Jakub Superczyński 241381 29.11.2020

Projektowanie efektywnych algorytmów

Implementacja i analiza efektywności algorytmu genetycznego dla problemu komiwojażera.

Prowadzący zajęcia:

Dr inż. Dariusz Banasiak

1. Wstęp teoretyczny

Problem komiwojażera to zagadnienie optymalizacyjne z teorii grafów polegające na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Dla łatwiejszego zobrazowania możemy przedstawić to w ten sposób: przedstawiciel handlowy ma listę pewnej liczby miast, które musi odwiedzić a następnie powrócić do miasta początkowego. Problem polega na tym, aby tak zaplanował swoją podróż aby przejazdy między miastami kosztowały go jak najmniej (np. aby odległości były jak najkrótsze lub jak najmniej zapłacił za paliwo). Co ważne: zakładamy, że graf może być asymetryczny, tzn. waga tej samej krawędzi (koszt tej samej trasy) może być różna w zależności w którą stronę chcemy przez nią przejść.

Jednym z algorytmów rozwiązujących TSP jest algorytm genetyczny. Jego działanie wzoruje się na biologii, a dokładniej na procesie ewolucji opisanej przez Karola Darwina. Według jego teorii wraz z biegiem historii zwierzęta z pokolenia na pokolenie się zmieniają, tzn. ewoluują. Zmiany te jednak nie są przypadkowe: ponieważ wszystkie zwierzęta umieszczone są w środowisku z ograniczony zasobami, tylko część zdoła przetrwać i przekazać swoje cechy następnym pokoleniu, inne zaś powymierają. Oczywistym jest, że przetrwają osobniki najlepsze (czyli: najlepiej przystosowane do środowiska). Tak więc z każdym pokoleniem coraz częściej będą pojawiać się cechy pozytywne, pozwalające przetrwać, te negatywne zaś będą zanikać, w teorii więc każde kolejne pokolenie będzie miało więcej lepszych osobników.

W algorytmie genetycznym poszczególnymi osobnikami są konkretne rozwiązania danej instancji zakodowane w odpowiedni sposób. Wybór osobników które mają przetrwać i przekazać swoje cechy dalej (czyli *selekcję*) dokonuje się na podstawie funkcji dopasowania, która określa jak dobrze przystosowany jest dany osobnik. Dla TSP może to być np. odwrotność odległości danego rozwiązania. *Krzyżowanie* może być wykonywane właściwie dowolne, musi jednak spełniać pełne warunki: dziecko musi posiadać część cech jednego i część cech drugiego rodzica oraz musi być dopuszczalnym rozwiązaniem. Należy również określić sposób *mutacji*. Tutaj również jest pewna dowolność, należy jednak pamiętać że ma ona wprowadzać niewielkie zmiany w jednym osobniku, a zmutowany osobnik również musi być dopuszczalnym rozwiązaniem. Ostatnią kwestią jest *warunek stopu*. Ewolucja biologiczna może trwać bez końca (ewentualnie do wyginięcia danego gatunku), algorytm jednak należy w pewnym momencie zakończyć by był użyteczny. Może to być ilość iteracji lub czas działania.

Pseudokod algorytmu można przedstawić następująco:

1. wybór populacji początkowej chromosomów (losowy)
2. ocena przystosowania chromosomów
3. sprawdzanie warunku zatrzymania
   a. selekcja chromosomów - wybór populacji macierzystej (*ang. mating pool*)
   b. krzyżowanie chromosomów z populacji rodzicielskiej
   c. mutacja - może być również wykonana przed krzyżowaniem
   d. ocena przystosowania chromosomów
   e. utworzenie nowej populacji
4. wyprowadzenie ,,najlepszego'' rozwiązania

2. Opis implementacji:

Program składa się z trzech klas: *TSP* odpowiedzialnej za przechowywanie informacji o instancji (tablica sąsiedztwa grafu oraz rozmiar problemu), generowanie losowych instancji, wczytywanie instancji z pliku i wypisywanie obecnie wczytanego grafu; *GA* odpowiedzialnej za znalezienie rozwiązania dla danej instancji oraz klasy pomocniczej *Chromosome* reprezentującej konkretnego osobnika.

Tablica sąsiedztwa grafu przechowywana jest w klasie *TSP* dwuwymiarowej tablicy alokowanej dynamicznie. Pamięć na nią alokowana jest zawsze przed wygenerowaniem lub wczytaniem nowej instancji poprzez wywołanie funkcji *Initialize()*. Dla ułatwienia implementacji klasa *GA* również posiada zapisane rozmiar problemu oraz wskaźnik na tablicę z reprezentacją grafu. Klasa *GA* posiada również w sobie zapisane parametry potrzebne do przeprowadzenia algorytmu oraz 3 tablice zaimplementowane jako vector przechowujące: populację, populację macierzystą, elitę. Poszczególne rozwiązanie reprezentowane jest przez obiekt klasy *Chromosome*, geny kodowane są w postaci tablicy zawierającej indeksy wierzchołków które po kolei odwiedzamy (nie zapisujemy wierzchołka początkowego i końcowego, który zawsze jest wierzchołkiem zerowym)

Najważniejsze elementy kodu:

- Generacja pierwszej populacji:

  Wypełniamy tablicę wartościami: 1, 2, 3, …, rozmiar_problemu-1. Następnie zmieniamy losowo kolejność wykorzystując funkcję *random_shuffle()* z biblioteki *algorithm* i dodajemy zmienioną tablicę do tablicy populacji. Powtarzamy losowanie i dodawanie aż osiągniemy odpowiednią liczbę osobników.

- Generowanie populacji macierzystej:

  Wykorzystana jest metoda ruletki. Najpierw sumujemy funkcje przystosowania wszystkich obecnych osobników. Losujemy liczbę z przedziału (0, obliczona suma). Następnie iterujemy przez całą populację, ponownie sumując funkcje przystosowanie. Gdy liczona suma przekroczy wylosowaną liczbę, dany osobnik dodawany jest do populacji macierzystej.

- Generowanie nowego pokolenia:

  Zastosowany jest elityzm. To znaczy w pierwszym kroku wybieramy 10% najlepszych osobników z populacji i przekopiowujemy bez zmian do nowej populacji. Następnie iterujemy przez kolejne pary osobników w populacji macierzystej. Losujemy liczbę, jeśli będzie mniejsza niż szansa na krzyżowanie, krzyżujemy osobnika i-tego z i+1-ym oraz i+1-ego z i-tym i dodajemy dzieci do następnej populacji. Jeśli liczba będzie większa nowe osobniki są kopiami rodziców. W algorytmie zaimplementowano mutację *Ordered Crossover.*

  Mutacja OX:
  Losujemy dwa indeksy w chromosomie. Wszystkie elementy od pierwszego do drugiego indeksu włącznie kopiujemy na te same miejsca w potomku. Następne

elementy kopiowane są w kolejności z drugiego rodzica. Jeśli nie jest to możliwe (tzn. dany element został przekopiowany już z pierwszego), próbujemy następny element tak długo, aż będziemy mogli. Do sprawdzenia czy dany element już został przekopiowany wykorzystujemy funkcję *find*.

Następnie mutuje się pokolenie. Iteruje się przez całe pokolenie i losuje liczbę. Jeśli jest mniejsza niż szansa mutacji, zmienia się danego osobnika w nowym pokoleniu na jego zmutowaną wersję. Zaimplementowano dwa rodzaje mutacji:

Mutacja *transposition:*
Losuje się dwa indeksy w chromosomie tak długo aż będą różne. Następnie zamienia się miejscami elementy o tych indeksach.

Mutacja *inversion:*
Losuje się dwa indeksy w chromosomie tak długo aż będą różnie. Następnie zamienia się kolejność między tymi indeksami (łącznie z elementami o tych indeksach). Do zamiany kolejności wykorzystujemy funkcję *reverse*.

- Zakończenie algorytmu:

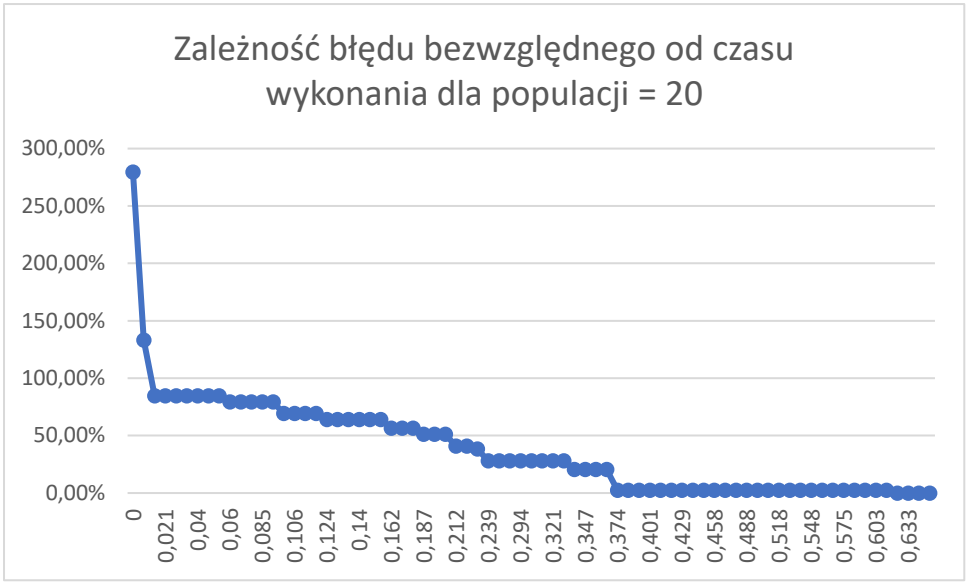
Warunkiem skończenia algorytmu jest ilość generacji bez poprawy. W algorytmie jest licznik zliczający wszystkie pokolenia, a zerujący się gdy najlepsze rozwiązanie się zmienia (po każdej iteracji najlepsze rozwiązywanie jest porównywane z najlepszym osobnikiem w danej populacji, jeśli nastąpiła poprawa, osobnik zostaje nowym najlepszym rozwiązaniem). Gdy licznik będzie równy liczbie podanej przez użytkownika, algorytm zostaje zakończony i zwraca obecnie najlepszą wartość.

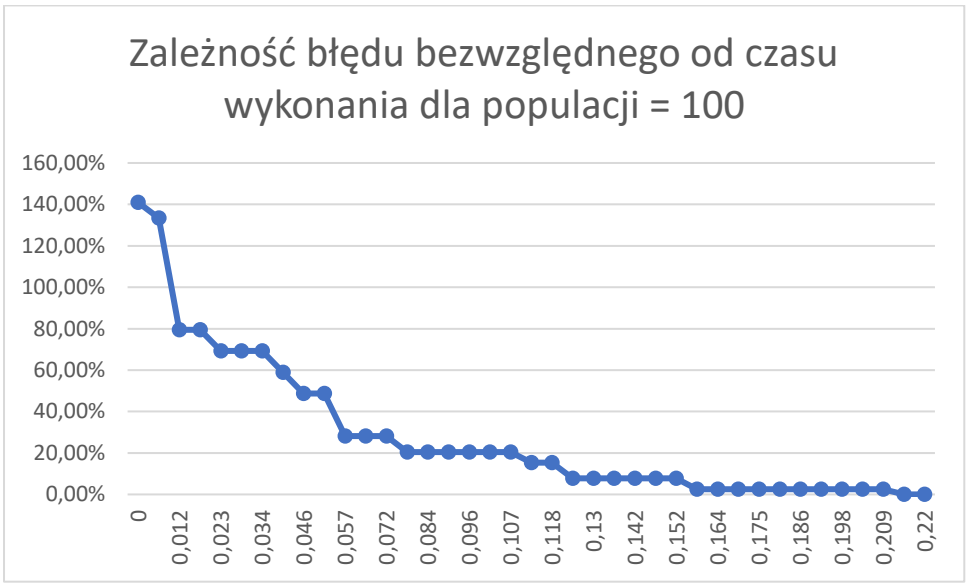
3. Plan eksperymentu

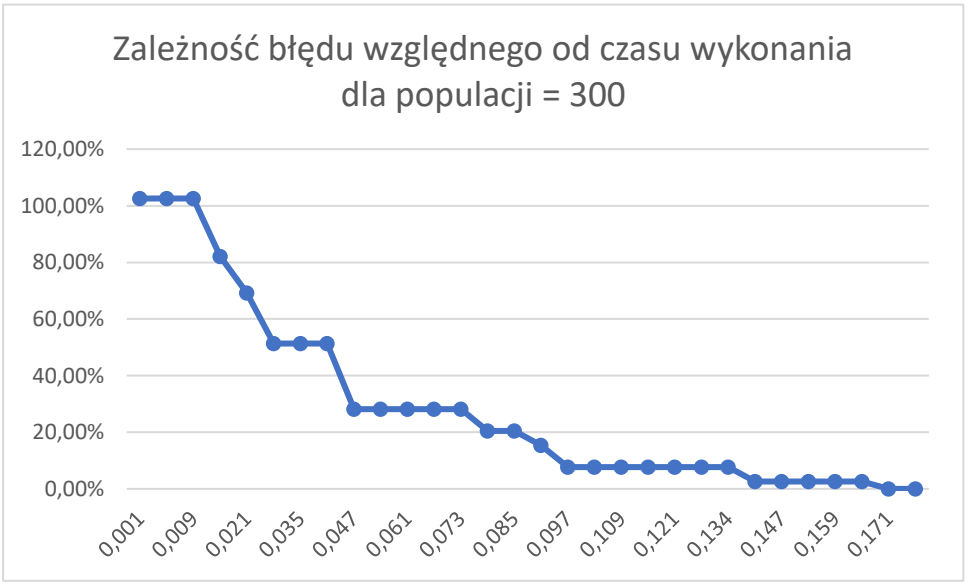
W przypadku tego algorytmu najważniejszą kwestią jest analiza zależności błędu względnego rozwiązania od czasu działania. Wybrano pięć instancji problemu: *br17 (39), ftv35 (1473), data58 (25395), data120 (6942), data323 (1326)* i dla każdej z nich uruchomiono algorytm z trzema różnymi wartościami liczby populacji: 20, 100, 300. Aby dostarczyć kompleksowej informacji o zależności błąd – czas, algorytm zapisywał w pliku tekstowym po każdej iteracji koszt obecnie najlepszego rozwiązania oraz dotychczasowy czas. Maksymalna ilość generacji została dobrana w taki sposób aby błąd był dopuszczalny i zależy od rozmiaru problemu. Współczynniki mutacji i krzyżowania natomiast są takie same i wynoszą odpowiednio 0.7 oraz 0.1.

4. Wyniki eksperymentu

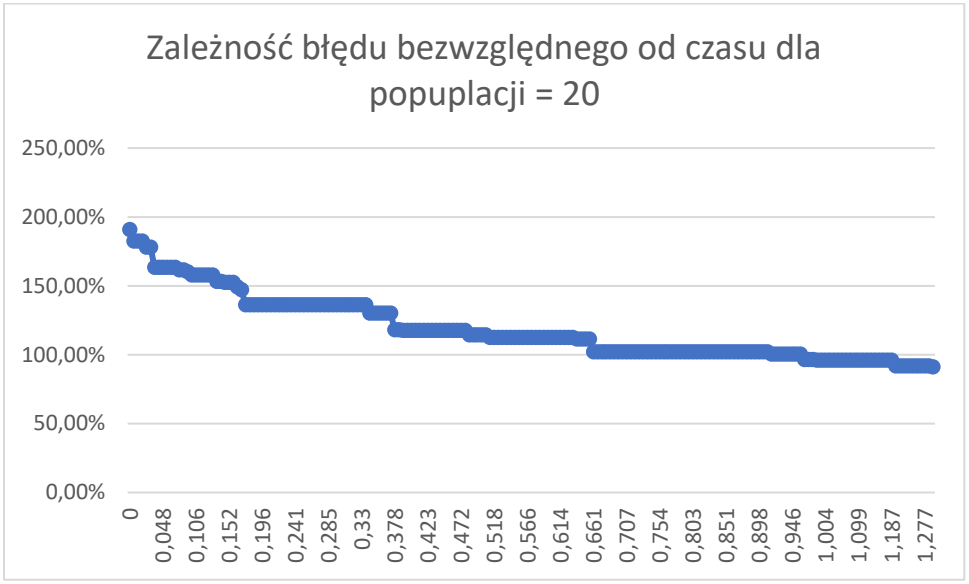
br17

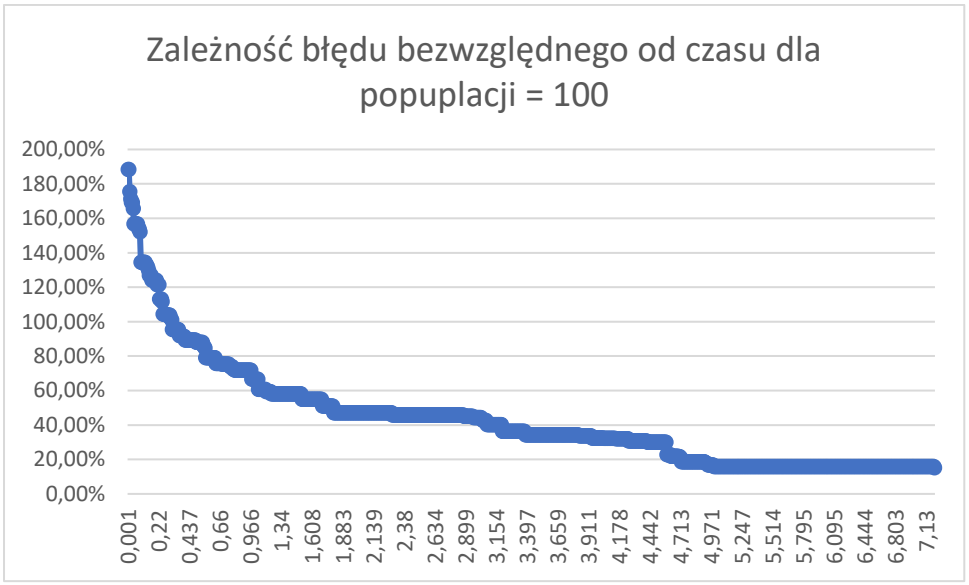


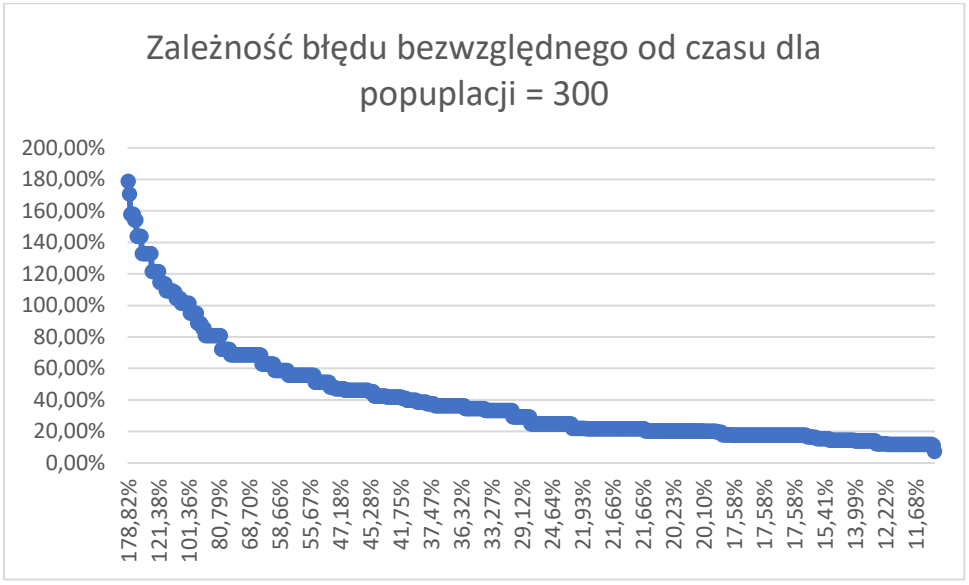




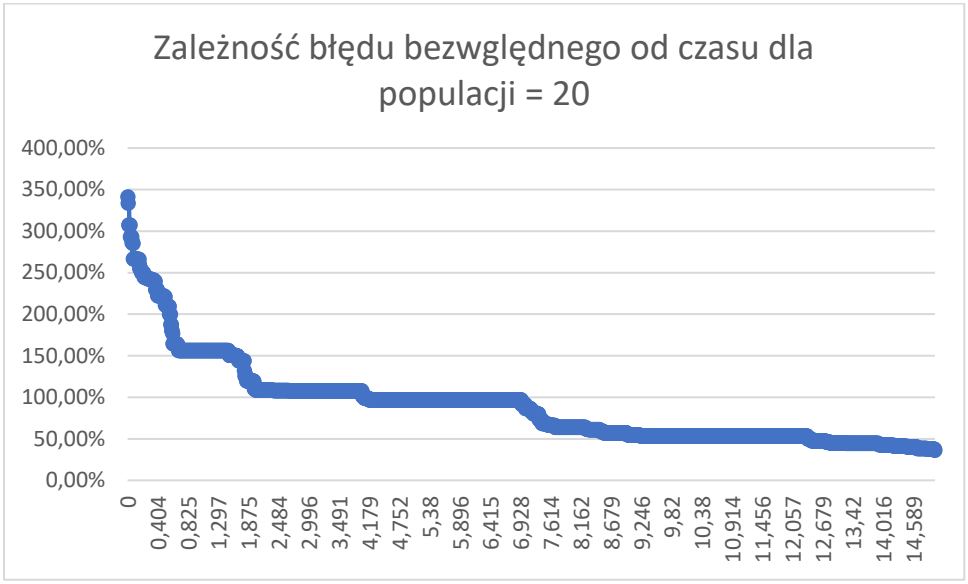
ftv35


Zależność błędu bezwzględnego od czasu dla popuplacji = 20
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0
0,048
0,106
0,152
0,196
0,241
0,285
0,33
0,378
0,423
0,472
0,518
0,566
0,614
0,661
0,707
0,754
0,803
0,851
0,898
0,946
1,004
1,099
1,187
1,277

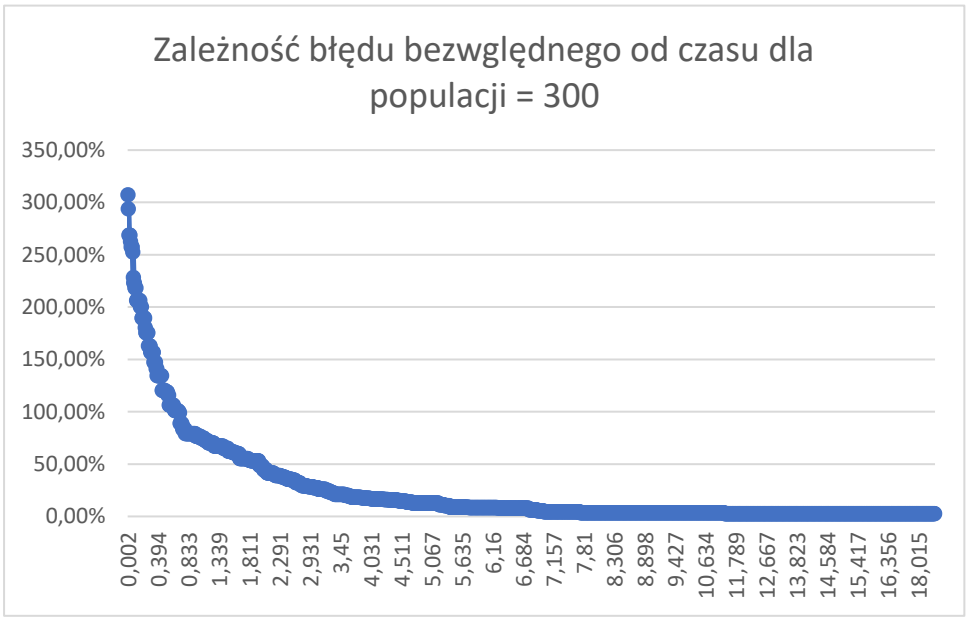

Zależność błędu bezwzględnego od czasu dla popuplacji = 100
200,00%
180,00%
160,00%
140,00%
120,00%
100,00%
80,00%
60,00%
40,00%
20,00%
0,00%
0,001
0,22
0,437
0,66
0,966
1,34
1,608
1,883
2,139
2,38
2,634
2,899
3,154
3,397
3,659
3,911
4,178
4,442
4,713
4,971
5,247
5,514
5,795
6,095
6,444
6,803
7,13


Zależność błędu bezwzględnego od czasu dla popuplacji = 300
200,00%
180,00%
160,00%
140,00%
120,00%
100,00%
80,00%
60,00%
40,00%
20,00%
0,00%
178,82%
121,38%
101,36%
80,79%
68,70%
58,66%
55,67%
47,18%
45,28%
41,75%
37,47%
36,32%
33,27%
29,12%
24,64%
21,93%
21,66%
21,66%
20,23%
20,10%
17,58%
17,58%
17,58%
15,41%
13,99%
12,22%
11,68%

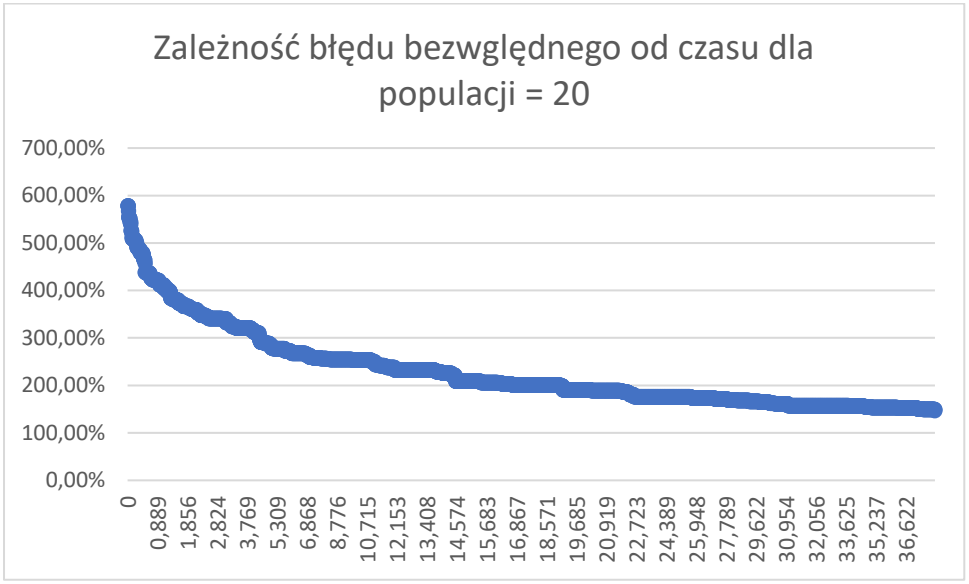
data58

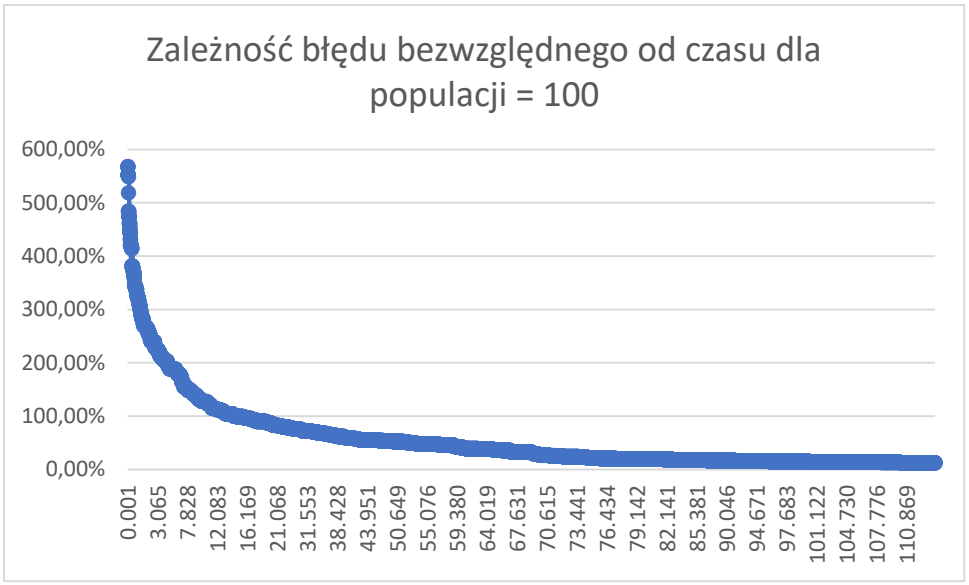

Zależność błędu bezwzględnego od czasu dla populacji = 20
400,00%
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0
0,404
0,825
1,297
1,875
2,484
2,996
3,491
4,179
4,752
5,38
5,896
6,415
6,928
7,614
8,162
8,679
9,246
9,82
10,38
10,914
11,456
12,057
12,679
13,42
14,016
14,589

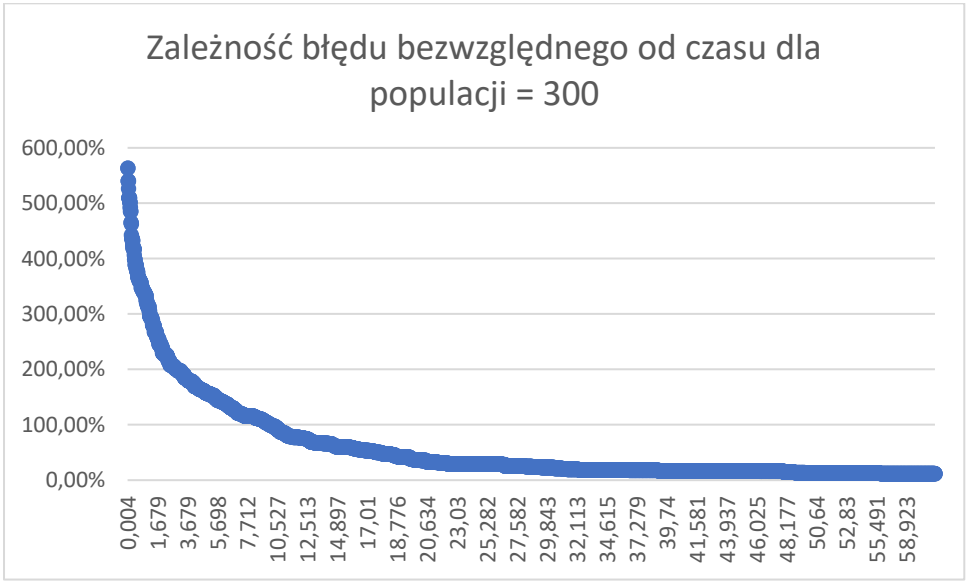

Zależność błędu bezwzględnego od czasu dla populacji = 100
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0,001
0,672
1,398
2,117
2,909
3,78
4,794
5,609
6,381
7,926
9,518
10,966
11,96
12,88
13,789
14,781
16,363
17,931
19,051
20,042
21,054
21,968
22,778
23,552
24,295
25,3
26,514
27,69
28,527


Zależność błędu bezwzględnego od czasu dla populacji = 300
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0,002
0,394
0,833
1,339
1,811
2,291
2,931
3,45
4,031
4,511
5,067
5,635
6,16
6,684
7,157
7,81
8,306
8,898
9,427
10,634
11,789
12,667
13,823
14,584
15,417
16,356
18,015

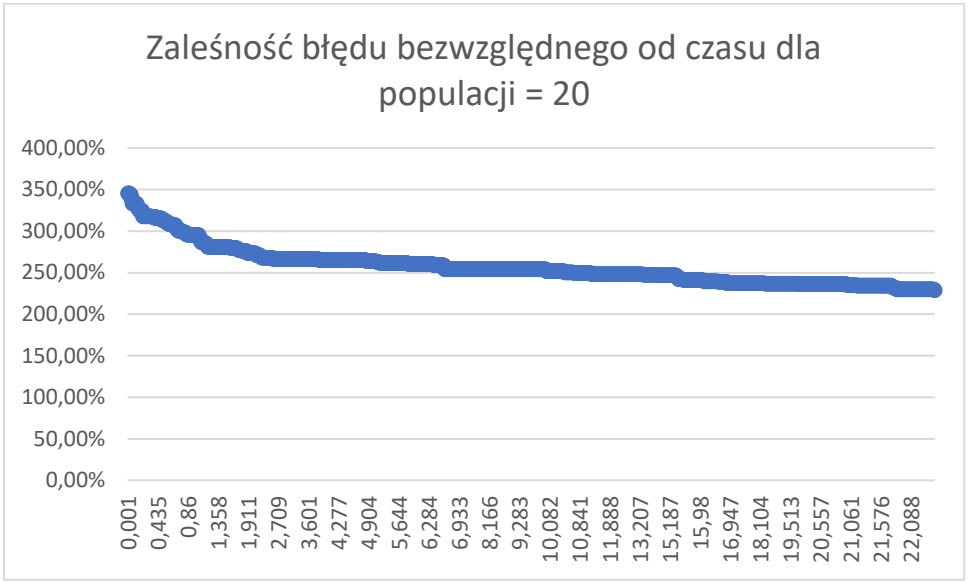
data120

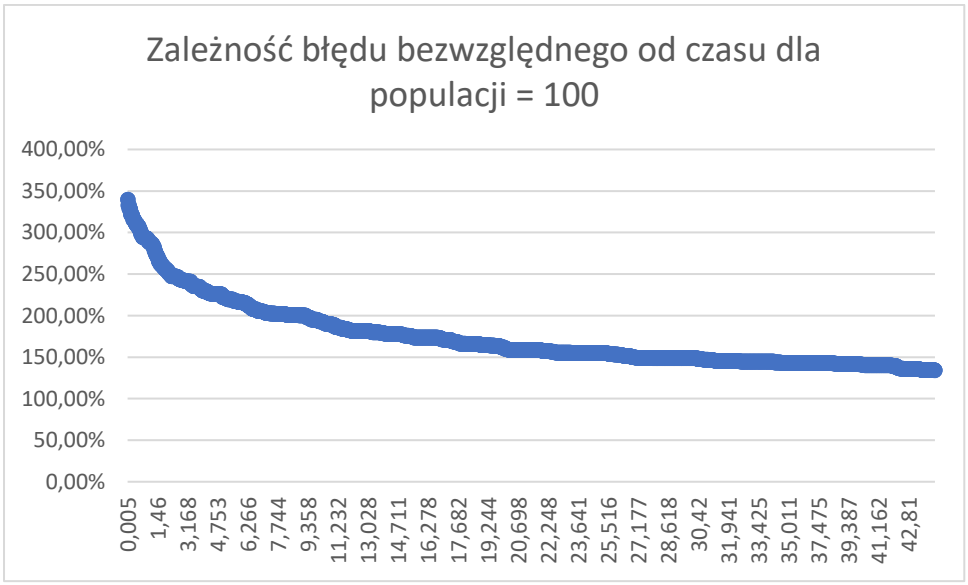

Zależność błędu bezwzględnego od czasu dla populacji = 20
700,00%
600,00%
500,00%
400,00%
300,00%
200,00%
100,00%
0,00%
0
0,889
1,856
2,824
3,769
5,309
6,868
8,776
10,715
12,153
13,408
14,574
15,683
16,867
18,571
19,685
20,919
22,723
24,389
25,948
27,789
29,622
30,954
32,056
33,625
35,237
36,622

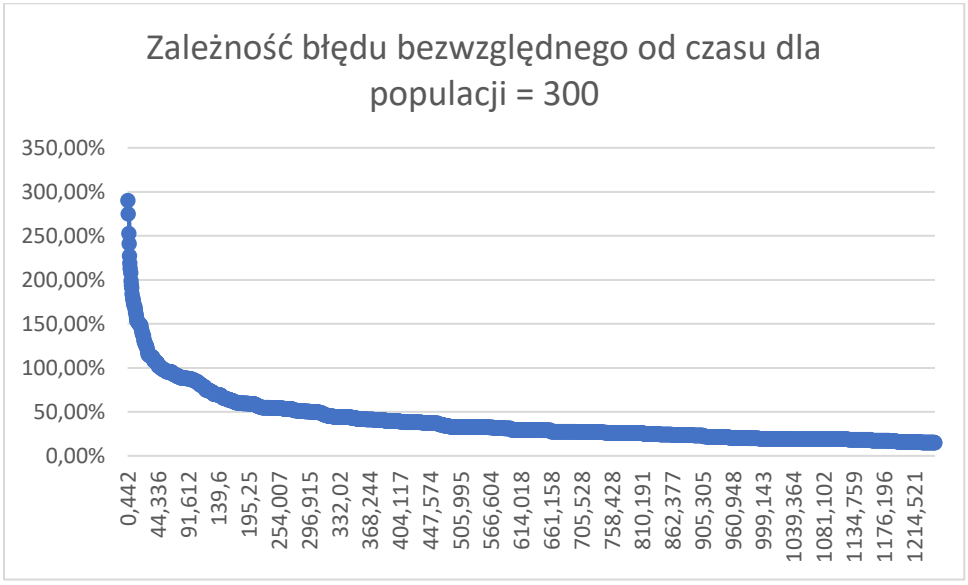

Zależność błędu bezwzględnego od czasu dla populacji = 100
600,00%
500,00%
400,00%
300,00%
200,00%
100,00%
0,00%
0.001
3.065
7.828
12.083
16.169
21.068
31.553
38.428
43.951
50.649
55.076
59.380
64.019
67.631
70.615
73.441
76.434
79.142
82.141
85.381
90.046
94.671
97.683
101.122
104.730
107.776
110.869


Zależność błędu bezwzględnego od czasu dla populacji = 300
600,00%
500,00%
400,00%
300,00%
200,00%
100,00%
0,00%
0,004
1,679
3,679
5,698
7,712
10,527
12,513
14,897
17,01
18,776
20,634
23,03
25,282
27,582
29,843
32,113
34,615
37,279
39,74
41,581
43,937
46,025
48,177
50,64
52,83
55,491
58,923

data323

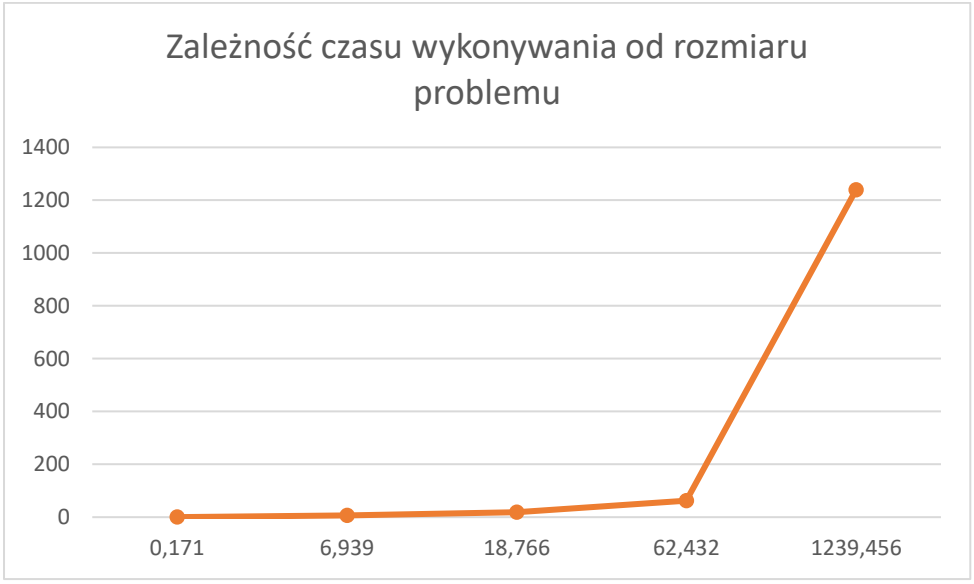

Zaleśność błędu bezwzględnego od czasu dla populacji = 20
400,00%
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0,001
0,435
0,86
1,358
1,911
2,709
3,601
4,277
4,904
5,644
6,284
6,933
8,166
9,283
10,082
10,841
11,888
13,207
15,187
15,98
16,947
18,104
19,513
20,557
21,061
21,576
22,088


Zależność błędu bezwzględnego od czasu dla populacji = 100
400,00%
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0,005
1,46
3,168
4,753
6,266
7,744
9,358
11,232
13,028
14,711
16,278
17,682
19,244
20,698
22,248
23,641
25,516
27,177
28,618
30,42
31,941
33,425
35,011
37,475
39,387
41,162
42,81


Zależność błędu bezwzględnego od czasu dla populacji = 300
350,00%
300,00%
250,00%
200,00%
150,00%
100,00%
50,00%
0,00%
0,442
44,336
91,612
139,6
195,25
254,007
296,915
332,02
368,244
404,117
447,574
505,995
566,604
614,018
661,158
705,528
758,428
810,191
862,377
905,305
960,948
999,143
1039,364
1081,102
1134,759
1176,196
1214,521

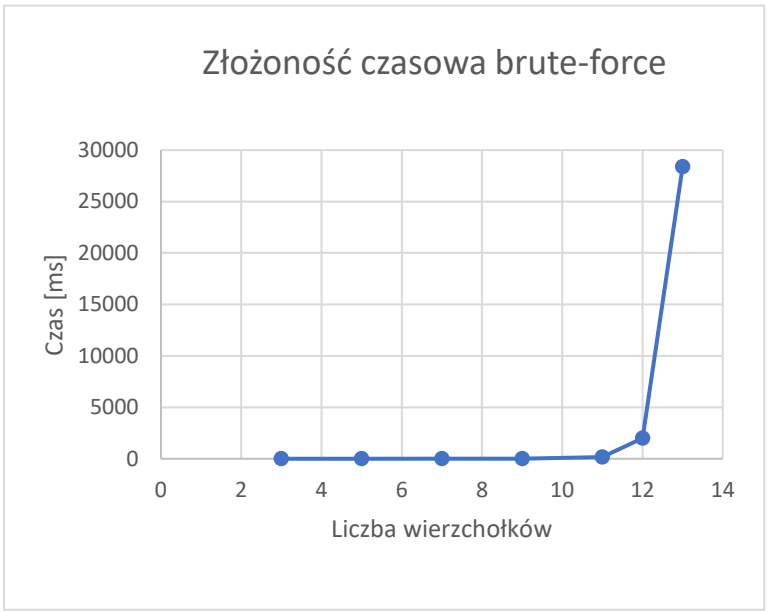
Zestawmy czas po jakim algorytm się kończy dla populacji = 300

Rozmiar problemu	Czas	Błąd względny
17	0,171 ms	0%
35	6,939 ms	7,26%
58	18,766 ms	2,38%
120	62,432 ms	11,39%
323	1239,456 ms	14,56%

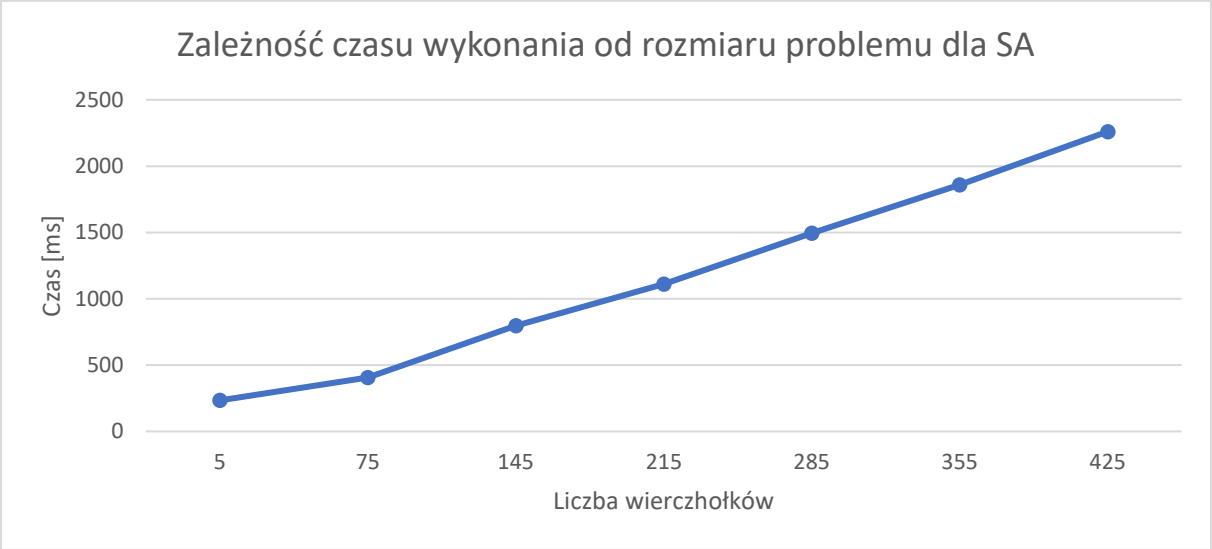


5. Porównanie z innymi algorytmami

W przypadku metody brute-force oraz programowania dynamicznego porównywać możemy tylko mniejsze instancje problemów, jednak nawet takie porównanie może nam dużo powiedzieć.



Jak widzimy, algorytm genetyczny działa dużo szybciej niż algorytmy przeszukiwania zupełnego. Należy jednak zwrócić uwagę, że to w tych drugich mamy pewność że znalezione rozwiązanie jest rozwiązaniem optymalnym.



Tutaj również algorytm genetyczny lepiej sprawdza się w kwestii złożoności czasowej. Jednak dla dużych instancji problemu symulowane wyżarzanie może dać bardziej optymalne wyniki. Dla przykładu: dla instancji o rozmiarze 323 miast SA daje wynik z błędem względnym na poziomie 8,73% (choć z większym czasem), kiedy algorytm genetyczny aż 14,56%. Prawdopodobnie można ten wynik ulepszyć zmieniając parametry algorytmu lub zmieniając warunek stopu, pozwalając algorytmowi działać dłużej.

6. Wnioski

Algorytm genetyczny ma stosunkowo niską złożoność czasową, co czyni go odpowiednim do wykorzystania w rozwiązaniu TSP. Niestety, ma też kilka wad. Co oczywiste, nie daje rozwiązania optymalnego. Jest też mocno zależny od instancji problemu: aby go ulepszyć, należałoby zbadać konkretną instancję problemu i docelowo do niej dostosować implementację poszczególnych elementów algorytmu jak sposób krzyżowania, mutacji czy selekcji.