04	2183	22%
1.01e+04	2183	22%
8.81e+03	2234	25%
3.76e+04	2145	20%
2.58e+04	2189	22%

Źródło: opracowanie własne

Tabela 4

Problem ftv47.atsp
Mutacja: scramble Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.02e+05	2130	19%
2.69e+03	2234	25%
5.89e+03	2210	24%
2.94e+04	2199	23%
7.13e+04	2181	22%
4.90e+04	2195	23%
9.86e+04	2199	23%
9.50e+04	2210	24%
5.28e+04	2176	22%
1.86e+04	2273	27%
5.25e+04	2200	23%

Źródło: opracowanie własne

Tabela 5

Problem ftv47.atsp
Mutacja: scramble Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
9.58e+04	2183	22%
1.14e+05	2183	22%
1.07e+05	2170	22%
8.25e+04	2165	21%
8.17e+04	2146	20%
6.60e+03	2218	24%
4.55e+04	2170	22%
8.01e+04	2141	20%
9.17e+04	2146	20%
1.14e+03	2243	26%
7.06e+04	2176	21%

Źródło: opracowanie własne

Tabela 6

Problem ftv47.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.34e+04	2135	20%
3.86e+04	2234	25%
1.16e+05	2175	22%
1.16e+05	2122	19%
1.55e+04	2215	24%
9.71e+04	2217	24%
8.64e+03	2234	25%
1.83e+04	2194	23%
1.03e+04	2183	22%
1.02e+04	2217	24%
4.45e+04	2192	22%

Źródło: opracowanie własne

Tabela 7

Problem ftv170.atsp
Mutacja: inverse Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
4.89e+04	3824	38%
2.38e+05	3822	38%
1.85e+05	3856	39%
2.00e+04	3861	40%
2.68e+03	3901	41%
9.90e+04	3852	39%

Źródło: opracowanie własne

Tabela 8

Problem ftv170.atsp
Mutacja: inverse Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
2.06e+03	3810	38%
5.10e+04	3844	39%
2.28e+05	3810	38%
5.48e+03	3921	42%
3.41e+03	3848	39%
5.79e+04	3846	39%

Źródło: opracowanie własne

Tabela 9

Problem ftv170.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.75e+04	3810	38%
2.41e+04	3785	37%
1.75e+04	3771	36%
1.88e+04	3824	38%
1.59e+04	3771	36%
1.87e+04	3792	37%

Źródło: opracowanie własne

Tabela 10

Problem ftv170.atsp
Mutacja: scramble Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.74e+05	3822	38%
1.27e+05	3771	36%
2.40e+05	3819	38%
5.87e+04	3700	34%
1.09e+05	3861	40%
1.42e+05	3794	37%

Źródło: opracowanie własne

Tabela 11

Problem ftv170.atsp
Mutacja: scramble Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
2.13e+05	3834	39%
1.20e+03	3846	39%
5.05e+04	3771	36%
1.63e+05	3771	36%
2.13e+05	3790	37%
1.28e+05	3802	37%

Źródło: opracowanie własne

Tabela 12

Problem ftv170.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.66e+04	3771	36%
1.57e+04	3771	36%
7.92e+04	3734	35%
1.76e+04	3771	36%
1.57e+04	3771	36%
2.90e+04	3763	35%

Źródło: opracowanie własne

Tabela 13

Problem rbg403.atsp
Mutacja: inverse Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.53e+05	3332	35%
3.43e+05	3330	35%
3.30e+05	3322	34%
3.27e+05	3349	35%
3.21e+05	3308	34%
3.35e+05	3328	34%

Źródło: opracowanie własne

Tabela 14

Problem rbg403.atsp
Mutacja: inverse Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.47e+05	3009	22%
3.59e+05	3282	33%
3.54e+05	3065	24%
3.46e+05	3234	31%
3.58e+05	3066	24%
3.53e+05	3131	26%

Źródło: opracowanie własne

Tabela 15

Problem rbg403.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.59e+05	2999	21%
3.56e+05	3006	21%
3.55e+05	2979	20%
3.58e+05	2991	21%
3.60e+05	2984	21%
3.57e+05	2991	20%

Źródło: opracowanie własne

Tabela 16

Problem rbg403.atsp
Mutacja: scramble Populacja: 1000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.58e+05	3142	27%
3.60e+05	3180	29%
3.55e+05	3193	29%
3.50e+05	3188	29%
3.49e+05	3219	30%
3.54e+05	3184	28%

Źródło: opracowanie własne

Tabela 17

Problem rbg403.atsp
Mutacja: scramble Populacja: 10000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.51e+05	3042	23%
3.31e+05	3194	29%
3.55e+05	3173	28%
3.47e+05	3156	28%
3.50e+05	3064	24%
3.47e+05	3125	26%

Źródło: opracowanie własne

Tabela 18

Problem rbg403.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.01
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.57e+05	2964	20%
3.57e+05	2983	21%
3.59e+05	3001	21%
3.55e+05	2926	18%
3.58e+05	2982	20%
3.57e+05	2971	20%

Źródło: opracowanie własne

Tabela 19

Problem ftv47.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
8.95e+04	2059	15%
1.11e+04	2210	24%
1.14e+04	2135	20%
9.94e+04	2174	22%
1.50e+04	2151	21%
4.53e+04	2145	20%

Źródło: opracowanie własne

Tabela 20

Problem ftv47.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.08e+05	2107	18%
1.67e+04	2210	24%
5.33e+04	2129	19%
1.08e+05	2073	16%
7.80e+04	2130	19%
7.29e+04	2129	19%

Źródło: opracowanie własne

Tabela 21

Problem ftv47.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
5.29e+04	1974	11%
9.19e+04	2093	17%
8.87e+04	2039	14%
1.18e+05	1984	11%
5.00e+04	2020	13%
8.02e+04	2022	13%

Źródło: opracowanie własne

Tabela 22

Problem ftv47.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.25e+04	2269	27%
6.93e+04	2122	19%
4.74e+04	2176	22%
7.97e+04	2082	17%
9.56e+04	2156	21%
6.09e+04	2161	21%

Źródło: opracowanie własne

Tabela 23

Problem ftv47.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
6.50e+04	2199	23%
8.26e+04	2133	20%
3.39e+04	2135	20%
7.60e+04	2191	23%
6.23e+04	2103	18%
6.40e+04	2152	20%

Źródło: opracowanie własne

Tabela 24

Problem ftv47.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 120 s.

Czas [ms]	Koszt ścieżki	Błąd
1.16e+05	2122	19%
7.74e+04	2114	19%
1.63e+04	2218	24%
8.48e+04	2122	19%
1.17e+05	2027	14%
8.23e+04	2120	19%

Źródło: opracowanie własne

Tabela 25

Problem ftv170.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
2.59e+04	3822	38%
1.80e+04	3771	36%
1.72e+04	3771	36%
1.89e+04	3771	36%
1.99e+04	3771	36%
2.00e+04	3781	36%

Źródło: opracowanie własne

Tabela 26

Problem ftv170.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
2.41e+04	3695	34%
1.81e+04	3771	36%
3.72e+04	3768	36%
1.89e+04	3771	36%
1.12e+05	3771	36%
4.20e+04	3755	35%

Źródło: opracowanie własne

Tabela 27

Problem ftv170.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
3.08e+04	3822	38%
3.84e+04	3693	34%
2.02e+04	3771	36%
2.08e+04	3771	36%
2.34e+04	3771	36%
2.67e+04	3765	36%

Źródło: opracowanie własne

Tabela 28

Problem ftv170.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.62e+05	3781	37%
2.00e+04	3771	36%
2.51e+04	3771	36%
1.64e+05	3747	36%
1.91e+04	3771	36%
7.79e+04	3768	36%

Źródło: opracowanie własne

Tabela 29

Problem ftv170.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.94e+04	3771	36%
2.46e+04	3771	36%
1.49e+05	3695	34%
2.36e+04	3771	36%
2.26e+04	3771	36%
4.78e+04	3755	35%

Źródło: opracowanie własne

Tabela 30

Problem ftv170.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 240 s.

Czas [ms]	Koszt ścieżki	Błąd
1.39e+05	3695	34%
9.77e+04	3676	33%
2.24e+05	3676	33%
1.70e+05	3710	34%
2.31e+05	3694	34%
1.72e+05	3690	33%

Źródło: opracowanie własne

Tabela 31

Problem rbg403.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.58e+05	3040	23%
3.58e+05	2982	20%
3.60e+05	2986	21%
3.52e+05	2966	20%
3.58e+05	3024	22%
3.57e+05	2999	21%

Źródło: opracowanie własne

Tabela 32

Problem rbg403.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.60e+05	3030	22%
3.60e+05	3032	23%
3.59e+05	3011	22%
3.58e+05	3037	23%
3.55e+05	2987	21%
3.58e+05	3019	22%

Źródło: opracowanie własne

Tabela 33

Problem rbg403.atsp
Mutacja: inverse Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.53e+05	3024	22%
3.59e+05	3042	23%
3.49e+05	3048	23%
3.57e+05	3036	23%
3.57e+05	2987	21%
3.55e+05	3027	22%

Źródło: opracowanie własne

Tabela 34

Problem rbg403.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.02
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.56e+05	3004	21%
3.57e+05	2996	21%
3.59e+05	2959	20%
3.53e+05	3032	23%
3.58e+05	3014	22%
3.56e+05	3001	21%

Źródło: opracowanie własne

Tabela 35

Problem rbg403.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.05
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.56e+05	3062	24%
3.56e+05	3005	21%
3.59e+05	2984	21%
3.58e+05	3043	23%
3.56e+05	3050	23%
3.57e+05	3028	22%

Źródło: opracowanie własne

Tabela 36

Problem rbg403.atsp
Mutacja: scramble Populacja: 100000 Wsp. mutacji: 0.1
Wsp. krzyżowania: 0.8 Limit czasu: 360 s.

Czas [ms]	Koszt ścieżki	Błąd
3.59e+05	3075	24%
3.58e+05	3058	24%
3.59e+05	3095	25%
3.56e+05	3067	24%
3.57e+05	3076	24%
3.58e+05	3074	24%

Źródło: opracowanie własne

4.2. Najlepsze wyniki dla poszczególnych problemów

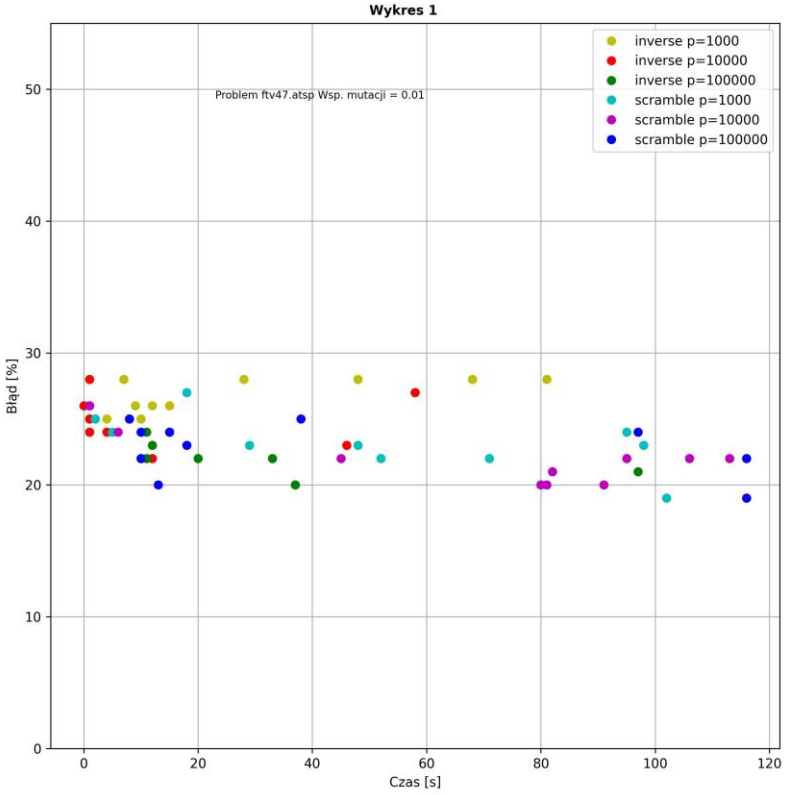
Wyznaczono najlepsze rozwiązania dla badanych problemów uzyskane podczas badań:

- ftv47.atsp – koszt ścieżki 1974 – w poprzednim projekcie uzyskano wynik 1911 za pomocą algorytmu symulowanego wyżarzania (o około 3% więcej),
- ftv170.atsp – koszt ścieżki 3693 - w poprzednim projekcie uzyskano wynik 3632 za pomocą algorytmu tabu search (o około 1% więcej),
- rbg403.atsp – koszt ścieżki 2959 - w poprzednim projekcie uzyskano wynik 2597 za pomocą algorytmu tabu search (o około 13% więcej).

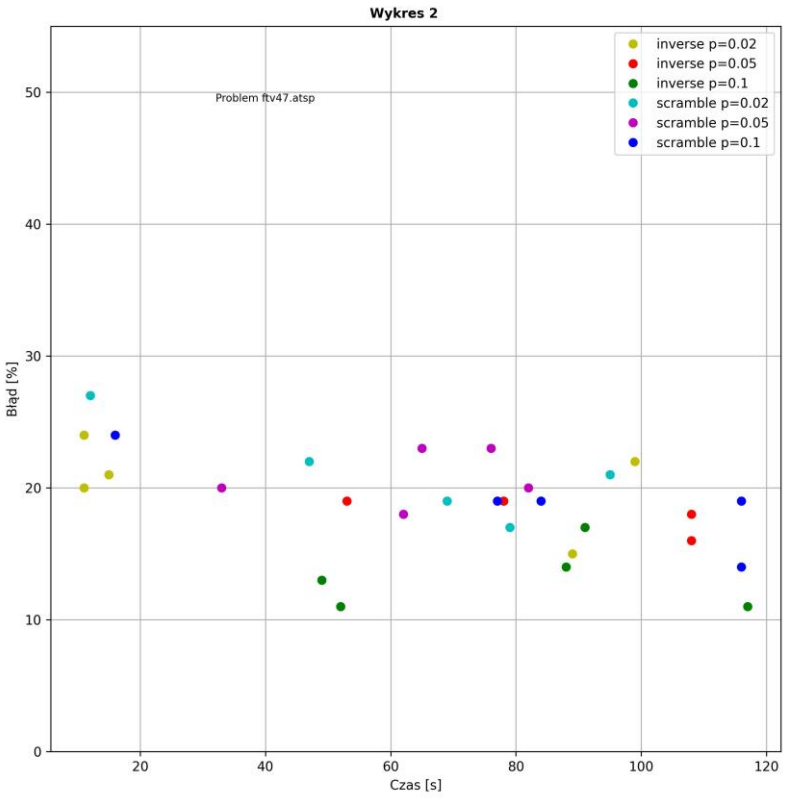
4.3. Wyniki przedstawione za pomocą wykresów – ostateczne rozwiązania

Zgodnie z założeniami dot. projektu, sporządzono wykresy współczynnika błędu względnego w funkcji czasu działania algorytmu – najpierw dla ostatecznych rozwiązań.

Po każdym wykresie sporządzonym dla poszczególnych problemów dla różnych wielkości populacji początkowej (p) podany jest wykres dla różnych wartości współczynnika mutacji dla danego problemu – dla populacji 10^5.

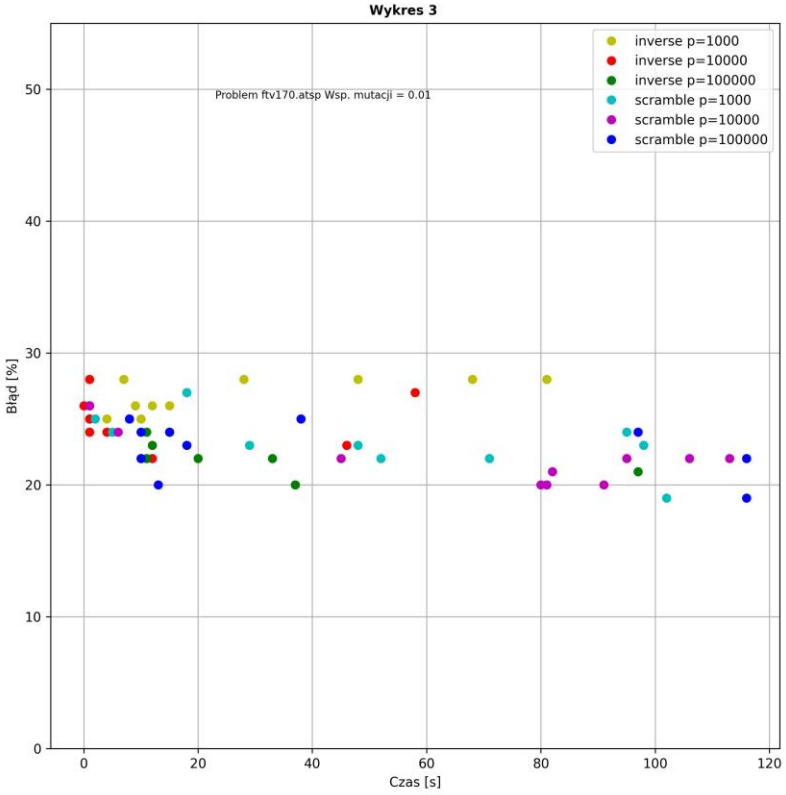


Źródło: opracowanie własne



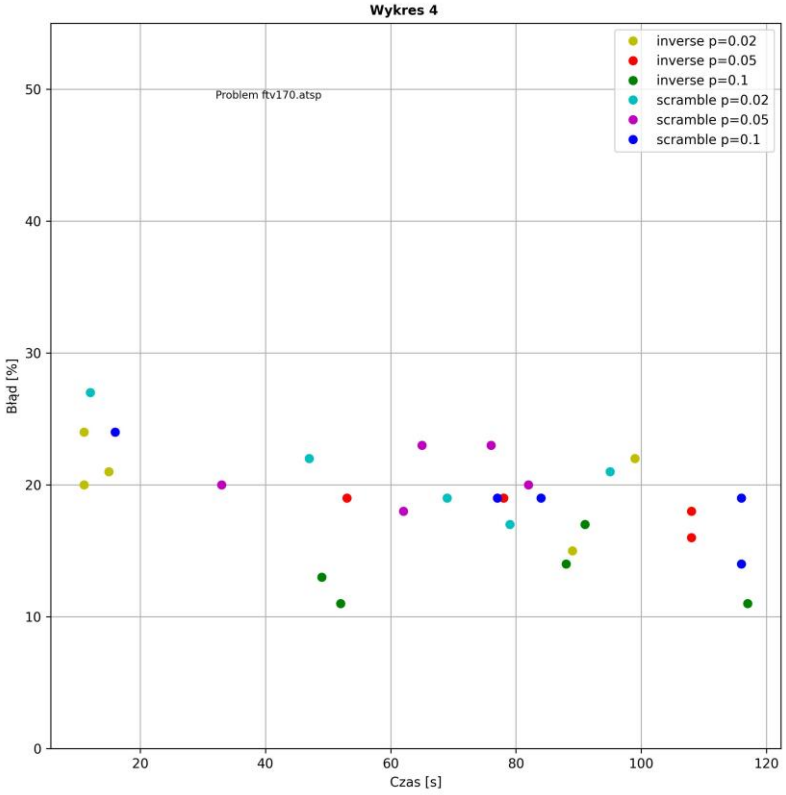
Źródło: opracowanie własne

Wykres 3



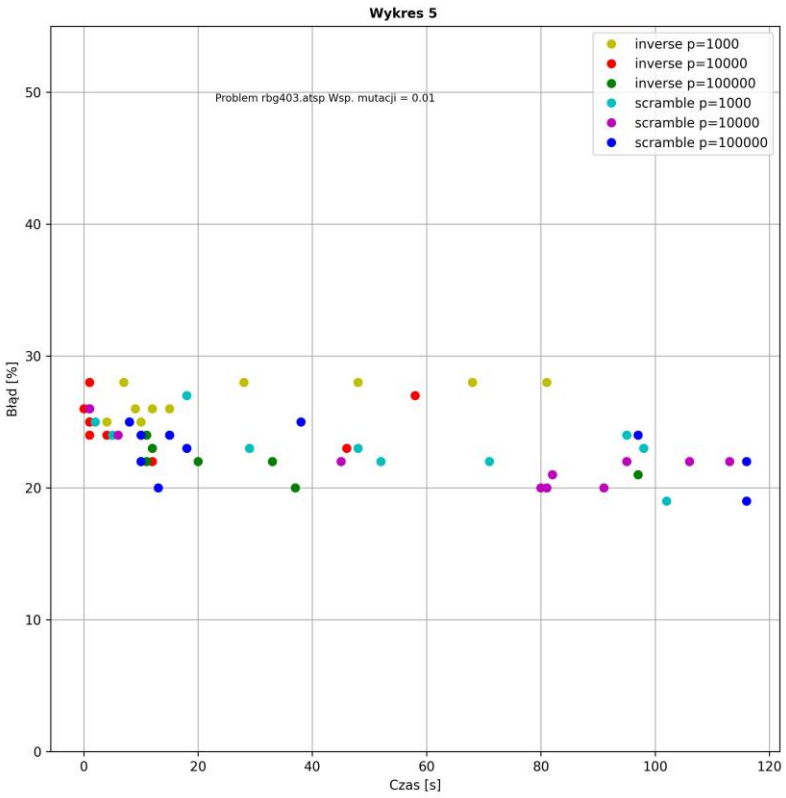
Źródło: opracowanie własne

Wykres 4



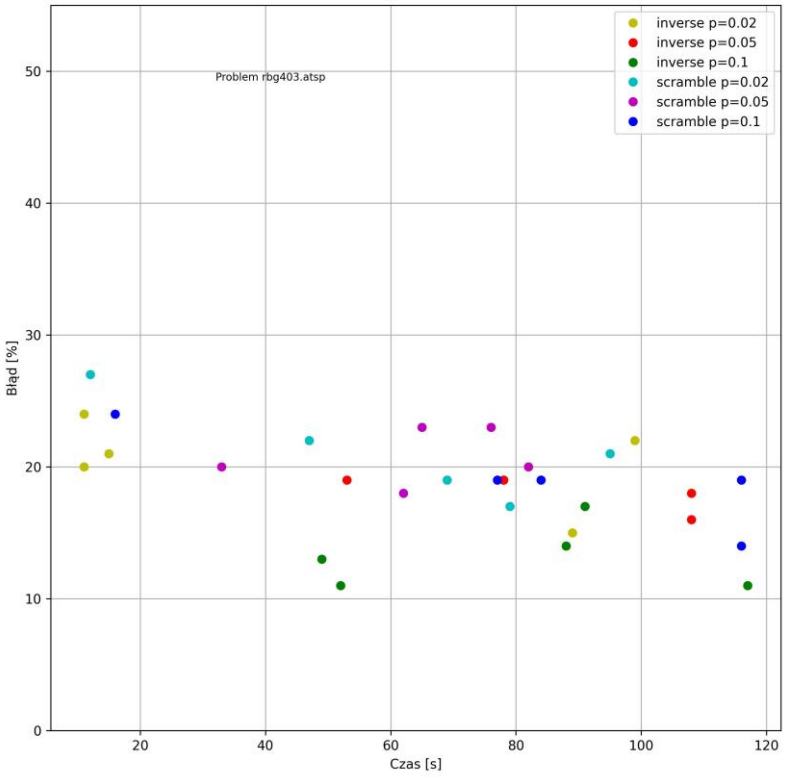
Źródło: opracowanie własne

Wykres 5



Źródło: opracowanie własne

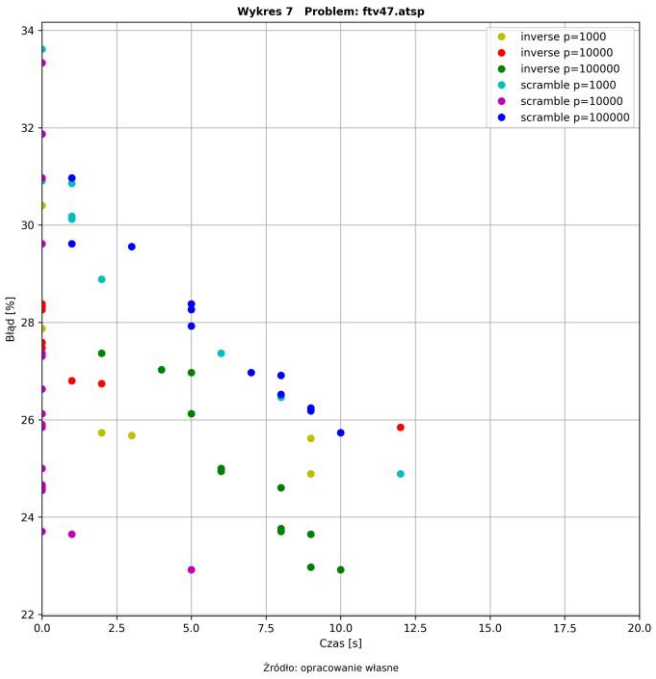
Wykres 6



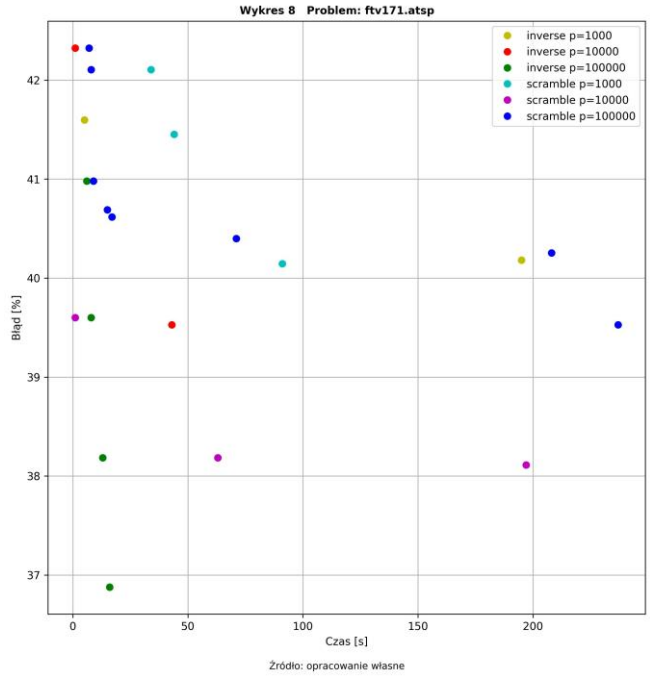
Źródło: opracowanie własne

4.4. Wyniki przedstawione za pomocą wykresów – zmiana błędu w funkcji czasu

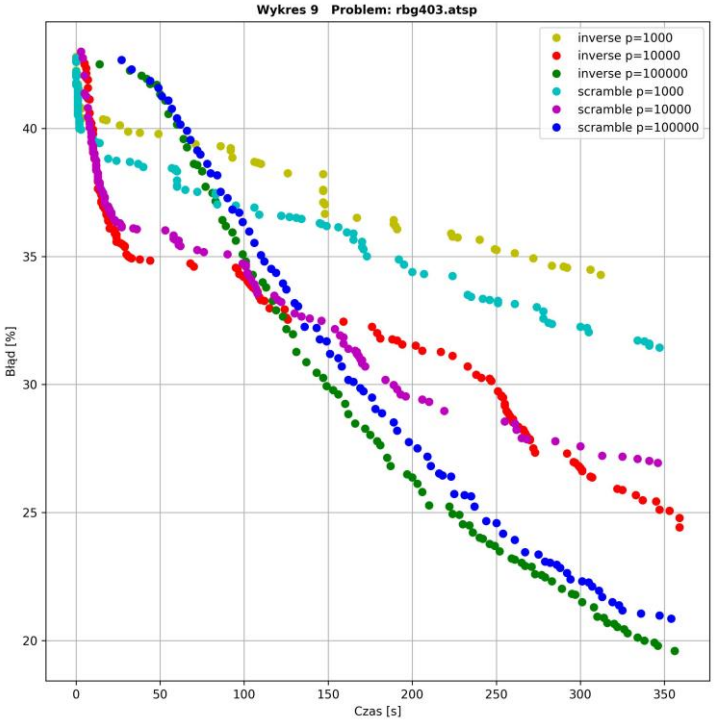
Zgodnie z sugestiami dotyczącymi projektu, sporządzono również wykresy przedstawiające zmiany współczynnika błędu względnego dla pojedynczych uruchomień algorytmu przy zadanych parametrach (Wykresy 7-12). Wykresy 7-9 przedstawiają pierwszy etap badania (różne wielkości populacji początkowej), wykresy 10-12 przedstawiają wpływ różnych wartości współczynnika mutacji przy populacji początkowej 10^5.



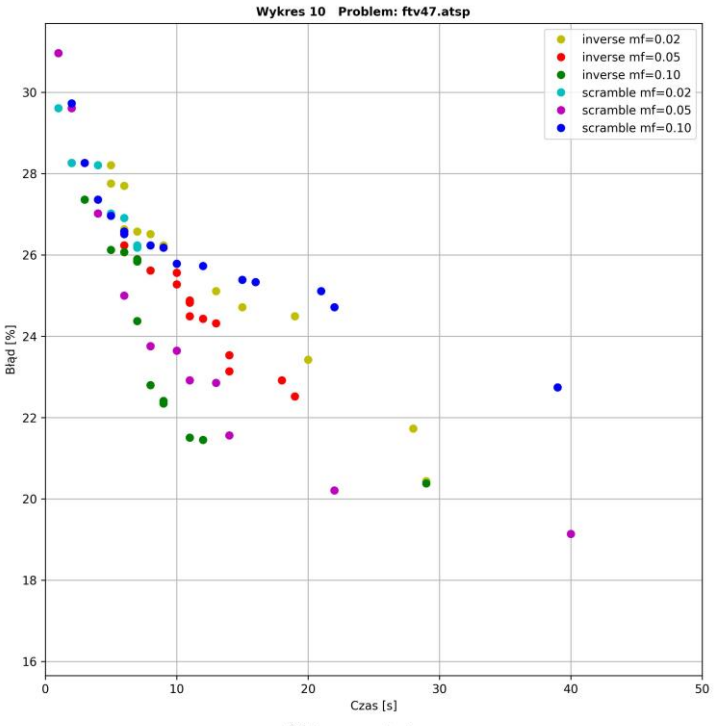
Źródło: opracowanie własne



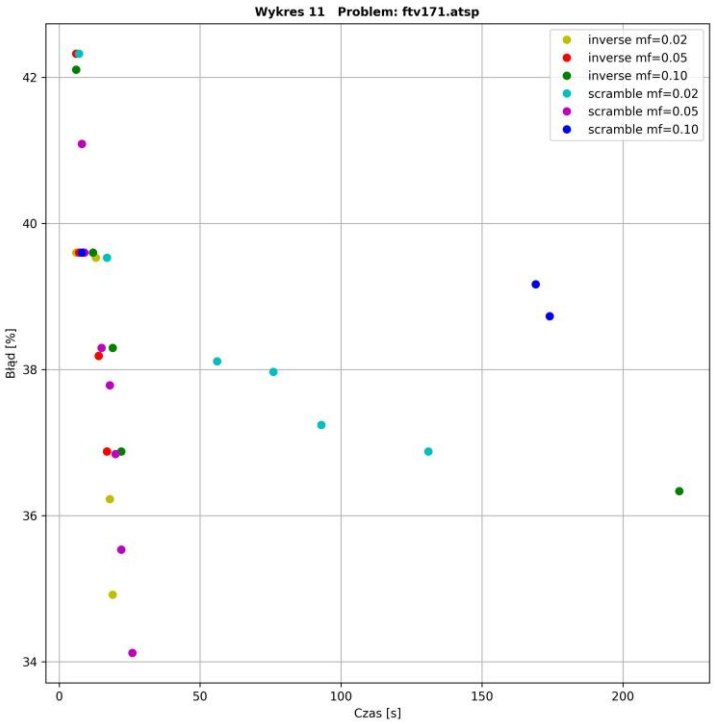
Źródło: opracowanie własne



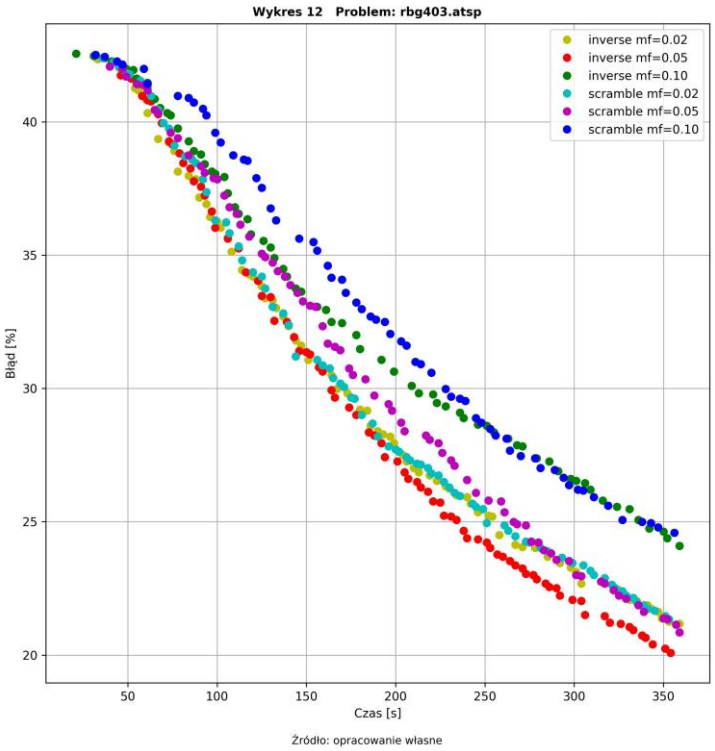
Źródło: opracowanie własne



Źródło: opracowanie własne



Źródło: opracowanie własne



Źródło: opracowanie własne

5. Komentarz do uzyskanych wyników

Na podstawie zebranych danych trudno określić znaczącą różnicę pomiędzy wynikami otrzymywanymi za pomocą mutacji *inverse* i *scramble* (por. Wykres 9). Zwiększenie współczynnika mutacji pozwala na uzyskanie mniejszych wartości błędu – lepszych rozwiązań (porównanie wartości z tabeli 1-18 oraz 19-36).

Problem *ftv170.atsp*, podobnie jak w poprzednim projekcie, okazał się problemem bardzo „trudnym" – nie udało się osiągnąć błędu względnego mniejszego niż 30%. Problem okazał się stosunkowo „nieczuły" nawet na zmianę parametrów mutacji.

W przypadku problemu *rbg403.atsp* – można zauważyć, że znajdowanie nowych rozwiązań miało miejsce jeszcze tuż przed upływem limitu czasu, więc mógł on być niewystarczający dla algorytmu zaimplementowanego w ten sposób (Tabele 32-36, Wykresy 9 i 12).

6. Wnioski

Wnioskuje się, że, podobnie jak omówione wcześniej algorytmy symulowanego wyżarzania oraz tabu search, algorytm genetyczny pozwala na wygenerowanie rozwiązań dla tych samych problemów, co algorytmy dokładne, lecz nie gwarantuje znalezienia najlepszego (optymalnego) rozwiązania. Są to algorytmy o efektywności zbliżonej do tych badanych wcześniej (0), jednakże z różnicami, które mogą wynikać z konkretnego sposobu implementacji algorytmu. Stwierdza się, że udało się poprawnie zaimplementować algorytm.

7. Bibliografia

[1] https://ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503404.pdf