Projektowanie efektywnych algorytmów

Łukasz Mieczyński 256764

Zadanie projektowe nr 3

Grupa Wtorek 17:05

Prowadzący mgr. inż. Antoni Sterna

Plan projektu

Projekt ma na celu zaimplementowanie struktury grafu i przeprowadzenie na nim testów za pomocą algorytmów dla problemu komiwojażera(TSP). Problem komiwojażera, jest to zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Wynikami testów będą czasy wykonania oraz znalezione drogi dla algorytmu genetycznego. Graf zostanie zaimplementowany jako macierz sąsiedztwa.

Program został napisany w języku C++, struktury są alokowane dynamicznie. Grafy są traktowane jako grafy nieskierowane.

Opis algorytmu genetycznego

1. Wygenerowanie losowej populacji startowej
2. Wybór osobników do dalszej reprodukcji(selekcja)
3. Krzyżowanie genotypów rodziców i mutacja powstałych dzieci
4. Powstanie nowej populacji składającej się z dzieci oraz osobników z poprzedniej populacji
5. Ocena nowej populacji, znalezienie najlepszego osobnika z nowej populacji, jeśli znalezione rozwiązanie jest lepsze, zapamiętanie go.
6. Powrót do punktu 2.

Koniec algorytmu następuje na podstawie kryterium stopu, które ustalane jest przez programistę.

Wygenerowanie populacji startowej

Generowanie populacji startowej polega na wylosowaniu tylu permutacji, z ilu ma składać się cała populacja, i dodaniu ich do wektora przechowującego informacje o populacji.

```
//losowanie permutacji
void GA::generatePermutation(vector<int>& permutation)
{
    for (int i = 0; i < countOfCities; i++)
        permutation[i] = i;

    random_shuffle(permutation.begin(),permutation.end());
    permutation.push_back(permutation.at(0));

}

//wybór populacji startowej
void GA::getStartedPopulation()
{
    for (int i = 0; i < populationSize; i++)
    {
        vector<int> permutation(countOfCities);
        generatePermutation(permutation);
        population.push_back(permutation);
    }

}
```

Obraz 1. Losowanie permutacji i dodawanie ich do populacji startowej

Wybór osobników do dalszej reprodukcji

Wybór rodziców polega na podział całej populacji na 4 grupy. Z każdej grupy będzie pochodzić dany % rodziców do dalszej reprodukcji. Przydział do grup jest zależny od długości ścieżki jaką posiada każdy osobnik. Dzięki takiemu rozwiązaniu, rodzicami nie zostają tylko najlepsze, ale również gorsza część osobników w populacji. Powoduje to większą różnorodność populacji.

```
void GA::getParents()
{

    srand(time(NULL));
    parents.clear();
    vector <pair<int, int>> valueOfEachPermutation;
    //podzielenie populacji na 4 grupy w zależności od długości ścieżki
    int countOfRanks = 4;
    int sizeOfRanks = populationSize / countOfRanks;
    int cost;

    //przypisanie każdej grupie liczbę osobników, które będą rodzicami
    vector <int>ranks;
    //z pierwszej grupy, będzie pochodzić 55% rodziców
    int percent = 0.55 * parentsPopulationSize;
    ranks.push_back( percent);
    //z drugiej grupy, będzie pochodzić 30% rodziców
    percent = 0.3 * parentsPopulationSize;
    ranks.push_back(percent);
    //z trzeciej grupy, będzie pochodzić 10% rodziców
    percent = 0.1* parentsPopulationSize;
    ranks.push_back(percent);
    //z czwartej grupy, będzie pochodzić 5% rodziców
    percent = 0.05 * parentsPopulationSize;
    ranks.push_back(percent);
    int sumOfPercent = 0;


    //zabezpieczenie dla różnicy między zadaną populacją rodziców
    //a liczbą rodziców wynikającą z poszczególnych grup
    for (int i = 0; i < countOfRanks; i++)
        sumOfPercent = sumOfPercent + ranks[i];

    while (sumOfPercent != parentsPopulationSize) {
        ranks[0]++;
        sumOfPercent++;
    }
```

Obraz 2. Metoda wybierająca rodziców

```
//sortowanie osobników wg długości ścieżki
sort(valueOfEachPermutation.begin(), valueOfEachPermutation.end(), [this](const pair<int, int>& a,
    const pair<int, int>& b)->bool {return a.second < b.second; });


int k = 0;
int j = 0;
int countOfParentsPerRank;

//losowanie rodziców z poszczególych grup rankingowych i dodawanie ich do wektora rodziców
for (int i = 0; i < countOfRanks; i++)
{
    countOfParentsPerRank = ranks.at(i);

    for (int p : randParents(j, j + sizeOfRanks, countOfParentsPerRank))
    {

        parents.push_back(population[valueOfEachPermutation.at(p).first]);

    }
    j =j + sizeOfRanks;

}
```

Obraz 3. Metoda wybierająca rodziców

```
//funkcja losojąca zadaną liczbę osobników, zadanego przedziału
vector <int> GA::randParents(int min, int max, int count)
{
    srand(time(NULL));
    vector < int > vect;
    vect.reserve(count);

    if (count > max) {
        for (int i = 0; i < max; i++)
            vect.push_back(i);

        do
        {

            vect.push_back(min);
            min++;
        } while (min != count - max);

    }
    else
    {
    do
    {

        vect.push_back(min);
        min++;
    } while (min != max);
}
    random_shuffle(vect.begin(), vect.end());


    int size = vect.size();

    for (int i = 0; i < size - count; i++)
        vect.pop_back();


    return vect;
```

Obraz 3. Metoda losująca zadaną liczbę rodziców, z danej grupy rankingowej

Krzyżowanie i mutacja

Metoda krzyżowania jaka została zaimplementowana to metoda opierająca się na idei Order Crossover(OX, Davis, 1985). Natomiast metoda mutacji polega na wylosowaniu segmentu, w którym losowo zostają zamienione miasta. Jest to tzw. Scramble Mutation.

```
void GA::crossingAndMutate()
{

    //wybór rodziców
    vector<int> parent1 = parents.at(rand() % parentsPopulationSize);
    vector<int> parent2 = parents.at(rand() % parentsPopulationSize);

    vector<int> child1(countOfCities);
    vector<int> child2(countOfCities);


    //losowanie początku i końca segmentu
    int startOfSegment;
    int endOfSegment;

    do {
        startOfSegment = rand() % countOfCities;
        endOfSegment = rand() % countOfCities;
    } while (startOfSegment == endOfSegment);


    if (startOfSegment > endOfSegment) {
        int temp = startOfSegment;
        startOfSegment = endOfSegment;
        endOfSegment = temp;
    }

    //skopiowanie danego segmentu od rodziców do dzieci
    for (int i = startOfSegment; i <= endOfSegment; i++) {
        child1[i] = parent1[i];
        child2[i] = parent2[i];

    }
    vector<int> temp1;
    vector<int> temp2;
    temp1.reserve(countOfCities);
    temp2.reserve(countOfCities);
```

Obraz 4. Metoda tworzenia dzieci na podstawie krzyżowania i mutacji

```
int pos = endOfSegment;

//skopiowanie kolejności miast zaczynając od końca segmentu
for (int i = 0; i < countOfCities; i++)
{
    if (pos == countOfCities-1)
        pos = 0;
    else
        pos++;


    temp1.push_back(parent1.at(pos));
    temp2.push_back(parent2.at(pos));

}



//sprawdzenie czy miasto znajduje się w segmencie oraz wstawienie reszty miast do potomka
for (int i = 0; i < countOfCities; i++)
{
    isOnList1 = false;
    isOnList2 = false;

    for (int j = startOfSegment; j <= endOfSegment; j++) {
        if (temp1[i] == child2[j])
            isOnList1 = true;
        if (temp2[i] == child1[j])
            isOnList2 = true;


    }

    if (isOnList1)
        temp1[i] = -1;
    if (isOnList2)
        temp2[i] = -1;

}
```

Obraz 5. Metoda tworzenia dzieci na podstawie krzyżowania i mutacji

```
    pos = endOfSegment + 1;
for (int i = 0; i < countOfCities; i++) {
    if (pos == countOfCities )
        pos = 0;

    if (temp2[i] != -1)
    {
        child1[pos] = temp2[i];
        pos++;
    }


}


pos = endOfSegment + 1;

for (int i = 0; i < countOfCities; i++) {
    if (pos == countOfCities)
        pos = 0;

    if (temp1[i] != -1)
    {
        child2[pos] = temp1[i];
        pos++;
    }

}
```

Obraz 6. Metoda tworzenia dzieci na podstawie krzyżowania i mutacji

```
//wylosowanie 2 liczb
double r1 = (double)rand() / (double)RAND_MAX;
double r2 = (double)rand() / (double)RAND_MAX;

//jeśli  wylosowana liczba jest mniejsza od prawdopodobieństwa mutacji, to następuje mutacja
if (r1 < mutationProbability)
    mutate(child1);

if (r2 < mutationProbability)
    mutate(child2);

//dodanie powrotu do wierzchołka startowego
child1.push_back(child1[0]);
child2.push_back(child2[0]);

//dodanie nowego dziecka do wektora dzieci
childrens.push_back(child1);
childrens.push_back(child2);

temp1.resize(0);
temp2.resize(0);
child1.resize(0);
child2.resize(0);
```

Obraz 7. Metoda tworzenia dzieci na podstawie krzyżowania i mutacji

```
//metoda mutacji, Scramble mutation
void GA::mutate(vector<int>& child)
{
    int startOfSegment;
    int endOfSegment;

    do {
        startOfSegment = rand() % countOfCities;
        endOfSegment = rand() % countOfCities;
    } while (startOfSegment == endOfSegment);

    if (startOfSegment > endOfSegment) {
        int temp = startOfSegment;
        startOfSegment = endOfSegment;
        endOfSegment = temp;

    }


    random_shuffle(child.begin()+startOfSegment , child.end() - (child.size() - endOfSegment) );

}
```

Obraz 8. Metoda mutacji dziecka

Utworzenie nowej populacji

Nowa populacja tworzona jest na podstawie % nowych osobników, zależnego od współczynnika krzyżowania oraz reszty osobników ze starej populacji. Liczba osobników w populacji jest z góry założona. Do nowej populacji dodani są najlepsi osobnicy ze starej populacji.

```
vector<vector<int>>  GA::getNewPopulation()
{
    vector<vector<int>> newPopulation;
    newPopulation.reserve(populationSize);

    //sortowanie starej populacji
    sort(population.begin(), population.end(), [this](auto i, auto j)->bool {return getCostOfPath(i) > getCostOfPath(j); });


    //dodanie dzieci do populacji
    // liczba dzieci stanowi %(crossoverProbability) nowej populacji
    for (int i = 0; i < crossoverProbability * populationSize; i++) {
        newPopulation.push_back(childrens.back());
        childrens.pop_back();

    }

        int size = newPopulation.size();

    //dodanie najlepszych osobników ze starej populacji
    for (int i = 0; i < populationSize - size; i++)
    {
        newPopulation.push_back(population.back());
        population.pop_back();
    }

        return newPopulation;

}
```

Obraz. 9 Metoda tworząca nową populacje

Ogólny przebieg algorytmu

Algorytm ma przebieg opisany na samym początku. Wynikiem algorytmu jest najkrótsza ścieżka jaka została znaleziona.

```
Graph GA::GeneticAlgorithm(Graph graph, int endTime, double mutationProb, double crossoverProb, int size)
{
    srand(time(NULL));;
    this->graph = graph;
    countOfCities = graph.getSize();
    populationSize = size;
    mutationProbability = mutationProb;
    crossoverProbability = crossoverProb;

    double time = 0;    //zmienna licząca czas trwania algorytmu
    initialization();   //inicjalizacja wektorów
    getStartedPopulation(); // losowewybranie początkowej populacji
    vector<int> currnetOrder;
    int currentBest;
    clock_t begin = clock();



    while (time < endTime) {

            getParents(); //wybór rodziców

        for (int i = 0; i < (populationSize * crossoverProbability) / 2; i++)
            crossingAndMutate(); // krzyżowanie i mutacja powstałych dzieci
        population = getNewPopulation();    //utworzenie nowej populacji z powstałych dzieci i reszty populacji

        //znalezienie najlepszego osobnika z nowej populacji
        currnetOrder = getBestSolutionFromPopulation();
        currentBest = getCostOfPath(currnetOrder);

        //sprawdzenie czy nowe rozwiązanie jest lepsze, jeśli tak zapamiętanie go
        if (currentBest < bestSolution)
        {
            bestSolution = currentBest;
            bestOrder = currnetOrder;
            costToPrint.push_back(bestSolution);
            timeToPrint.push_back(time);
        }
        clock_t end = clock();
        double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
        time = elapsed_secs;

    }

    population.clear();
    childrens.clear();
    parents.clear();
    graph.updateSolution(bestOrder);
    return  graph;
}
```

Obraz 10. Zaimplementowany algorytm genetyczny

Wyniki testów

Sposób testowania

Testy zostały przeprowadzone dla 3 różnych plików. Dla każdego pliku przeprowadzono testy z dywersyfikacją oraz bez. Parametry dla każdego testu zostały dobrane eksperymentalnie. Każdy test trwał 120 s. Wynikami testów będą wykresy zależności błędu względnego w funkcji czasu algorytmu. Błąd względny to:

$$|f - f_{opt}|/ f_{opt}$$

Gdzie:

f – wartość obliczona przez algorytm

f_{opt} – wartość optymalna

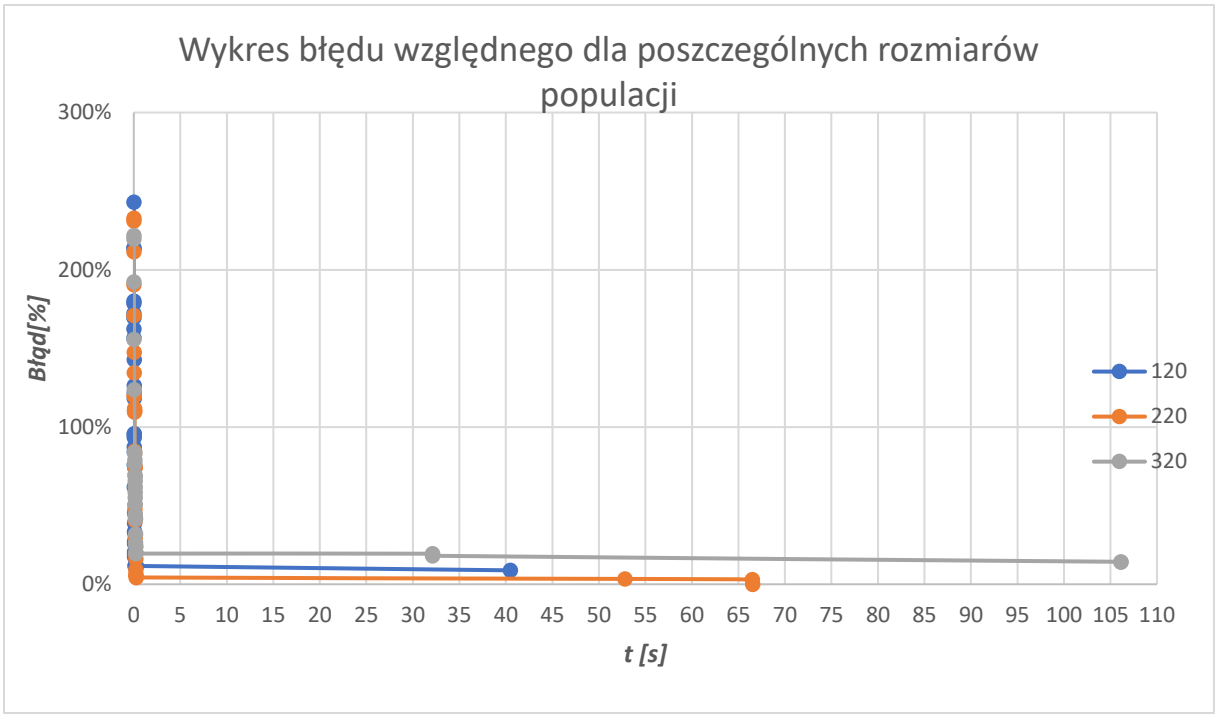
```
GA::GA()
{
    populationSize = 6000;      //musi być podzielna przez 4
    crossoverProbability = 0.8;
    parentsPopulationSize =0.75*populationSize;
    mutationProbability = 0.01;

}
```

Obraz 11. Parametry testowania

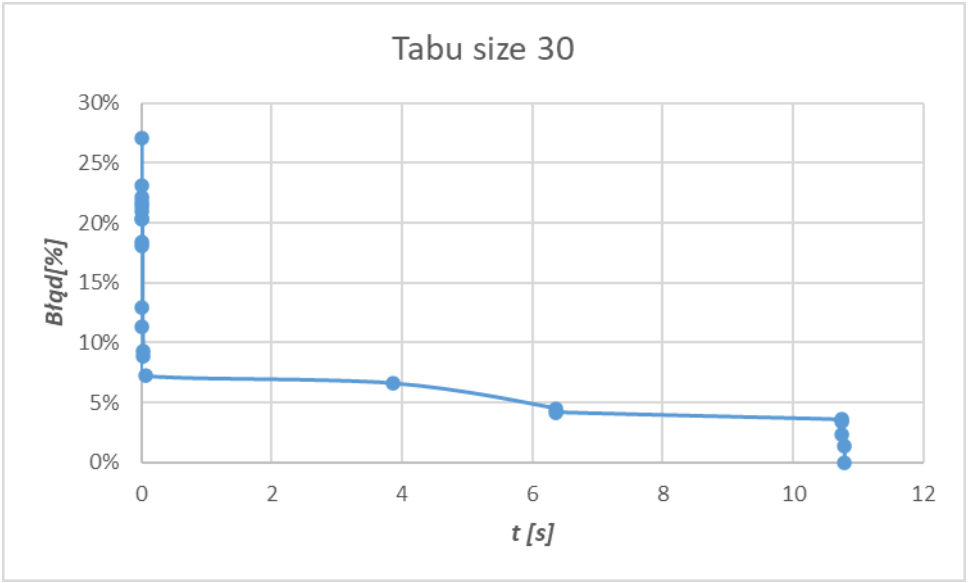
Wielkość populacji musi być podzielna przez 4, ponieważ wymaga tego wybrany sposób krzyżowania rodziców.

Wyniki testów dla pliku ftv33.txt

Rozmiar populacji	***120***		***220***		***320***	
	Błąd [%]	*t [s]*	*Błąd [%]*	*t [s]*	*Błąd [%]*	*t [s]*
1	243%	4412	233%	4281	222%	4140
2	214%	4037	231%	4258	220%	4110
3	214%	4032	212%	4006	192%	3759
4	180%	3603	191%	3736	156%	3288
5	179%	3585	171%	3486	124%	2878
6	172%	3501	148%	3185	84%	2366
7	170%	3467	135%	3016	83%	2353
8	162%	3374	120%	2834	79%	2304
9	157%	3303	120%	2826	77%	2273
10	143%	3123	112%	2722	70%	2180
11	126%	2912	110%	2705	69%	2172
12	122%	2855	110%	2704	66%	2132
13	118%	2808	109%	2694	62%	2082
14	96%	2518	85%	2376	58%	2036
15	95%	2510	75%	2253	55%	1995
16	93%	2487	74%	2244	51%	1941
17	87%	2408	74%	2239	44%	1858
18	84%	2369	74%	2237	41%	1819
19	76%	2263	51%	1937	32%	1696
20	62%	2083	48%	1899	26%	1625
21	45%	1869	44%	1849	24%	1594
22	39%	1785	41%	1808	20%	1545
23	34%	1718	30%	1667	20%	1542
24	33%	1709	20%	1538	20%	1537
25	27%	1637	16%	1495	19%	1536
26	26%	1618	9%	1405	18%	1519
27	21%	1553	6%	1367	14%	1470
28	19%	1535	6%	1360	14%	1469
29	18%	1512	4%	1342		
30	12%	1438	3%	1330		
31	12%	1436	3%	1324		
32	9%	1400	0%	1286		



Wykres 1. Wykres błędu względnego dla 3 różnych populacji

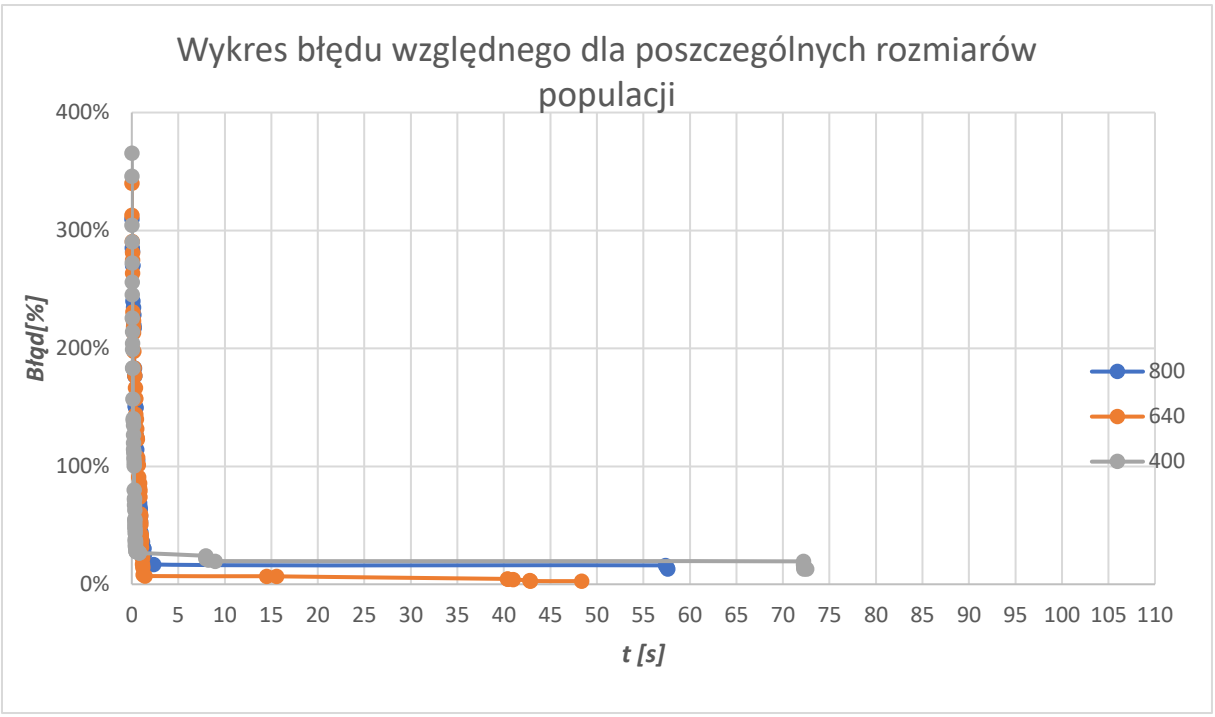


Wykres 2. Wykres błędu względnego dla najlepszego rozwiązania przez algorytm Tabu Search

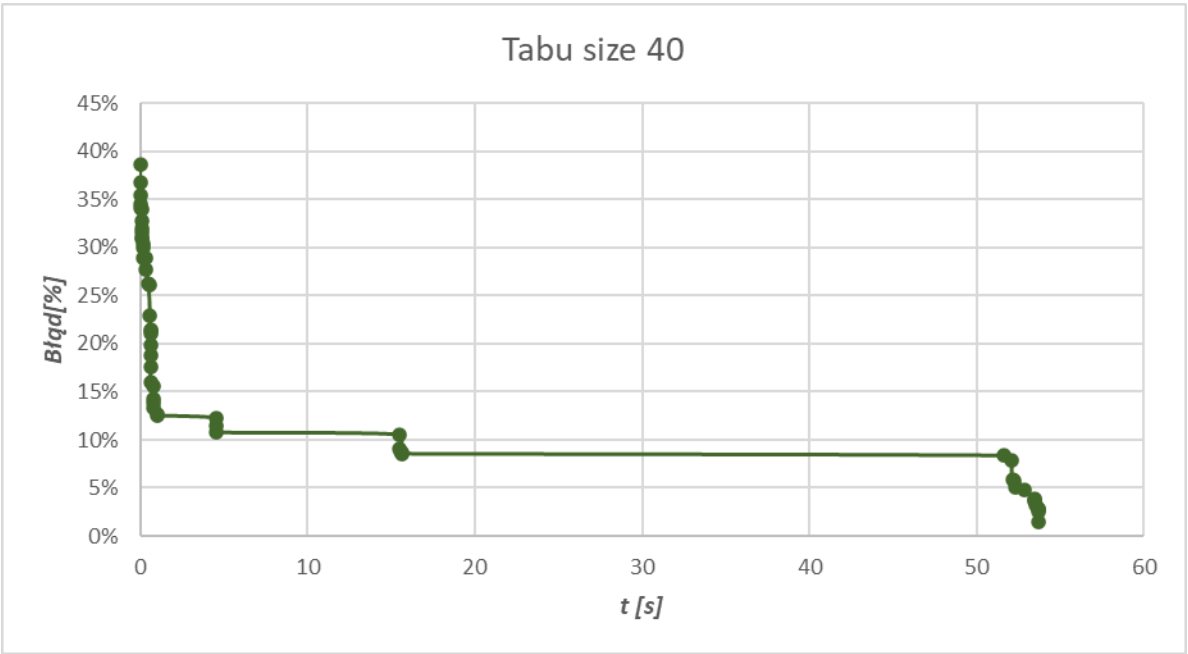
Wyniki testów dla pliku ftv55.txt

Rozmiar populacji	***400***		***640***		***800***	
	Błąd [%]	*t [s]*	*Błąd [%]*	*t [s]*	*Błąd [%]*	*t [s]*
1	366%	0	340%	0	310%	0
2	346%	0,005	313%	0,009	285%	0,032
3	305%	0,009	291%	0,033	282%	0,053
4	291%	0,02	281%	0,063	271%	0,074
5	272%	0,028	275%	0,073	270%	0,095
6	256%	0,04	264%	0,081	240%	0,107
7	246%	0,044	231%	0,094	235%	0,179
8	226%	0,051	226%	0,121	228%	0,202
9	214%	0,061	223%	0,163	220%	0,215
10	204%	0,065	218%	0,173	217%	0,249
11	199%	0,072	213%	0,183	183%	0,263
12	183%	0,075	198%	0,202	177%	0,307
13	183%	0,104	184%	0,244	151%	0,356
14	157%	0,107	177%	0,307	150%	0,39
15	141%	0,131	167%	0,366	150%	0,445
16	139%	0,142	158%	0,384	127%	0,456
17	135%	0,164	157%	0,404	114%	0,508
18	127%	0,168	144%	0,422	107%	0,553
19	120%	0,174	140%	0,484	105%	0,603
20	114%	0,18	132%	0,509	83%	0,714
21	113%	0,192	125%	0,532	80%	0,801
22	111%	0,199	124%	0,558	69%	0,832
23	107%	0,211	123%	0,584	66%	0,856
24	103%	0,225	107%	0,605	64%	0,902
25	100%	0,231	101%	0,686	58%	0,913
26	80%	0,239	91%	0,711	53%	0,944
27	73%	0,268	86%	0,819	44%	0,979
28	72%	0,28	81%	0,834	36%	sty.00
29	67%	0,287	79%	0,863	31%	1,261
30	63%	0,304	74%	0,878	30%	1,278
31	55%	0,308	60%	0,9	25%	1,345
32	54%	0,312	59%	0,914	17%	2,347
33	51%	0,319	58%	0,93	16%	57,361
34	48%	0,325	53%	0,944	14%	57,549
35	46%	0,335	51%	0,975	13%	57,621
36	43%	0,339	42%	0,984		
37	38%	0,351	41%	1		
38	37%	0,362	37%	1,028		
39	37%	0,374	36%	1,036		
40	35%	0,376	31%	1,043		
41	34%	0,384	23%	01.sty		
42	33%	0,392	22%	1,127		
43	32%	0,396	17%	1,145		

44	30%	0,41	14%	1,168		
45	28%	0,433	8%	1,204		
46	28%	0,444	7%	1,424		
47	28%	0,456	7%	14,494		
48	27%	0,764	7%	15,55		
49	27%	0,793	5%	40,358		
50	27%	0,803	4%	40,444		
51	24%	7,934	4%	40,974		
52	23%	7,947	3%	42,776		
53	22%	7,957	3%	42,849		
54	21%	8,214	3%	48,339		
55	20%					
56	19%					
57	17%					
58	16%					
59	14%					
60	13%					
61	13%					



Wykres 2. Wykres błędu względnego dla 3 różnych populacji



Wykres 3. Wykres błędu względnego dla najlepszego rozwiązania przez algorytm Tabu Search

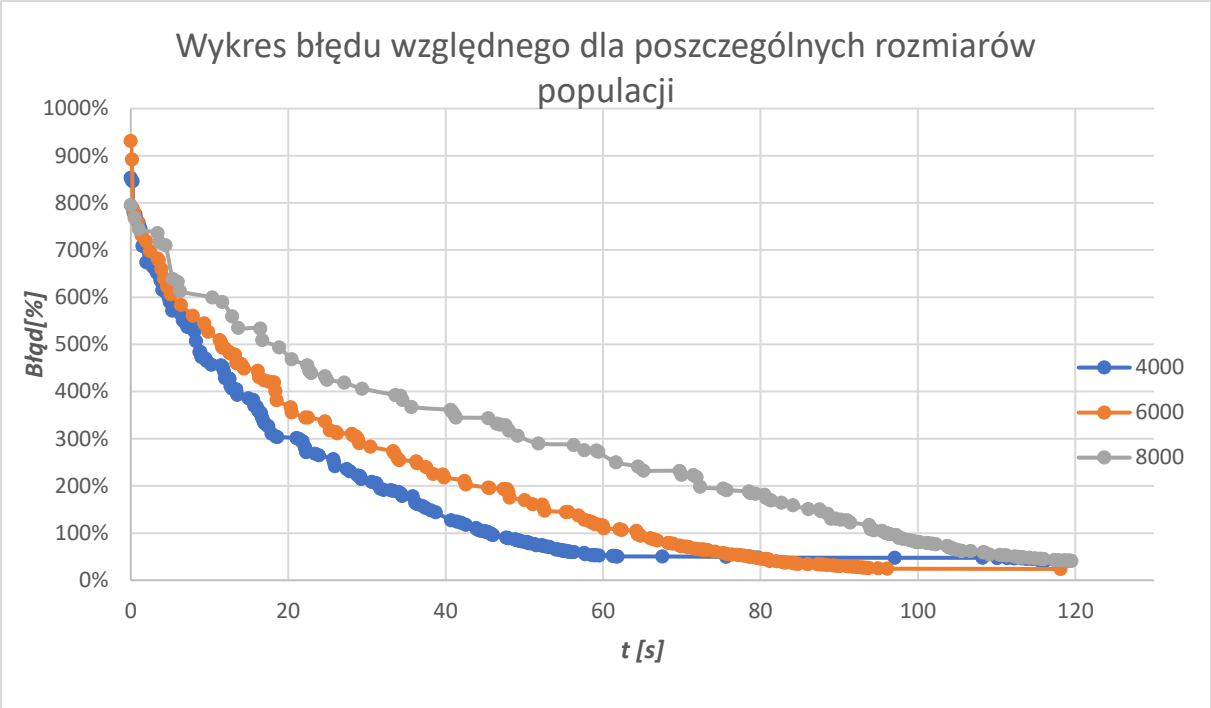
Wyniki testów dla pliku ftv170.txt

Rozmiar populacji	***4000***		***6000***		***8000***	
	Błąd [%]	*t [s]*	*Błąd [%]*	*t [s]*	*Błąd [%]*	*t [s]*
1	853%	0	931%	0	796%	0
2	849%	0,112	892%	0,138	768%	0,484
3	846%	0,222	787%	0,274	745%	0,994
4	780%	0,339	779%	0,426	736%	3,377
5	776%	0,588	768%	0,568	716%	3,644
6	766%	0,699	760%	0,863	710%	4,405
7	758%	0,986	732%	1,366	639%	5,312
8	745%	1,244	719%	01.sty	632%	5,971
9	737%	1,359	698%	2,409	613%	6,238
10	708%	1,489	682%	3,398	600%	10,364
11	674%	1,993	678%	3,563	590%	11,614
12	667%	2,757	660%	3,894	559%	12,889
13	663%	2,866	639%	4,228	535%	13,656
14	652%	3,306	637%	4,566	534%	16,468
15	636%	3,748	623%	04.sty	509%	16,708
16	634%	3,875	606%	5,066	494%	18,867
17	616%	3,994	584%	6,385	469%	20,423
18	610%	4,456	561%	7,863	455%	22,438
19	599%	4,822	545%	9,354	444%	22,679
20	590%	4,935	527%	9,833	439%	22,927
21	572%	5,269	510%	11,288	433%	24,666
22	565%	6,307	504%	11,455	425%	24,903
23	551%	6,625	494%	11,622	419%	27,091
24	550%	6,732	492%	11,945	406%	29,382
25	549%	6,841	485%	12,428	393%	33,605
26	544%	7,058	481%	12,585	391%	34,258
27	537%	7,167	479%	13,061	382%	34,493
28	536%	7,977	478%	13,231	368%	35,645
29	528%	8,082	460%	13,386	361%	40,638
30	507%	8,293	458%	14,059	356%	40,863
31	484%	8,723	453%	14,227	351%	41,088
32	484%	8,828	449%	14,384	345%	41,31
33	473%	8,959	444%	16,143	343%	45,389
34	470%	9,497	430%	16,302	332%	46,484
35	465%	9,603	424%	16,951	330%	46,913
36	457%	10,232	421%	17,579	329%	47,617
37	456%	11,466	420%	18,213	317%	48,065
38	453%	11,651	402%	18,375	306%	49,16
39	444%	11,826	382%	18,545	290%	51,783
40	429%	11,978	367%	20,296	287%	56,266
41	428%	12,454	356%	20,456	276%	57,6
42	427%	12,562	345%	22,19	275%	59,16
43	412%	12,666	345%	22,498	272%	59,381
44	410%	12,772	337%	24,654	250%	61,655
45	406%	12,878	318%	25,281	241%	64,45

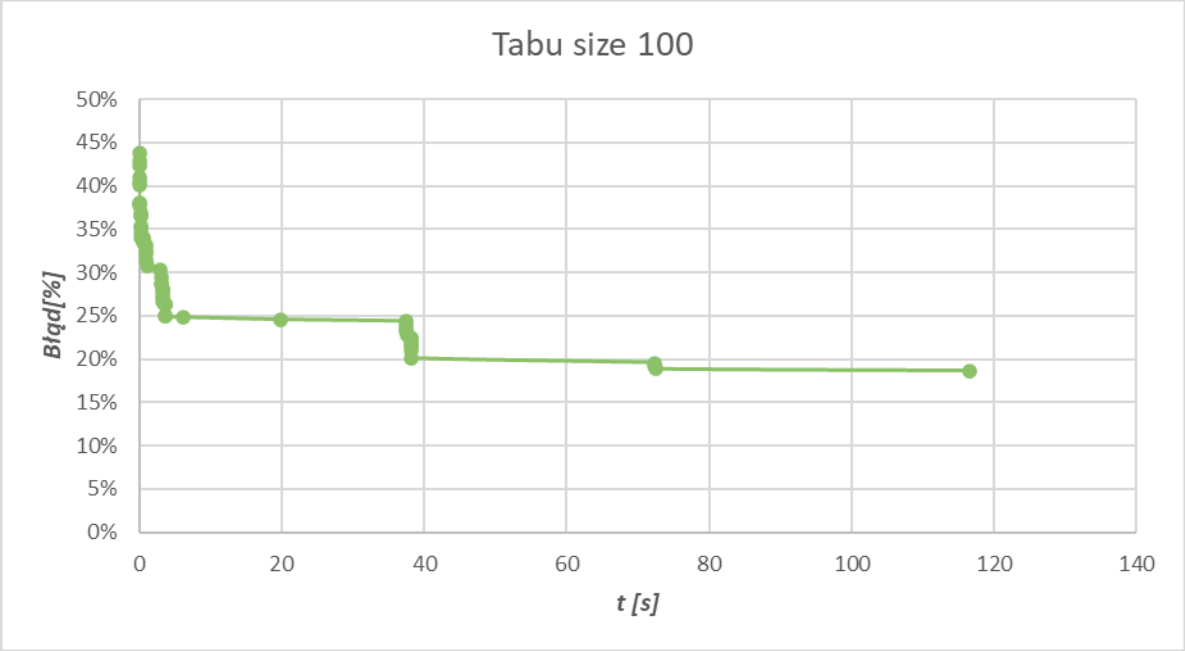
46	405%	13,413	314%	26,065	233%	65,123
47	393%	13,515	312%	26,223	232%	69,748
48	386%	14,931	311%	28,089	223%	69,967
49	382%	15,554	307%	28,245	223%	71,46
50	370%	15,69	305%	28,556	219%	71,891
51	368%	16,014	302%	28,715	198%	72,319
52	360%	16,162	299%	28,869	194%	75,253
53	358%	16,302	290%	29,024	191%	75,72
54	356%	16,513	283%	30,425	188%	78,507
55	348%	16,614	274%	33,329	185%	78,713
56	342%	16,721	269%	33,482	183%	79,364
57	333%	16,932	258%	33,946	181%	80,501
58	328%	17,345	255%	34,099	175%	80,716
59	327%	17,451	252%	36,246	171%	81,141
60	311%	17,866	248%	36,395	169%	81,353
61	305%	18,385	240%	37,502	164%	82,656
62	304%	18,604	226%	38,42	159%	84,13
63	302%	21,061	225%	39,637	151%	86,074
64	300%	21,391	218%	39,795	150%	87,515
65	295%	21,71	211%	42,393	146%	87,726
66	295%	21,822	203%	42,547	140%	88,551
67	283%	22,099	197%	45,42	131%	88,967
68	281%	22,152	196%	45,575	130%	89,577
69	272%	22,263	194%	47,363	128%	90,184
70	268%	23,39	193%	47,819	127%	91,004
71	265%	23,894	187%	47,97	123%	91,402
72	257%	25,718	175%	48,126	117%	93,826
73	251%	25,821	170%	50,04	108%	94,021
74	242%	25,924	161%	51,08	106%	94,423
75	236%	27,45	160%	52,271	105%	95,427
76	231%	27,859	154%	52,414	101%	95,829
77	222%	28,815	148%	52,57	100%	96,046
78	221%	29,142	145%	55,298	98%	96,255
79	214%	29,244	145%	55,593	98%	96,456
80	209%	30,605	137%	56,881	96%	97,227
81	206%	31,194	128%	57,602	93%	97,418
82	194%	31,694	125%	58,291	90%	97,606
83	192%	32,106	123%	58,568	88%	97,981
84	191%	33,017	122%	58,711	88%	98,168
85	190%	33,337	121%	58,852	85%	98,98
86	187%	34,042	119%	58,993	83%	99,536
87	183%	34,349	118%	59,556	82%	99,718
88	179%	34,456	116%	59,97	81%	99,892
89	178%	35,822	110%	60,109	81%	100,249
90	165%	36,123	108%	62,096	79%	101,121
91	162%	36,328	107%	62,37	79%	101,291
92	158%	37,039	105%	64,246	77%	101,801
93	154%	37,446	97%	64,386	76%	102,316
94	148%	38,128	94%	64,787	73%	103,712
95	144%	38,737	89%	65,968	70%	103,89

96	127%	40,688	87%	66,365	69%	104,243
97	125%	41,404	85%	66,884	67%	104,424
98	122%	41,996	79%	68,257	65%	104,961
99	117%	42,554	79%	68,381	63%	105,433
100	111%	43,845	78%	68,882	62%	105,596
101	109%	43,956	73%	69,764	62%	106,692
102	106%	44,247	73%	69,892	59%	108,271
103	105%	44,528	71%	70,559	59%	108,447
104	104%	44,998	70%	70,834	54%	109,08
105	103%	45,094	70%	70,967	54%	110,386
106	103%	45,275	67%	71,623	53%	110,965
107	102%	45,368	66%	71,967	53%	111,288
108	99%	45,674	65%	72,408	51%	112,282
109	98%	45,773	65%	72,619	50%	112,705
110	97%	45,87	65%	72,726	48%	112,966
111	96%	45,954	65%	72,839	47%	113,25
112	96%	46,037	65%	72,951	46%	113,398
113	91%	47,679	64%	73,294	45%	114,14
114	89%	47,951	60%	74,259	45%	114,865
115	87%	48,769	58%	75,28	43%	115,502
116	86%	48,861	55%	75,799	42%	115,85
117	85%	49,144	55%	76,406	42%	116,024
118	84%	49,238	54%	77,076	42%	117,301
119	84%	49,42	53%	77,558	42%	117,868
120	82%	49,689	52%	78,025	41%	118,593
121	81%	50,155	51%	78,401	41%	118,905
122	80%	50,321	51%	78,528		
123	80%	50,399	50%	78,643		
124	80%	50,485	50%	78,771		
125	80%	50,572	49%	79,023		
126	79%	50,668	47%	79,621		
127	77%	51,274	46%	80,069		
128	74%	51,501	45%	80,503		
129	74%	52,193	45%	80,839		
130	74%	52,28	44%	81,051		
131	73%	52,367	41%	81,163		
132	71%	52,786	41%	82,019		
133	71%	53,092	39%	82,79		
134	70%	53,258	38%	83,021		
135	66%	53,993	37%	83,244		
136	65%	54,21	37%	84,051		
137	65%	54,414	36%	84,27		
138	64%	54,696	35%	84,481		
139	63%	54,784	35%	84,781		
140	63%	54,86	34%	86,001		
141	63%	55,201	34%	87,215		
142	62%	55,274	33%	87,646		
143	62%	55,432	33%	87,944		
144	61%	55,583	32%	88,479		
145	60%	55,791	31%	89,255		

146	60%	55,856	31%	89,663		
147	60%	56,316	30%	90,072		
148	58%	57,625	30%	90,977		
149	56%	57,752	29%	91,181		
150	54%	58,579	29%	91,673		
151	53%	58,918	28%	92,247		
152	53%	59,009	27%	92,798		
153	53%	59,474	27%	93,082		
154	52%	61,214	27%	93,447		
155	51%	61,435	26%	93,78		
156	51%	61,67	25%	94,964		
157	51%	61,826	25%	96,084		
158	50%	67,55	24%	118,14		
159	50%	75,632				
160	48%	79,652				
161	48%	97,069				
162	48%	108,215				
163	47%	110,072				
164	47%	111,337				
165	47%	112,229				
166	46%	113,225				
167	46%	113,7				
168	46%	113,764				
169	45%	114,175				
170	45%	114,711				
171	45%	114,78				
172	45%	115,292				
173	44%	115,362				
174	43%	115,435				
175	41%	116,063				
176	41%	116,137				
177	41%	117,691				
178	40%	118,036				
179	40%	118,694				



Wykres 5. Wykres błędu względnego dla 3 różnych populacji



Wykres 6. Wykres błędu względnego dla najlepszego rozwiązania przez algorytm Tabu Search

Wnioski

Wyniki dla algorytmu genetycznego różnią się przebiegiem od wyników dla algorytmu Tabu Search, mimo podobnych wyników końcowych. Algorytm ten zaczyna przeglądać rozwiązania od dużo gorszych rozwiązań startowych niż implementowany wcześniej algorytm Tabu Search. Dla każdego pliku inaczej dobierano wielkość populacji.

Dla pliku fftv33 najlepsze znalezione rozwiązanie to rozwiązanie optymalne, dla populacji wielkości 220. Reszty populacji rozwiązania są dużo gorsze, od 9% do 14%. Najlepsze rozwiązanie zostało znalezione po 66,524 s. W przypadku Tabu Search również znaleziono rozwiązanie optymalne, jednak w czasie 10,783 s, czyli około 6 razy szybciej. Warto również dodać, że algorytm Tabu Search znalazł optymalne rozwiązanie dla kilku różnych rozmiarów listy tabu, natomiast algorytm genetyczny tylko dla jednej populacji.

Dla pliku fftv55 najlepsze znalezione rozwiązanie przez algorytm genetyczny to rozwiązanie rzędu 3% błędu względnego, otrzymane po 48,339 s, dla populacji wielkości 640. Dla innych wielkości znalezione rozwiązania wyniosły 13% błędu względnego. W przypadku algorytmu Tabu Search najlepsze znalezione rozwiązanie wynosiło 1% błędu względnego, znalezione po 53,679 s.

Dla pliku fftv170 najlepsze znalezione rozwiązanie przez algorytm genetyczny to rozwiązanie rzędu 24% błędu względnego, otrzymane po 118,14 s, dla populacji wielkości 6000. Reszta znalezionych waha się między 40 a 41 % błędu względnego. Algorytm Tabu Search znalazł rozwiązanie wynoszące 19% błędu względnego po 116,598 s.

Dla plików fftv55 oraz fftv170 algorytm genetyczny znajduje gorsze rozwiązania niż algorytm Tabu Search. Warto również zauważyć, że w przypadku algorytmu genetycznego trzeba dokładniej dobrać parametr wielkości populacji niż parametr wielkości listy tabu dla Tabu Search, żeby otrzymać chociaż podobne wyniki co Tabu Search.

Być może udałoby się osiągnąć lepsze wyniki dla algorytmu genetycznego, gdyby inaczej dobrać współczynniki mutacji i krzyżowania. Jednak dla każdego testowanego pliku wynosiły 0.01 dla współczynnika mutacji oraz 0.8 dla współczynnika krzyżowania.