Uszeregowanie *n* zadań (minimum 100) na dwóch maszynach (Flowshop), z przerwami konserwującymi na pierwszej i drugiej maszynie mającymi ustalony wcześniej czas trwania oraz czas rozpoczęcia. Liczba przerw w instancji - zmienna. Operacje niewznawialne. Minimalizować sumę czasów zakończenia wszystkich operacji.

ALGORYTM GENETYCZNY DLA PROBLEMU SZEREGOWANIA ZADAŃ

Patryk Gliszczyński [a], Krzysztof Prajs [b]

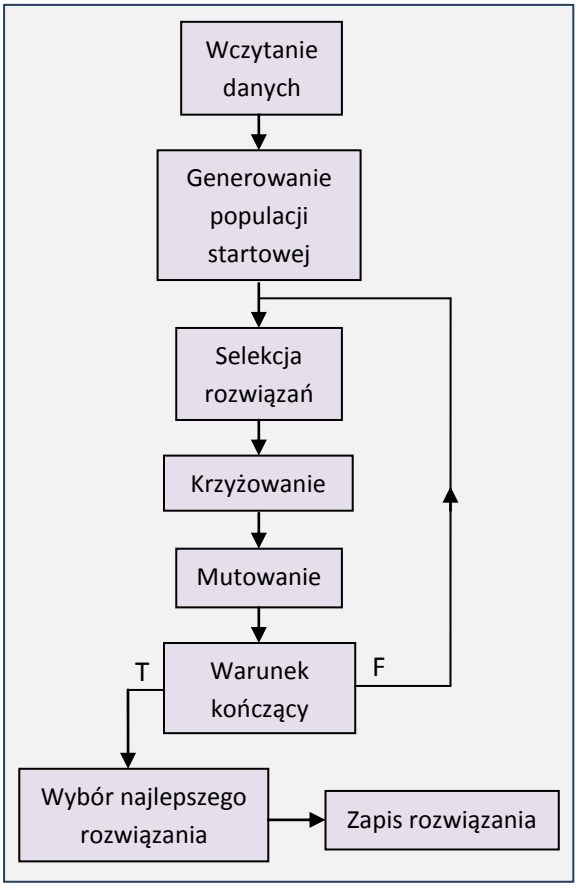
[a] nr Indeksu: 117288 | I5 | środa 13 ^{30} – 15 ^{00} | email: patryk.gliszczynski94@gmail.com

[b] nr Indeksu: 117298 | I5 | środa 13 ^{30} – 15 ^{00} | email: kryszak78@o2.pl

1. Wstęp teoretyczny

Algorytm genetyczny jest pewnym rodzajem algorytmu przeszukującego przestrzeń alternatywnych rozwiązań problemu w celu wyszukania rozwiązania optymalnie najlepszego. Naszym zadaniem było przygotować ten algorytm do rozwiązania problemu szeregowania zadań, czyli jednego z trudniejszych i zarazem najważniejszych aspektów współczesnej informatyki [1]. Poprzez szeregowanie zadań rozumiemy: takie ułożenie poszczególnych zadań aby wspólnie tworzyły pełną całość spełniając jednocześnie podstawowe założenia jakimi są: brak możliwości wznawiania zadań, niemożliwość wykonywania zadania gdy na danej maszynie nastąpił czas na przerwę konserwującą, oraz wykonanie drugiego zadania tylko po zakończeniu pierwszego (Flowshop). Problem ten przypomina nic innego jak wkładanie klocków o różnych wielkościach do wolnych dziur. Należy w taki sposób ułożyć te zadania, aby zminimalizować czas wykonania całego uszeregowania, oraz przerw typu idle (takich, podczas których nic się nie dzieje). Naszym zadaniem było minimalizować sumę czasów zakończenia wszystkich operacji. Algorytm genetyczny posiada zestaw operacji które należy wykonywać w odpowiedniej kolejności, aby uzyskać poprawny końcowy efekt.

- **Wczytanie danych** – pobranie danych na temat aktualnie rozwiązywanego problemu, w naszym przypadku wczytanie tych danych z pliku tekstowego.
- **Generowanie populacji startowej** – wylosowanie *n* uszeregowań spełniających kryteria problemu szeregowania zadań i utworzenie z niej nowej populacji.
- **Selekcja rozwiązań** – wybranie spośród wszystkich rozwiązań z populacji *m* rozwiązań, z których zostanie utworzona nowa populacja.
- **Krzyżowanie** – polega na utworzeniu nowego genotypu (dziecka), z dwóch, lub więcej innych genotypów (rodziców).
- **Mutowanie** – delikatna zmiana genotypu dla danego uszeregowania.
- **Warunek kończący** – sprawdzenie, czy znaleziono optymalnie dobre rozwiązanie, bądź warunek czasowy (zakończenie po przekroczeniu czasu).
- **Wybór najlepszego rozwiązania** – wybranie z populacji rozwiązań, najlepszego uszeregowania.



- **Zapis rozwiązania** – zapisanie do pliku tekstowego, wybranego uprzednio najlepszego chromosomu, zgodnie z wcześniej określoną funkcją celu.

Należałoby również dokładniej opisać główne operatory dla algorytmu genetycznego, jakimi są krzyżowanie, mutacja, oraz selekcja:

- **Krzyżowanie** - w tej operacji wszystkie rozwiązania z danej populacji są losowo grupowane w podzbiory dwóch chromosomów [2]. Po połączeniu parami rozwiązań następuje „wymiana informacji" pomiędzy tymi dwoma uszeregowaniami, z którego powstaje jedno lub dwa nowe uszeregowania (w naszym programie tworzymy pojedyncze rozwiązanie). Możemy wyróżnić wiele sposobów wykonania operacji krzyżowania.

Najprostszym jest tzw. **krzyżowanie w jednym punkcie**, które polega na „przecięciu" rozwiązań na dwie części w losowym punkcie, a następnie połączenie ich w celu utworzenia nowego (dziecka). Sposób ten może być jednak niekorzystny dla problemu szeregowania zadań ze względu na różne przeszkody wynikające z idei szeregowania zadań (przerwy na maszynach, brak integralności zgodnie z systemem Flowshop, itp).

Rodzic I	Zadanie	5	2	3	6	7	4	1
	Maszyna	1	4	2	3	2	1	3
Rodzic II	Zadanie	3	2	5	7	4	6	1
	Maszyna	3	2	4	1	4	2	3
Dziecko I	Zadanie	5	2	3	7	4	6	1
	Maszyna	1	4	2	1	4	2	3

Drugim sposobem jest tzw. **krzyżowanie w dwóch punktach**. Polega ono na wylosowaniu przedziału zadań w dwóch rozwiązaniach, które mogą zostać połączone w jedno, lub dwa nowe rozwiązania. Ten sposób wydaje się być najsensowniejszy do zastosowania dla problemu szeregowania zadań, ponieważ możliwe jest ewentualne, manipulowanie rozmiarem przedziału, w celu uniknięcia powyżej wymienionych problemów. Krzyżowanie jest przeprowadzone z prawdopodobieństwem 0.7***** [3].

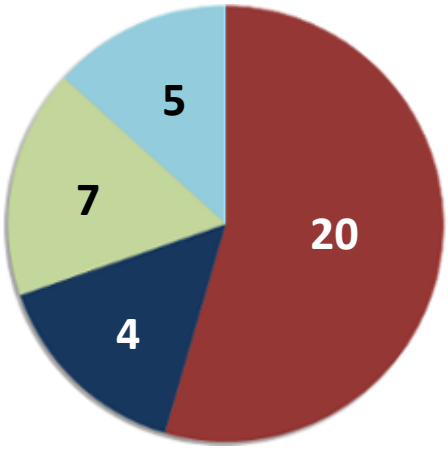
Rodzic I	Zadanie	7	2	6	3	4	5	1
	Maszyna	3	4	2	4	2	1	3
Rodzic II	Zadanie	5	1	3	4	6	2	7
	Maszyna	1	2	4	1	3	2	2
Dziecko I	Zadanie	7	2	3	4	6	5	1
	Maszyna	3	4	4	1	3	1	3

- **Mutacja** – polega na losowym wybraniu dwóch zadań spośród chromosomu a następnie zamianie ich miejscami. Rodzaj tej mutacji, możemy określić mianem mutacji wymiany bitu, lub *ang. Swap Mutation*. Jest ona używana w celu zachowania różnorodności populacji. Możemy również wyróżnić inne rodzaje mutacji: wstawienie bitu, usunięcie genu, mutacja infekcją wirusową, oraz mutacja z różnicowaniem. Mutacja jest przeprowadzana z prawdopodobieństwem 0.5 *[3] .

Przed Mutacją	Zadanie	7	2	6	3	4	5	1
	Maszyna	3	4	2	4	2	1	3
Po Mutacji	Zadanie	4	2	6	3	7	5	1
	Maszyna	2	4	2	4	3	1	3

*****Propozycja autorów pracy naukowej [3], wartość ta została uwzględniona również w testach.

- **Selekcja** - jest procesem podczas którego wybierana jest nowa populacja chromosomów. Dzięki temu nawiązując do teorii Darwina najlepsze egzemplarze powinny przetrwać i mieć szansę "rodzić" nowych potomków. Pierwszym krokiem jest wyznaczenie funkcji celu dla każdego chromosomu w populacji. Każdy osobnik w puli chromosomów posiada pewną wartość funkcji celu. W naszym zadaniu należało minimalizować sumę czasów zakończenia wszystkich operacji. Wartość ta jest używana w późniejszym etapie selekcji. Istnieje bardzo wiele metod selekcji.



Jedną z najpopularniejszych metod, jest **metoda ruletki**. W pierwszej kolejności budujemy wirtualne koło, którego wycinki odpowiadają poszczególnym osobnikom. Im lepszy osobnik, tym większy wycinek koła zajmuje. Rozmiar wycinków zależy od wartości funkcji celu. Poprzez to prawdopodobieństwo, że lepszy osobnik zostanie wybrany jako rodzic, jest większe. Niestety ewolucja przy takim algorytmie z każdym krokiem zwalnia. Jeżeli osobniki są podobne, to każdy dostaje równy wycinek koła fortuny i presja selekcyjna spada. Algorytm słabej rozróżnia osobniki dobre od słabszych. Przykładowo, korzystając z rysunku obok, mamy 4 różne genotypy, różniące się wartością funkcji celu. Najlepszym z nich jak widać jest genotyp posiadający, aż 20 pul w naszym "kole ruletki", najgorszą wartość funkcji celu posiada natomiast genotyp granatowy (4 pola). Następnie losowana jest wartość z przedziału od 1 do SUM(x_i), gdzie x jest wartością funkcji celu, dla danego genotypu. Rozpoczynając od pierwszego w populacji, od wylosowanej liczby odejmowana jest wartość funkcji celu, i dla rozwiązania gdy spadnie ona poniżej zera jest to nasz wyselekcjonowany genotyp, dołączany do nowej populacji.

Drugą, równie ciekawą metodą jest **selekcja turniejowa**. Polega ona na tym, że spośród populacji rozwiązań, losowanych jest N dowolnych genotypów, które następnie są ze sobą porównywane na podstawie wartości funkcji celu. Osobnik z większą wartością przechodzi do nowej populacji, proces ten jest powtarzany, aż cała nowa populacja zostanie utworzona. Selekcja metodą turniejową, jest pozbawiona niedogodności metody koła ruletki, gdzie wymagana jest maksymalizacja funkcji oceny, w turnieju ważna jest jedynie informacja o "lepszości" jednego rozwiązania nad innym. Metoda ta jest o tyle ciekawa, że na pewno zawsze zostanie wybrany genotyp z najwyższą wartością funkcji celu, a zarazem najgorsze zostaną odrzucone. W przypadku metody koła ruletki, mogło się tak stać, że najlepsze rozwiązania nie zostały wybrane. Metoda ta może jednak okazać się nieskuteczna, gdy wpadniemy w pewne minimum lokalne, i nie będziemy w stanie poprawić wyniku naszego uszeregowania. Problemy tego typu można rozwiązywać, przechowując w pamięci gorsze osobniki z poprzednich selekcji, i ewentualnie, gdy przez wystarczająco długi czas nasz wynik nie polepsza się, dodać je do nowej populacji.

	x_1	x_2
1	256	102
2	321	493
3	208	124
4	515	692
5	332	98
6	174	821
7	392	713
8	82	555
9	284	201
10	332	291
11	994	672

2. Implementacja operatorów w programie

- **KRZYŻOWANIE**

  Pierwszym operatorem, który wydaje się być najistotniejszym w algorytmie genetycznym jest operator **krzyżowania**. W naszym programie wykorzystaliśmy metodę **krzyżowania w dwóch punktach**, ponieważ uznaliśmy, iż będzie ona zdecydowanie najefektywniejsza przy problemie szeregowania zadań, ze względu na mniejszy losowo wybierany przedział - możliwe wystąpienie błędów uszeregowania zostaje zmniejszone. W naszej metodzie założyliśmy, że przedział: nigdy nie będzie się zaczynał od pierwszego zadania w uszeregowaniu, nie będzie się kończył na ostatnim zadaniu, oraz rozmiar przedziału będzie większy od 1, aby uniknąć wystąpienia krzyżowania w jednym punkcie. Poniżej zamieszczam naszą funkcję użytą przy wyznaczaniu przedziału.

```
1    do {
2        rand_start_range = rand() % (numberOfTasks - 2) + 1;
3        rand_over_range = rand() % (numberOfTasks - 2) + 1;
4    } while(rand_start_range == rand_over_range
5            || rand_start_range == rand_over_range + 1
6            || rand_start_range == rand_over_range - 1);
7
8    if(rand_start_range > rand_over_range) {
9        int temp = rand_start_range;
10       rand_start_range = rand_over_range;
11      rand_over_range = temp;
12   }
```

Jeżeli wyznaczyliśmy już od którego zadania nasz przedział będzie się rozpoczynał i na którym kończył kolejnym krokiem jest przejście do zadania z punktu początkowego i końcowego przedziału, oraz dodatkowo poprzedzającego początek zakresu, jak i następującego po końcu zakresu. Indeksy 1 i 2 przy nazwach zmiennych odpowiadają numerom rozwiązań (rodzicom), użytych w krzyżowaniu.

```
1    for(int i = 0 ; i < rand_start_range ; i++){
2       start_task1 = start_task1->next;
3       start_task2 = start_task2->next;
4   }
5
6   for(int i = 0 ; i < rand_over_range ; i++){
7       over_task1 = over_task1->next;
8       over_task2 = over_task2->next;
9   }
10
11   while(before_start_task1->next != start_task1)
12     before_start_task1 = before_start_task1->next;
13
14   while(before_start_task2->next != start_task2)
15     before_start_task2 = before_start_task2->next;
16
17   after_over_task1 = over_task1->next;
18   after_over_task2 = over_task2->next;
```

Następnym etapem jest zamiana czasów rozpoczęcia zadań dla dwóch rozwiązań oraz podmiana wskaźników w taki sposób, aby z dwóch rodziców, powstała dwójka dzieci zgodnie z zasadami krzyżowania w dwóch punktach.

```
 1   task_start_time = start_task1->start_time;
 2   start_task1->start_time = start_task2->start_time;
 3   start_task2->start_time = task_start_time;
 4
 5   over_task1->next = after_over_task2;
 6   over_task2->next = after_over_task1;
 7   before_start_task1->next = start_task2;
 8   before_start_task2->next = start_task1;
 9
10 //--AKTUALIZACJA-CZASOW-ROZPOCZECIA
11    before_start_task1 = start_task1;
12    while(before_start_task1 != over_task1) {
13      before_start_task1->next->start_time
14      = before_start_task1->start_time
15      + before_start_task1-  >time_length;
16
17      before_start_task1 = before_start_task1->next;
18    }
19
20   before_start_task2=start_task2;
21    while(before_start_task2 != over_task2) {
22      before_start_task2->next->start_time
23      = before_start_task2->start_time
24      + before_start_task2->time_length;
25
26      before_start_task2 = before_start_task2->next;
27    }
```

Powyższe operacje należy jeszcze raz powtórzyć dla maszyny drugiej (nie uwzględniając losowania nowego przedziału). Kiedy mamy przestawione obydwa rozwiązania, jedyne co nam pozostało, to sprawdzenie czy nasze uszeregowania są w pełni poprawne (zgodne z zasadami problemu szeregowania zadań), oraz ewentualne poprawienie niepasujących elementów (powstałe duplikaty zadań, zadania nachodzące na siebie, zadania nachodzące na przerwy, oraz wykonanie drugiej operacji przed skończeniem pierwszej).

```
1   repair(new_solution1, numberOfTasks);
2   repair(new_solution2, numberOfTasks);
```

Operacja krzyżowania przechodzi tylko i wyłącznie kilkukrotnie przez całe rozwiązanie, można zatem stwierdzić, że zależy tylko i wyłącznie od liczby zadań w uszeregowaniu. Przez co złożoność obliczeniowa tej operacji wynosi O(*n*), gdzie *n* to liczba zadań w uszeregowaniu.

- **MUTACJA**

Drugim operatorem stosowanym w algorytmie genetycznym jest **mutacja.** W naszym programie zastosowaliśmy tzw. mutację wymiany bitu (*ang. bit swap mutation*). Polega ona na tym, że wybierane są dwa losowe zadania spośród uszeregowania, a następnie zamieniane są one miejscami. Ogólna zasada jest bardzo prosta - w pierwszej kolejności losujemy różne numery dwóch zadań i przesuwamy wskaźniki, tak aby wskazywały na to zadanie.

```
1   int z1_rand = rand() % (numberOfTasks) ;
2   int z2_rand = rand() % (numberOfTasks);
3
4   while (z1_rand == z2_rand)
5      z2_rand = rand() % (numberOfTasks);
6
7   for(int i = 0 ; i < z1_rand ; i++) z1_op1 = z1_op1->next;
8   for(int i = 0 ; i < z2_rand ; i++) z2_op1 = z2_op1->next;
9
10  while(z1_op2->real_number != z1_op1->real_number)
11     z1_op2 = z1_op2->next;
12  while(z2_op2->real_number != z2_op1->real_number)
13     z2_op2 = z2_op2->next;
```

Następnie zadania przesyłane są do funkcji, która jest odpowiedzialna za zamianę ich w uszeregowaniu. Funkcji tej nie będę przedstawiał, ponieważ jest bardzo długa ze względu na wiele przypadków występujących przy zamianie pozycji zadań w uszeregowaniu, które należy uwzględnić (zamiana pierwszego z ostatnim, zamiana gdy zadania są obok siebie, itp.).

```
1   swapTasks(new_solution->machine_1_sequence, z1_op1, z2_op1);
2   swapTasks(new_solution->machine_2_sequence, z1_op2, z2_op2);
```

Ostatnim etapem, takim samym jak w krzyżowaniu jest naprawa ewentualnie powstałych błędów uszeregowania.

```
1   repair(new_solution, numberOfTasks);
```

Ponieważ operacja mutacji jest zdecydowanie prostszą niż krzyżowanie, to jest wykonywana w zdecydowanie krótszym czasie. W tym przypadku nie zależy w pełni od liczby zadań, lecz od numerów zadań które wylosowano, z przedziału od <0; *n*>, gdzie *n* to liczba zadań. Funkcja złożoności obliczeniowej wygląda zatem następująco O(*p*+*q*), gdzie *p* to pierwsza wylosowana liczba, a *q* to druga wylosowana liczba z przedziału.

- **SELEKCJA**

  Ostatnim operatorem, który jest kluczowym składnikiem algorytmu genetycznego jest **selekcja**. Metodą zastosowaną w naszym programie jest selekcja turniejowa. W pierwszej kolejności natomiast należy wyznaczyć wartość funkcji celu. Funkcja ją wyznaczająca na samym początku szuka największej wartości, którą minimalizujemy (w naszym przypadku jest to suma czasów zakończenia wszystkich operacji) i dodaje do niej wartość 1. Jeżeli wartość ta zostanie znaleziona następnym krokiem jest dla każdego rozwiązania w populacji odjęcie największej wartości od sumy czasów zakończenia wszystkich operacji. W ten sposób otrzymamy nową wartość funkcji celu, która dla najgorszego uszeregowania będzie równa 1. Im większa będzie ta liczba, tym lepsze uszeregowanie. Idea jest dość intuicyjna, więc nie będę zamieszczać tutaj implementacji tej operacji znajdującej się w naszym programie.

  Jeżeli mamy wyznaczoną tą wartość kolejnym krokiem jest wyszukanie dwóch losowo wybranych rozwiązań z populacji. Wybieranie sprawdza, czy wylosowane rozwiązanie nie zostało już wcześniej wybrane jako najlepsze rozwiązanie i dodane do nowej populacji.

```
1   while(true) {
2      solution1_number = rand() % (population_size - 1);
3      if(popArray[solution1_number] == 0) {
4           popArray[solution1_number] = 1;
5           break;
6      }
7   }
8
9   while(true) {
10     solution2_number = rand() % (population_size - 1);
11     if(popArray[solution2_number] == 0) {
12          popArray[solution2_number] = 1;
13          break;
14     }
15  }
```

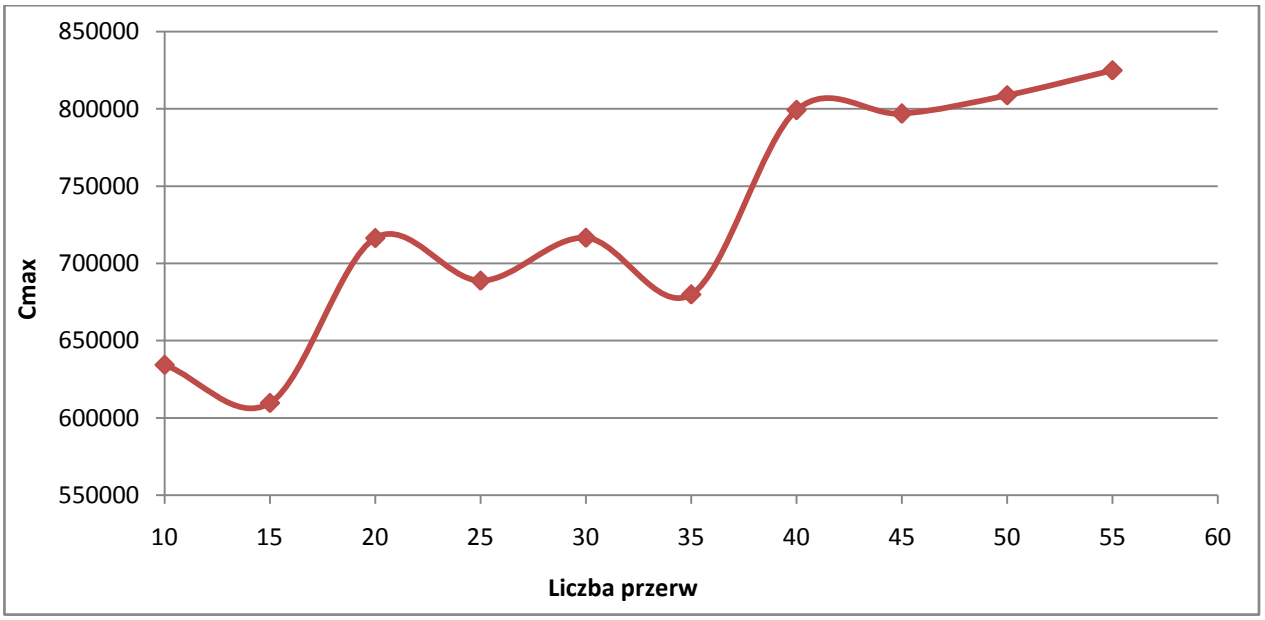
  Kiedy mamy już wybrane dwa rozwiązania, możemy je porównać między sobą i wybrać to które jest lepsze, pod względem wartości funkcji celu. Lepsze rozwiązanie jest dodawane do nowej populacji.

```
1   for(int i = 0 ; i < solution1_number ; i++)
2      solution1 = solution1->next;
3
4   for(int i = 0 ; i < solution2_number ; i++)
5      solution2 = solution2->next;
6
7   if(solution1->fitness_value > solution2->fitness_value) {
8       popArray[solution2_number] = 0;
9       new_population->add_solution(solution1);
10  } else {
11      popArray[solution1_number] = 0;
12      new_population->add_solution(solution2);
13  }
```

3. Testy

TEST 1 - LICZBA PRZERW

Pierwszym testem jaki postanowiliśmy przeprowadzić, było sprawdzenie w jaki sposób nasz program zareaguje na **zmianę liczby przerw** dla danej instancji problemu. **Liczba przerw** była jednym z parametrów generatora w naszym zadaniu. Była to wartość znana z góry równa (*k*n)*, gdzie *k* jest danym parametrem, a *n* jest liczbą zadań w danym problemie. W naszych testach liczba zadań była stała i wynosiła ona 100, natomiast wartość parametru *k* zmieniała się od 0.1 do 0.55, ze stopniem wzrostu równym 0.05.



Wykres 1. Zależność zmiany wartości funkcji celu, od liczby przerw w uszeregowaniu.

	Liczba przerw									
Pomiar	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**	**55**
1	640887	609943	717922	687447	719822	685445	801690	795000	810656	832434
2	635768	605065	716927	688304	708884	673225	794754	789858	816002	823945
3	627238	616063	710390	682819	718927	674260	801813	798404	806899	818742
4	643844	606567	727165	676523	712489	685230	790786	804252	813598	822351
5	622868	610810	714593	682895	723808	682314	792317	798275	816261	822915
6	639472	601823	720407	692978	723760	688615	802502	790073	813946	828172
7	640459	606623	714726	694944	721526	674220	805283	798406	801231	833528
8	634436	616859	709376	694413	710965	678765	795238	796460	801328	826378
9	626019	612165	713226	697930	722104	686363	805387	799097	805286	817670
10	632649	611275	719590	689577	703888	671517	801133	798408	801929	822537
ŚREDNIA	634364	609719	716432	688783	716617	679995	799090	796823	808714	824867

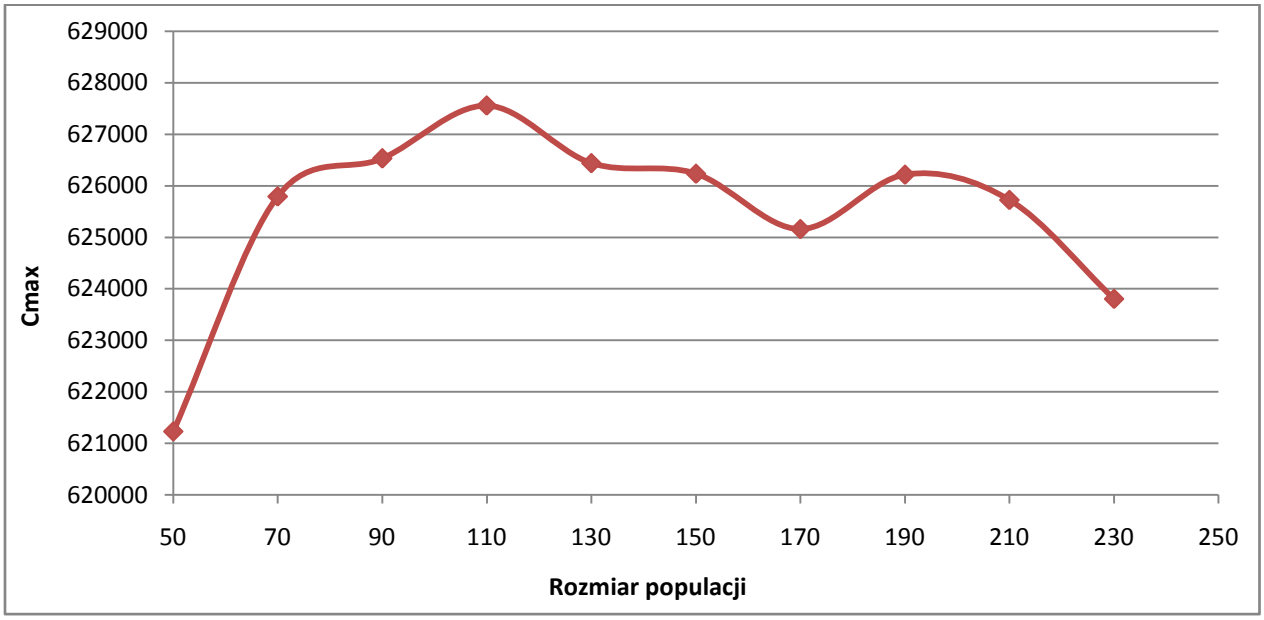
Tabela 1. Wyznaczone wartości funkcji. Dla każdego punktu pomiarowego po 10 pomiarów.

Do każdego punktu pomiarowego zostało wykonanych 10 pomiarów, z których wyliczona została uśredniona wartość funkcji celu. Jak widać na *wykresie 1* wartości te delikatnie skaczą w niektórych punktach. Może być to spowodowane "chwilowym" lepszym ułożeniem przerw, przez co łatwiej było umieścić zadania między

nimi. Widać natomiast, że wartość ta ma tendencję wzrostową, względem liczby przerw, co jest dość logicznym i intuicyjnym wnioskiem z tego względu, że trudniej jest umieścić losowo zadania między tymi przerwami tak, aby tworzyły pełne poprawne uszeregowanie.

TEST 2 - ROZMIAR GŁÓWNEJ POPULACJI

Tym razem badaliśmy w jak dużym stopniu **rozmiar populacji** ma wpływ na wynik końcowy algorytmu genetycznego. Rozmiar populacji jest jednym z parametrów, którego przestrzega operator selekcji (dobiera on tyle nowych rozwiązań ile wymaga rozmiar populacji).



Wykres 2. Zależność zmiany wartości funkcji celu, od rozmiaru populacji.

	Rozmiar populacji									
Pomiar	**50**	**70**	**90**	**110**	**130**	**150**	**170**	**190**	**210**	**230**
1	621354	632562	632285	634122	627192	620327	628205	624706	624556	623341
2	629366	621214	624248	629521	623963	622717	630292	619964	623832	617809
3	615711	631728	629002	630863	616922	631732	628962	624282	633735	624302
4	623891	631642	616262	625717	617808	630548	615868	623081	626579	627632
5	622739	634304	635272	625983	632297	627403	618672	622170	625634	630163
6	619424	630340	625876	629849	633605	625275	626486	625068	627239	626006
7	619114	625956	623165	632050	632317	628987	625869	633380	625397	623214
8	610639	609495	623829	616421	627008	627858	628838	633129	624738	618381
9	625428	611923	633897	626528	620959	614124	622658	628722	622099	623597
10	624672	628752	621455	624508	632331	633380	625755	627666	623449	623583
ŚREDNIA	621234	625792	626530	627556	626440	626235	625161	626217	625726	623803

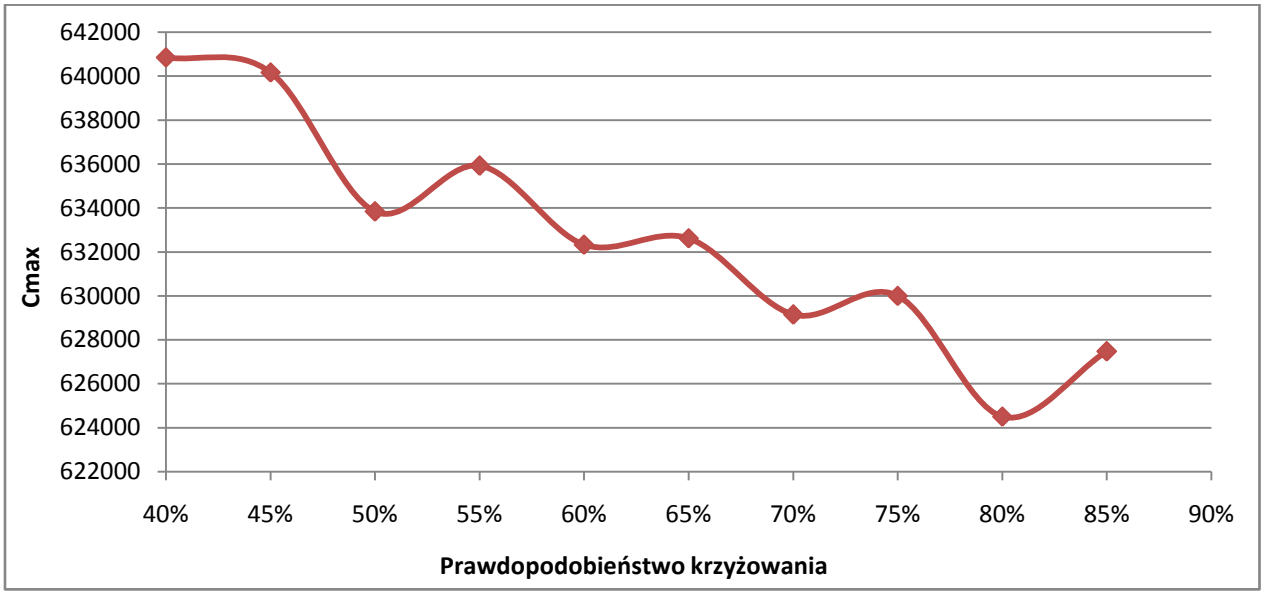
Tabela 2. Wyznaczone wartości funkcji. Dla każdego punktu pomiarowego po 10 pomiarów.

W ogólności, z danych które można odczytać z *wykresu 2* stwierdzić można, że rozmiar populacji nie wpływa znacząco na jakość uszeregowania. Co jest dość dziwnym zjawiskiem, ponieważ przed przeprowadzeniem tego eksperymentu spodziewaliśmy się, że rozmiar populacji będzie bardzo istotny dla tworzenia nowych

potomków, a przynajmniej dla tych populacji mniejszych rozmiarem, będzie znacząca poprawa. Możliwe jest również , że lepiej byłoby to widać, gdybyśmy przeprowadzili ten eksperyment dla dużo większych populacji (nawet kilku-tysięcznych). Wykres sprawia wrażenie, że wartość funkcji celu wpierw rośnie, po czym maleje. Należy natomiast zwrócić uwagę, na to że wychylenie to jest niewielkie, ponieważ nie sięga ono więcej niż 700 jednostek czasu. Można zatem uznać wykres za prawie stały, ponieważ tak lekka zmiana wartości funkcji celu nie ma większego znaczenia i może wynikać np. z nieco gorszego wygenerowania populacji startowej.

TEST 3 - PRAWDOPODOBIEŃSTWO KRZYŻOWANIA

W tym teście badane było zachowanie algorytmu genetycznego na **zmianę prawdopodobieństwa krzyżowania**. Dla naszych testów za wartość minimalną przyjęliśmy 40% i zwiększaliśmy ją stopniowo co 5 punktów procentowych, aż do wartości 85%.



Wykres 3. Zależność zmiany wartości funkcji celu, od prawdopodobieństwa krzyżowania.

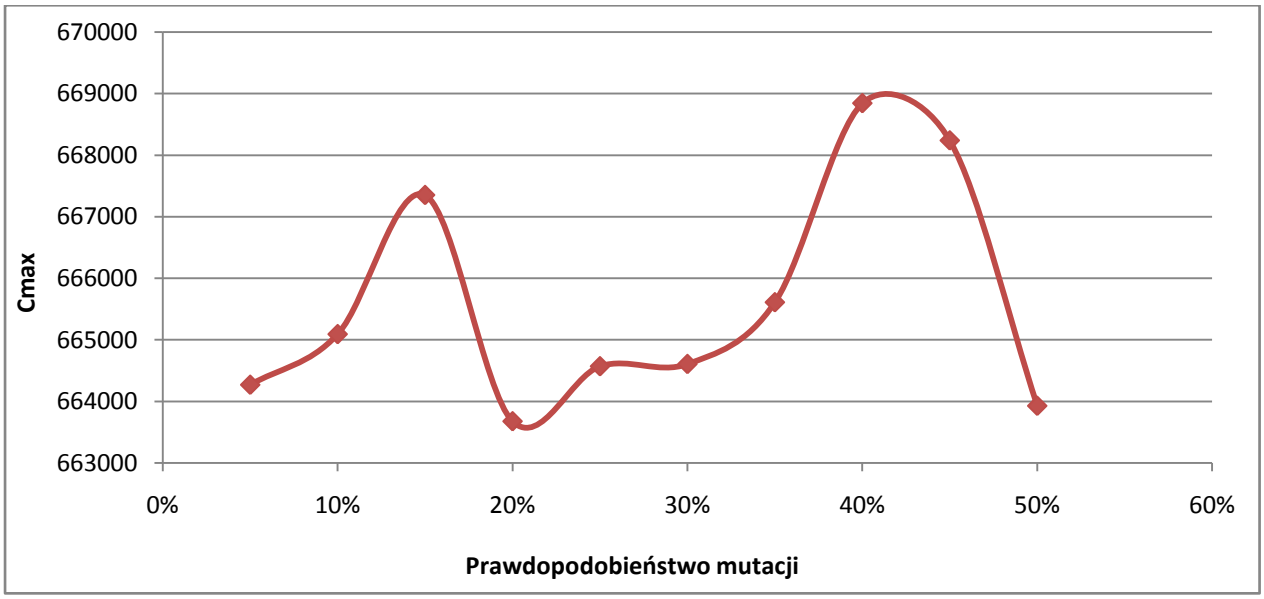
	Prawdopodobieństwo krzyżowania									
Pomiar	**40%**	**45%**	**50%**	**55%**	**60%**	**65%**	**70%**	**75%**	**80%**	**85%**
1	630455	637322	631869	634035	634489	626834	625344	627135	628712	627937
2	643931	650029	629024	634422	626231	633068	625772	632747	623908	627690
3	641962	645436	644636	625426	635793	634057	627880	627928	633932	626138
4	643383	633155	634672	635037	626683	633805	617315	629936	613385	626839
5	639965	630727	634253	632369	627742	631559	625906	631816	624286	630799
6	644847	641511	623677	641860	641957	630120	636257	627501	624390	625306
7	643729	639206	626198	638995	631874	635994	633584	633613	627249	629066
8	641565	640290	640565	640239	634731	636213	631896	632799	624964	626222
9	631053	642683	635717	638469	632946	636250	624369	628816	615839	627187
10	647564	641277	637729	638316	630881	628272	643164	627716	628377	627512
ŚREDNIA	640845	640164	633834	635917	632332,7	632617	629149	630001	624504	627470

Tabela 3. Wyznaczone wartości funkcji. Dla każdego punktu pomiarowego po 10 pomiarów.

Odczytując dane z *tabeli 3*, *oraz wykresu 3* widać, że zmiana wartości prawdopodobieństwa krzyżowania miała istotny wpływ na wynik końcowy (jakość uszeregowania). Wraz ze wzrostem tego parametru, malała suma zakończenia czasów wszystkich operacji. Spowodowane może to być tym, że operacja krzyżowania w znaczącym stopniu zmienia uszeregowanie, dzięki czemu, więcej osobników miało szansę być "poddanym" tej operacji (a co za tym stoi, zmienić swoje uszeregowanie, gdzie następnie operator selekcji dbał o to aby wybrać najlepsze rozwiązania).

TEST 4 - PRAWDOPODOBIEŃSTWO MUTACJI

Kolejnym eksperymentem jako postanowiliśmy przeprowadzić jest reakcja algorytmu na zmianę wartości **prawdopodobieństwa mutacji**. Postanowiliśmy rozpocząć badania dla 5% i zwiększać ją co 5 punktów procentowych, póki osiągnie wartość 50%.



Wykres 4. Zależność zmiany wartości funkcji celu, od prawdopodobieństwa mutacji.

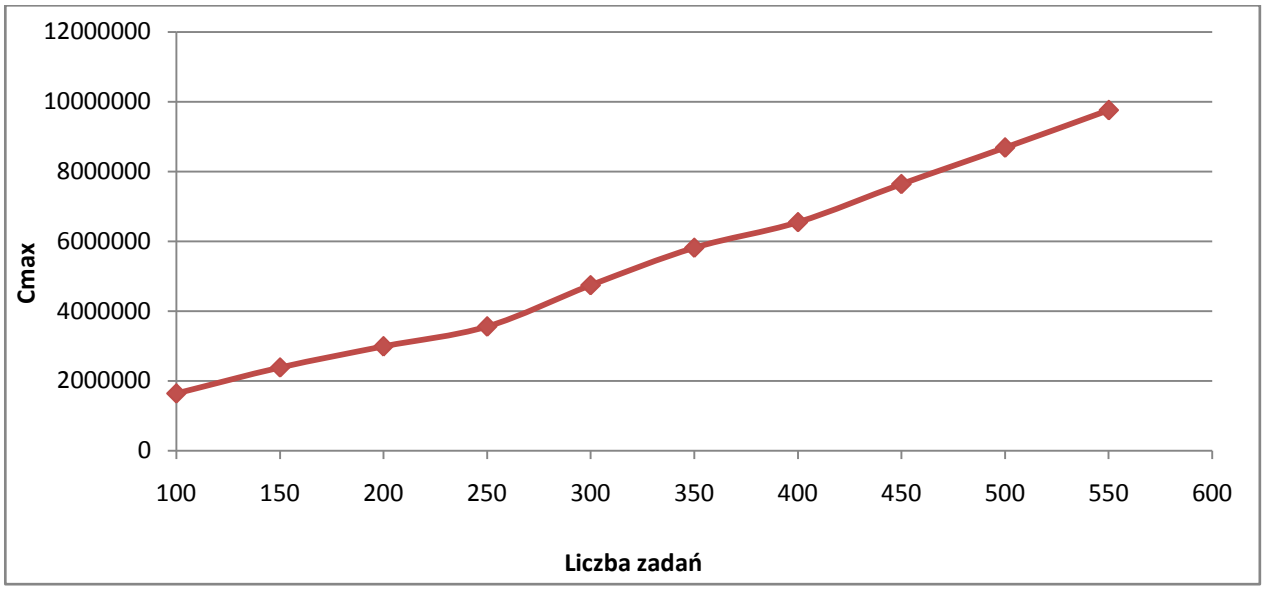
	Prawdopodobieństwo mutacji									
pomiar	**5%**	**10%**	**15%**	**20%**	**25%**	**30%**	**35%**	**40%**	**45%**	**50%**
1	667023	668726	667123	665410	659296	664686	664098	670295	668971	668090
2	663299	668796	671394	670431	664303	665589	667909	660799	666612	666474
3	666297	666498	661208	661883	666810	655864	668814	671851	667540	660701
4	665514	662287	668155	673320	658160	662103	662883	671231	663394	669847
5	667873	667620	665275	663978	653323	661370	653842	667670	673482	656861
6	656682	661376	674969	665574	666184	667566	663576	669640	670172	661986
7	664399	660311	662670	654917	664009	666738	664207	666269	678432	671249
8	669268	664935	664847	664265	671957	665165	669582	668501	658280	659667
9	654336	661218	666707	661201	672302	667272	671615	670339	667935	662621
10	667983	669137	671166	655765	669338	669698	669558	671840	667564	661760
ŚREDNIA	664267	665090	667351	663674	664568	664605	665608	668844	668238	663926

Tabela 4. Wyznaczone wartości funkcji. Dla każdego punktu pomiarowego po 10 pomiarów.

Przeprowadzony eksperyment, dał nieco dziwne wyniki. Ponieważ odczytując z *wykresu 4* sumaryczny czas zakończenia wszystkich operacji rośnie, dla pewnych wartości parametru prawdopodobieństwa mutacji, a dla innych maleje. Można zatem wywnioskować, że każda ustawiona przez nas wartość ma inny wpływ na uszeregowanie. W literaturze, preferowane są wartości bardzo niskie (można się również spotkać z implementacjami, w których prawdopodobieństwo wystąpienia mutacji wynosi 1%).

TEST 5 - LICZBA ZADAŃ (DLA STAŁEGO SUMARYCZNEGO CZASU TRWANIA OPERACJI)

Ostatnim, i możliwe najciekawszym testem jaki postanowiliśmy przeprowadzić jest badanie jak **zmiana liczby zadań** wpłynie na rozwiązanie problemu szeregowania zadań algorytmem genetycznym. Założyliśmy, że wszystkie zadania, dla każdej instancji problemu (każdej liczby zadań), będą miały ten sam sumaryczny czas, co oznacza, że "pliki" mające mniejszą liczbę zadań, będą potrzebować dłuższych zadań, niż pliki posiadające więcej zadań. Aby przeprowadzić ten test, musieliśmy zmienić dopuszczalną długość zadania do **130 jednostek**, ponieważ inaczej nie udałoby się uzyskać pełnego wykresu, jaki chcieliśmy przedstawić.

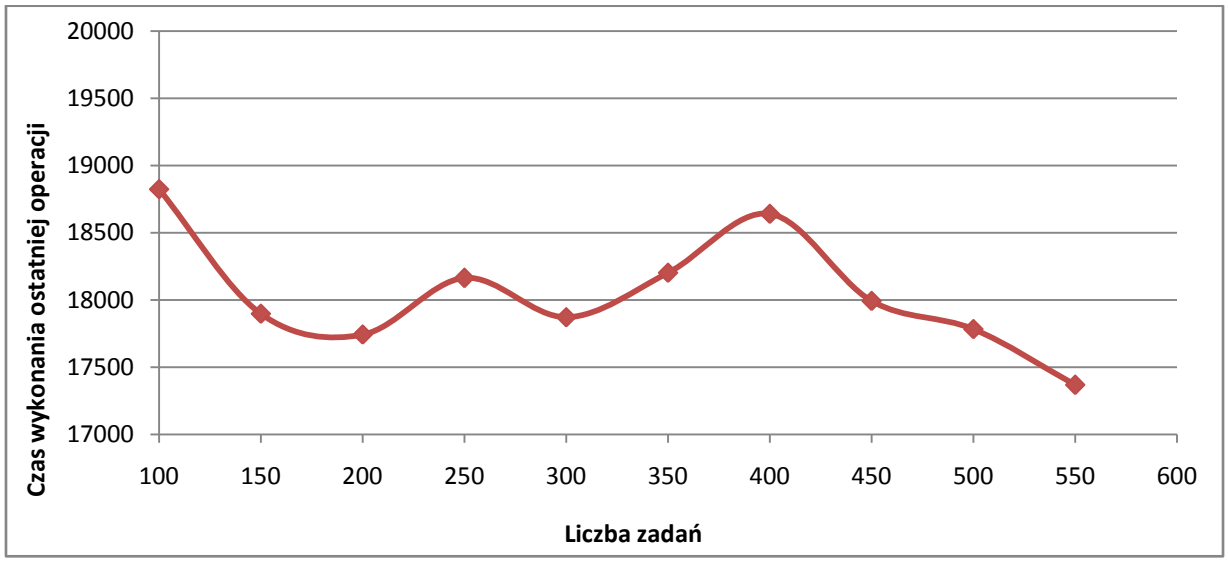


Wykres 5. Zależność zmiany wartości funkcji celu, od prawdopodobieństwa mutacji.

	Liczba zadań									
pomiar	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**	**550**
1	1711531	2385972	2994502	3577009	4736969	5821569	6551857	7662446	8612676	9743028
2	1580462	2380211	2988340	3608133	4750322	5764267	6586255	7678058	8689380	9750331
3	1669349	2384606	2984809	3516670	4762842	5887560	6563450	7598330	8718366	9754213
4	1632088	2402504	2969755	3555654	4708561	5860589	6602987	7526753	8777955	9765396
5	1656592	2389977	2981950	3555610	4681286	5861597	6426748	7631259	8679430	9688864
6	1656146	2383793	2980161	3548207	4770354	5868422	6591191	7683210	8610514	9760106
7	1632404	2395313	2997018	3506419	4732925	5803802	6566382	7655148	8654519	9797943
8	1647513	2390426	2993184	3628581	4783241	5808179	6507620	7614390	8584030	9788713
9	1645198	2357350	2988984	3520999	4719222	5776860	6519018	7758086	8742834	9773435
10	1627356	2370325	2990279	3553837	4764028	5743495	6584631	7640947	8772083	9763540
ŚREDNIA	1645864	2384048	2986898	3557112	4740975	5819634	6550014	7644863	8684179	9758557

Tabela 5. Wyznaczone wartości funkcji. Dla każdego punktu pomiarowego po 10 pomiarów.

Tak jak na *wykresie 5* widać, liczba zadań miała bardzo ważny wpływ na jakość wyniku. I mimo iż sumaryczna wartość trwania operacji jest stała dla każdego punktu, to wartość *Cmax* wzrasta liniowo. Należy jednak zwrócić koniecznie uwagę, co jest naszą funkcją celu, a jest nią **suma czasów zakończenia wszystkich operacji.** I to właśnie odgrywa tutaj prymarną rolę. Ponieważ, tak jak widać na *wykresie 5b* czas uszeregowania ostatniej operacji niewiele się zmienia wraz ze wzrostem liczby zadań, ale liczba tych zadań jest większa, przez co również dla naszej funkcji celu suma czasów zakończenia będzie rosła. Spodziewaliśmy się tutaj, dużo większych różnic, tym bardziej że w naszym zadaniu liczba przerw była zależna od liczby zadań.



Wykres 5b. Czas uszeregowania ostatniej operacji, względem liczby zadań.

---

4. Wnioski ogólne

Podsumowując całe ćwiczenie algorytm genetyczny jest dość ciekawym zagadnieniem, ze względu między innymi na możliwość implementacji go w wielu różnych problemach, które normalnie są nierozwiązywalne w wielomianowym czasie, a do których wolelibyśmy nie stosować metody przeglądu wyczerpującego (tzw. bruteforce). Nad łatwością implementacji tego algorytmu w kodzie można by spekulować, natomiast korzyści jakie on za sobą niesie są na pewno bardzo duże.

Chciałbym również nadmienić, kilka uwag ogólnych po przeprowadzonych testach :

- Należy bardzo precyzyjnie dobierać, wartości prawdopodobieństw wystąpienia zarówno mutacji jak i krzyżówki, ponieważ mogą one nieść za sobą bardzo poważne skutki, pogarszające działanie naszego algorytmu.

- Dla pewnych przedziałów prawdopodobieństwa, mutacja zamiast wprowadzać urozmaicenie w rozwiązaniach pogarsza wyniki algorytmu.

- To samo tyczy się rozmiaru populacji, jeżeli dobierzemy nieodpowiedni możemy stracić na jakości uzyskanych przez nas wyników.

- Również funkcja naprawiająca potrafi pochłonąć bardzo wiele czasu na naprawienie zepsutego uszeregowania, dlatego należałoby unikać krzyżowania metodą w jednym punkcie. Ponieważ może to narobić nam wielu problemów w uszeregowaniu, które później trzeba będzie naprawić.

References

[1] H. El-Rewini, T.G. Lewis, H.H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice-Hall International Editions, 1994
[2] Gheni Ahmed Ali, Dynamic Task Scheduling in Multiprocessor Real Time Systems Using Genetic Algorithms
[3] Mohammad I. Daoud, Nawwaf Kharma, An Efficient Genetic Algorithm for Task Scheduling in Heterogenous Distributed Computing systems, IEEE Congress on Evolutionary Computation, 2006