

Laboratorium z bioinformatyki

Sekwencjonowanie DNA w oparciu o chipy binarne z błędami pozytywnymi

Prowadzący laboratoria: **dr inż. Piotr Wawrzyniak**

Danylo Brushko
143137

Numer grupy dziekańskiej: L13
Termin zajęć: wtorek 8:00
danylo.brushko@student.put.poznan.pl

Adam Miernicki
135750

Numer grupy dziekańskiej: L13
Termin zajęć: wtorek 8:00
adam.miernicki@student.put.poznan.pl

Termin wymagany: 22-06-2022

Termin oddania: 15-09-2022

Spis treści

1	Opis problemu	1
2	Definicja problemu	1
3	Przedstawienie działania algorytmu dokładnego	1
3.1	Zbiór danych wstępnych	1
3.2	Środowisko implementacji danego algorytmu	1
3.3	Działanie algorytmu	2
3.3.1	Dekodowanie początkowej sekwencji	2
3.3.2	Główna funkcja poszukiwań	3
3.3.3	Znajdywanie potencjalnych następników - search	3
3.4	Szacowana złożoność obliczeniowa	4
3.5	Testy	4
3.5.1	Metodologia wykonywanych testów	4
3.5.2	Wyniki	4
4	Przedstawienie algorytmu genetycznego	5
4.1	Zbiór danych wstępnych	5
4.2	Środowisko implementacji danego algorytmu	5
4.3	Opis działania algorytmu	5
4.3.1	Wyznaczanie początkowych kandydatów	7
4.3.2	Permutacja powstałych kandydatów w celu zbudowania populacji	7
4.3.3	Sposób oceny osobników populacji oraz sposób selekcji	8
4.3.4	Sposób krzyżowania się osobników z populacji	8
4.3.5	Omówienie błędów implementacyjnych oraz sugestia polepszenia algorytmu dla metody crossover	9
4.3.6	Działanie mutacji	9
4.3.7	Omówienie błędów implementacyjnych oraz sugestia polepszenia algorytmu dla metody mutation	9
4.3.8	Ulepszanie kandydatów po wygenerowaniu nowej populacji	10
4.3.9	Warunek stopu	10
4.4	Omówienie dodatkowych zależności występujących w kodzie	10
4.5	Omówienie występujących błędów krytycznych w algorytmie	10
4.6	Szacowana złożoność czasowa	11
4.7	Testy	11
4.7.1	Metodologia wykonywanych testów	11
4.7.2	Wyniki	12
5	Oznaczenie zmiennych w algorytmie oraz sposób uruchamiania programu	13
6	Podsumowanie wyników oraz omówienie pracy działania algorytmów	13

1 Opis problemu

W niniejszym sprawozdaniu zostanie zaprezentowany algorytm rozwiązujący problem sekwencjonowania przez hybrydyzację oparty o nieklasyczny chip DNA - chipie binarnym. W definicji problemu zostanie opisana zasada konstrukcji rozpatrywanego chipu oraz sposób działania algorytmu rozwiązujący problem SBH algorytmem dokładnym oraz algorytmem genetycznym. Przedstawiony zostanie także wynik eksperymentów obliczeniowych, w których testowane będą omawiane algorytmy.

2 Definicja problemu

Rozwiązując dany problem sekwencjonowania DNA w oparciu o chipy binarne z błędami pozytywnymi, warto na początku wyjaśnić kilka podstawowych pojęć składających się na definicję danego zagadnienia. Proces sekwencjonowania składa się z dwóch faz: biochemicznej oraz obliczeniowej. Podczas procesu biochemicznego przeprowadzany jest eksperyment hybrydyzacyjny z pełną biblioteką oligonukleotydów o pewnej ustalonej długości n , nazywanych n -merami. W wyniku przeprowadzonego eksperymentu otrzymujemy zbiór, który określany jest mianem spektrum zawierającym wszystkie podciągi o długości n możliwe do wyróżnienia w analizowanej sekwencji DNA. W procesie sekwencjonowania wyróżniamy dwa możliwe do wystąpienia niedoskonałości, spowodowane procesem hybrydyzacji. W sytuacji, gdy w naszym spektrum nie występuje n -merów, mimo występowania ich w badanej sekwencji, mówimy o występowaniu błędów negatywnych. Natomiast, w przypadku występowania błędów pozytywnych możemy w wynikowym spektrum wyznaczyć n -merowe sekwencje, które przed dokonaniem procesu sekwencjonowania nie występują.

Sam chip binarny DNA będzie wykorzystywał inny alfabet znaków do zapisu sond niż alfabet stosowany w podejściu klasycznym A, C, G, T. Jest to alfabet 8-literowy W, S, R, Y, A, C, G, T, który kolejno tworzy dwa 6-literowe alfabety W, S, A, C, G, T oraz R, Y, A, C, G, T. Chip składa się z dwóch sond i to w każdym z nich będzie stosowany jeden rodzaj alfabetu: S_Z dla alfabetu W, S, A, C, G, T oraz S_P dla alfabetu R, Y, A, C, G, T. Warto pamiętać, że wszystkie litery, oprócz ostatniej, w sekwencji kodowane są za pomocą liter W, S, lub R, Y w zależności od sond. Każda kombinacja dwóch liter W, S lub R, Y odpowiada za kodowanie jednej litery A, C, G, T. Zastosowanie dwóch rodzajów sond umożliwia lepszą kontrolę błędów w procesie hybrydyzacji - proces przetwarzania i znajdowania poszukiwanej sekwencji będzie wykorzystywał dwie równocześnie budowane ścieżki, które wzajemnie będą miały możliwość sprawdzania czy budowana sekwencja jest prawidłowa.

3 Przedstawienie działania algorytmu dokładnego

W celu rozwiązania przedstawionego problemu został napisany algorytm rozwiązujący dany problem sekwencjonowania w sposób dokładny. Oznacza to, że dany algorytm skupia się na znalezieniu potencjalnych wszystkich rozwiązań oraz że dla większych instancji danych wejściowych może okazać się algorytmem mocno kosztownym czasowo. Najważniejsze części funkcjonowania algorytmu zostaną przedstawione poniżej.

3.1 Zbiór danych wstępnych

Początkowe dane zostały pobrane ze strony <https://www.cs.put.poznan.pl/pwawrzyniak/bio/bio.php> Algorytm został przetestowany odpowiednio dla sond w których wynikowa długość sekwencji wynosiła odpowiednio $n = 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500$. Jeśli chodzi o dodatkowe konfigurowalne parametry znajdujące się na wspomnianej wcześniej stronie to zostały one ustawione odpowiednio na:

- $k = 10$
- $sqpe$ ustawiony na maksymalną zawsze wartość równą zawsze $n/4$
- $pose$ ustawiony na maksymalną zawsze wartość równą zawsze $n/2$

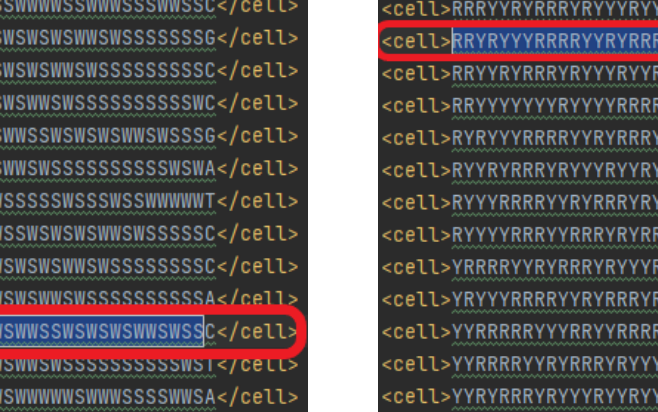
3.2 Środowisko implementacji danego algorytmu

Algorytm został stworzony w języku Python w wersji 3.9 z wykorzystaniem IDE Pycharm Community Edition.

W celu znalezienia pasujących sekwencji, warto przyrzeć się jak przygotowane są nasze dane wejściowe. Dla dwóch rodzajów sond S_Z oraz S_P zostały wygenerowane ciągi o długości k W, S oraz R, Y. W każdym ciągu ostatnia litera została zapisana za pomocą alfabetu A, G, C, T. Głównym zadaniem algorytmu będzie znajdowanie listy potencjalnych kandydatów posiadających największe pokrycie dla tworzonej sekwencji. W przypadku znalezienia takich kandydatów dla dwóch rodzajów sond, dokonujemy sprawdzenia, czy wybrani kandydaci dla obu sond posiadają takie same n-liczne pokrycie oraz czy są zakończone tym samym nukleotydem. W przypadku stwierdzenia poprawności algorytm akceptuje danych kandydatów z każdej z sond do budowanych ścieżek i zaczyna poszukiwania następnym możliwym kandydatów.

3.3.1 Dekodowanie początkowej sekwencji

```
Informacje początkowe:
Start: | AGTACCTGAGACTACAGGC
Długość szukanej sekwencji: | 30
Proba1: | ZZZZZZZZZZZZZZZZZZZN
Proba2: | PPPPPPPPPPPPPPPPPPN
Pierwsza sekwencja Z: | WSWWSSWSWSWSWSWSSS
Pierwsza sekwencja P: | RRYRYYYRRRRYYRYRRY
```



The image displays two side-by-side screenshots of a terminal window, illustrating a mutation in a list of cell identifiers. Each identifier is a string of 20 characters, where the first 17 characters are 'S' or 'W' and the last 3 characters are 'C', 'A', or 'G'. The left screenshot shows the original list, with the 18th cell, 'WSWSWSWSWSWSWSWS', highlighted by a red rectangle. The right screenshot shows the same list after a mutation, where the 18th cell is now 'RRYYRRYYRRYYRRYY', also highlighted by a red rectangle. The mutation has replaced the 'S' characters in the 18th cell with 'R' characters.

Cell ID	Cell ID
<cell>SSWSWSWSWSWSWSSSSC</cell>	<cell>RRRRYYRRYYRRYYRYC</cell>
<cell>SSWWWSWSWSWSWSWSC</cell>	<cell>RRYYRRYYRRYYRYRYA</cell>
<cell>SWSWSWSWSWSWSSSSSSG</cell>	<cell>RRYYRRYYRRYYRRYYRC</cell>
<cell>SWSWSWSWSWSWSSSSSSSSC</cell>	<cell>RRYYRRYYRRYYRRYYRYC</cell>
<cell>SWSWSWSWSWSWSSSSSSSSSWC</cell>	<cell>RRYYRRYYRRYYRRYYRRRA</cell>
<cell>SWSWSWSWSWSWSSSSSSSSSG</cell>	<cell>RYRYRRYYRRYYRRYYRYC</cell>
<cell>SWSWSWSWSWSWSSSSSSSSSWA</cell>	<cell>RYRYRRYYRRYYRRYYRYT</cell>
<cell>WSSSSWSWSWSWSWWWT</cell>	<cell>RYRRYYRRYYRRYYRYC</cell>
<cell>WSSWSWSWSWSWSSSSSSC</cell>	<cell>RYRRYYRRYYRRYYRYRT</cell>
<cell>WSWSWSWSWSWSSSSSSSSC</cell>	<cell>YRRYYRRYYRRYYRYRYC</cell>
<cell>WSWSWSWSWSWSSSSSSSSA</cell>	<cell>YRYRRYYRRYYRRYYRYC</cell>
<cell>WSWSWSWSWSWSWSWS</cell>	<cell>YRRYYRRYYRRYYRRYYC</cell>
<cell>WSWSWSWSWSWSWSWSI</cell>	<cell>YRRYYRRYYRRYYRYRYC</cell>
<cell>WSWWWSWSWSWSSSSSWSA</cell>	<cell>YRRYYRRYYRRYYRYRYA</cell>
<cell>WSSWSWSWSWSWSSSSC</cell>	<cell>YRRYYRRYYRRYYRYRYC</cell>
<cell>WWSWSWSWSWSWSWSG</cell>	<cell>YRRYYRRYYRRYYRYRYT</cell>

3.3.2 Główna funkcja poszukiwań

Główną funkcją odpowiedzialną za znalezienie szukanych sekwencji jest **bin_chip_recursion**. Jest to funkcja rekurencyjna, której warunkiem stopu jest znalezienie szukanej sekwencji lub zakończenie programu przez brak możliwych do znalezienia i dopasowania następnych kandydatów.

Jeśli warunki stopu nie są spełnione, funkcja przystępuje do znalezienia następnych potencjalnych możliwych maksymalnych dopasowanych sekwencji. Za znalezienie następnych możliwych kandydatów odpowiada funkcja **search**. Jeśli potencjalni kandydaci zostali znalezieni uruchamiana jest funkcja **addOutgoingVerticesToList**, która odpowiada za dodawanie znalezionych kandydatów do listy rozwiązania i budowania tekstowo danej sekwencji.

Algorytm 1: Główna pętla poszukiwań

```
1 bin_chip_recursion (first, second, created_by_algorithm_sequence_length):
2   if created_by_algorithm_sequence_length == expectedLength then
3     return create_by_algorithm_sequence_length
4   SZcandidate, SPcandidate, level ← search(SZparameters, SPparameters);
5   if return_from_add == True or created_by_algorithm_sequence_length > searched_sequence_length then
6     delete_last_sequence_for_both_polls
7   addOutgoingVerticesToList
```

W głównej pętli poszukiwań następników dla naszej sekwencji warunkami stopu pętli jest:

- znalezienie sekwencji równej długości zmiennej *expectedLength* zadeklarowanej w pliku xml,
- brak możliwości znalezienia potencjalnych następników (funkcja *foundSolutions*) poszukiwanej sekwencji dla sondy *S_Z* oraz *S_P*

W przypadku spełnienia wszystkich warunków tworzona zostaje sekwencja poprzez dołączenie znalezionych sekwencji nukleotydów. Głębokość nakładania się nukleotydów determinuje zmienna *level* określana przez funkcję *findSolution*.

3.3.3 Znajdywanie potencjalnych następników - search

W celu znalezienia następnych potencjalnych kandydatów możliwych do przyłączenia dla danej sekwencji, funkcja *search* dekoduje ostatnie litery stworzonej dotychczas sekwencji, odpowiednio dla sond *S_Z* oraz *S_P*. Szukanie potencjalnych dopasowań rozpoczynamy od najwyższego możliwego dopasowania sekwencji. Oznacza to, że dla *k*-długiej sekwencji największe możliwe dopasowanie jest równe *k* - 1. W naszym pseudokodzie poziom dopasowania został oznaczony przez zmienną *level*. Następnie utworzone słowniki *ols1* oraz *ols2* przekazują informację czy dane sekwencje ze zbiorów *s1* oraz *s2* były wykorzystane. Jeśli znalezieni kandydaci nie zostali wykorzystani, zostają oni dodani jako potencjalni możliwi kandydaci do dołączenia do budowanych sekwencji przez **bin_chip_recursion**.

Algorytm 2: Znajdywanie potencjalnych następników

```
1 Search (firstString, secondString, ols1, ols2, s1, s2):
2   firstString, secondString ← transform_last_letters(firstString, secondString)
3   found_seq_and_index_list_1, level_all_seq_list_1, level1 ← found_element(first, s1, ols1)
4   found_seq_and_index_list_2, level_all_seq_list_2, level2 ← found_element(second, s2, ols2)
5   if level1 == 0 or level2 == 0 then
6     return no_found_solution
7   else
8     return first, second, res1, res2, val1, val2
```

Algorytm kolejno poszukuje wszystkich możliwych następników dla obu rodzajów sond. W przypadku znalezienia podobnych sekwencji, z takim samym ostatnim nukleotydem, zwracana jest informacja o znalezieniu odpowiadającego kandydata.

Warto zauważyć, że algorytm został stworzony tak, że nie znajduje on tylko jednego maksymalnego kandydata. Algorytm przygotowuje listę wykorzystując wszystkie możliwe sekwencje które pozostały, układając ją od najgorszych do najlepszych dopasowań. W efekcie czas trwania danego algorytmu jest dłuższy, w porównaniu do zastosowania wspomnianej wcześniej innej metody, za to jednak algorytm jest bardziej uniwersalny i mógłby z powodzeniem po kilku przekształceniach być zastosowany do np. znajdowania błędów negatywnych albo znajdowania pokryć w sekwencjach gdzie nie zawsze dane pokrycie musi być maksymalne.

3.4 Szacowana złożoność obliczeniowa

Algorytm w trakcie działania w najbardziej pesymistycznym przypadku do znalezienia każdego możliwego maksymalnego dopasowania będzie musiał przejrzeć maksymalną ilość sekwencji która nie została wykorzystana. Oznacza to że złożoność czasowa będzie równa $O(n_{SW}! + m_{RY}!)$ (gdzie liczby n oraz m oznaczają liczbę zbiorów sond S_Z oraz S_P).

3.5 Testy

3.5.1 Metodologia wykonywanych testów

W testach wzięła udział wygenerowana wcześniej grupa sekwencji wspomniana w **3.1 Zbiór danych wstępnych**. W celu wyeliminowania rozbieżności wyników dla każdej sekwencji zostało wykonanych 30 testów i jako wynik został podany ich wynik arytmetyczny.

3.5.2 Wyniki

```
Algorytm dokładny
*****
* Sekwencja n: 30  Czas: 0.006
* Sekwencja n: 40  Czas: 0.026
* Sekwencja n: 50  Czas: 0.067
* Sekwencja n: 60  Czas: 0.125
* Sekwencja n: 70  Czas: 0.24
* Sekwencja n: 80  Czas: 0.307
* Sekwencja n: 90  Czas: 0.458
* Sekwencja n: 100  Czas: 0.601
* Sekwencja n: 200  Czas: 6.013
* Sekwencja n: 300  Czas: 11.588
* Sekwencja n: 400  Czas: 26.689
* Sekwencja n: 500  Czas: 51.989
*****
```

Rysunek 2: Osiągnięte wyniki dla algorytmu dokładnego

Możemy zauważyć, że dla bardzo małych sekwencji algorytm działa bardzo szybko. Jednakże wraz ze wzrostem populacji zgodnie z szacowaną złożonością obliczeniową czas rośnie i dla $n = 400$ wynosi on już prawie 30 sekund. Możemy zauważyć, że dla coraz co większych sekwencji efektywność czasowa mocno spada, dlatego też ten algorytm ten nie powinien być stosowany dla dużych sekwencji.

4 Przedstawienie algorytmu genetycznego

4.1 Zbiór danych wstępnych

Początkowe dane zostały pobrane ze strony <https://www.cs.put.poznan.pl/pwawrzyniak/bio/bio.php> Algorytm został przetestowany odpowiednio dla sond w których wynikowa długość sekwencji wynosiła odpowiednio $n = 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500$. Jeśli chodzi o dodatkowe konfigurowalne parametry znajdujące się na wspomnianej wcześniej stronie to zostały one ustawione odpowiednio na:

- $k = 10$
- $sqpe$ ustawiony na maksymalną zawsze wartość równą zawsze $n/4$
- $pose$ ustawiony na maksymalną zawsze wartość równą zawsze $n/2$

4.2 Środowisko implementacji danego algorytmu

Algorytm został stworzony w języku Python w wersji 3.9. W celu minimalnego polepszenia rezultatów został wykorzystany dodatkowy interpreter Cython w celu translacji powstałego kodu w języku Python na język C. Czas pomiędzy wersją Python oraz Cython w wykonanych testach był marginalny co oznacza w rezultacie, że osiągnięte rezultaty przez algorytm genetyczny będą porównywane z algorytmem dokładnym.

4.3 Opis działania algorytmu

Na samym początku opisu zaimplementowanego algorytmu trzeba uwzględnić fakt, że algorytm posiada pewien poważny błąd rzutujący na poprawność działania już na poziomie logicznym (autorzy nie przewidzieli możliwości występowania pewnej sytuacji; szczegółowy opis danego błędu znajduje się w rozdziale **”Omówienie występujących błędów w zaimplementowanym algorytmie”**).

W przesłanym programie za wykonywanie algorytmu genetycznego odpowiada klasa **GeneticAlgorithm**, posiadająca jedną metodę odpowiedzialną za wykonanie algorytmu o nazwie *run()*. Warto zauważyć, że klasa nie ma określonej samej w sobie deklaracji funkcji selekcji, warunku stopu oraz generowania pierwszych kandydatów. Zamiast tego klasa przyjmuje zewnętrznie zadeklarowanej funkcje zdefiniowane przez twórcę kodu. Poprzez słowo 'kandydat' w dalszej części opisu algorytmu będziemy rozumieć wygenerowane sekwencje składające się z maksymalnych pokryć, służące do budowania następnych populacji.

Działanie algorytmu możemy opisać za pomocą kilku występujących po sobie faz które się zapętłają. Są to:

1. **Faza generowania kandydatów** (występuje tylko raz na początku działania algorytmu)
2. **Budowanie populacji opartej o losową ilość permutacji na podstawie wygenerowanych kandydatów**
3. **Podanie wygenerowanej populacji selekcji w celu wytworzenia nowej populacji**
4. **Krzyżowanie wyselekcjonowanych osobników ze starej populacji w celu wytworzenia nowej**
5. **Potencjalne wystąpienie mutacji na wytworzonym nowym osobniku**
6. **Jeśli w nowej populacji wygenerowany osobnik posiada sekwencję, która składa się wszędzie z maksymalnych pokryć to występuje komunikat o znalezionej sekwencji i zakończenie programu**
7. **Jeśli w nowej populacji znaleziono maksymalne pokrycie lub też pokrycia to zastąpienie starej listy kandydatów, nowszą ze scalonymi kandydatami posiadającymi maksymalne pokrycie. Powrót do punktu drugiego**

W porównaniu do algorytmu dokładnego nie występuje tutaj faza pobierania i dekodowania startowej sekwencji (oznaczona w wygenerowanym pliku .xml przez słowo *start*). W przesłanym pliku klasa *GenethicAlghorithm* co prawda przyjmuje pole o nazwie *self.start*, jednak nie jest ono wykorzystywane w finalnej wersji algorytmu.

Algoritm 3: GeneticAlgorithm

```
1 GeneticAlgorithm:
2   SWSetPop, RYSetPop, SWGenCan, RYGenCan, PopLen  $\leftarrow$  first_generation_func
3   counterSW = 0 counterRY = 0 while True do
4     newPopulationSW, newPopulationRY  $\leftarrow$  select_model_SW_RY
5
6     ###generate for SW population
7     while newPopLen  $\neq$  PopLen do
8       child  $\leftarrow$  SWSetPop[randPopMember1].crossover(SWSetPop[randPopMember2])
9       if child  $\neq$  None then
10         if genRandomNumber  $\leq$  mutationProbability then
11           child.mutation
12         if not child.StringSeqSW in newPopulationSW.StringSeqSW then
13           newPopulationSW.append(child)
14
15     ###generate for RY population
16     while newPopLen  $\neq$  PopLen do
17       child  $\leftarrow$  RYSetPop[randPopMember1].crossover(RYSetPop[randPopMember2])
18       if child  $\neq$  None then
19         if genRandomNumber  $\leq$  mutationProbability then
20           child.mutation
21         if not child.StringSeqRY in newPopulationRY.StringSeqRY then
22           newPopulationRY.append(child)
23
24     BestMatchSW  $\leftarrow$  max(newPopulationSW.child.fitness)
25     BestMatchRY  $\leftarrow$  max(newPopulationRY.child.fitness)
26
27     if counterSW > 10 then
28       SWSetPop = generate_new_populationSW
29       counterSW = 0
30     else
31       ###if found max cover improve candidates and generated new population based on new created candidates
32       if BestMatchSW.found_max_cover then
33         PopulationSW, SWGenCan  $\leftarrow$  ImproveCanSW_AND_GenNewPopSW counterSW = 0
34       else
35         SWSetPop  $\leftarrow$  newPopulationSW
36
37     if counterRY > 10 then
38       RYSetPop = generate_new_populationRY counterRY = 0
39     else
40       ###if found max cover improve candidates and generated new population based on new created candidates
41       if BestMatchRY.found_max_cover then
42         PopulationRY, RYGenCan  $\leftarrow$  ImproveCanRY_AND_GenNewPopRY
43         counterRY = 0
44       else
45         RYSetPop  $\leftarrow$  newPopulationRY
46
```

4.3.1 Wyznaczanie początkowych kandydatów

Pod pojęciem wyznaczania kandydatów rozumiemy wygenerowanie takich sekwencji z których następnie będziemy mogli 'budować' nasze pokolenia. To czym musi się odznaczać kandydat to posiadanie maksymalnych pokryć w swoim stworzonym ciągu. W innym przypadku łączenie pokoleń posiadających łańcuchy nie mające maksymalnych pokryć nigdy nie doprowadziłoby nas do znalezienia szukanej końcowej sekwencji. Przykład wygenerowanych pierwszych kandydatów dla obydwu sond znajduje się poniżej.

```
Użytkane elementy które posłużą do budowania następnych populacji

Dla sondy SW:
0: SWSWSWSWSWSWSSSSSSSSSA
1: SSWWWWSWSWWSSSSSSSSC
2: SWSWSWSSSSSSSSSSSWSWA
3: SWSWSWSWSWSWSSSSSC
4: WSSSSWSWSWSWSWWWT
5: WSSWSWSWSWSWSSSSSC
6: WSWWWWSWSWSSSSSSSWA
7: WWSWSSSSSWWWSWSWSG
8: WSWWSWSWSWSWSWSSC

[[0, 2, 9, 3, 10], [1], [4, 12, 6], [5, 14], [7], [8], [13], [15], [11]]

Dla sondy RY:
0: RRRYYRYYRYYRYYRYYRYA
1: RRYYYRYYRYYRYYRRC
2: RYYRYYRYYRYYRYYRYYG
3: RYRYYRYYRYYRYYRYYRC
4: RYYYYRYYRYYRYYRYRT
5: YRRRYYRYYRYYRYYRYC
6: RRYYYYYRYYRYYRYYRRA
7: YRRRYYRYYRYYRYYRRC
8: YYYRYYRYYRYYRYYRYT

[[0, 1, 3, 6, 13], [2], [7, 14, 12], [5, 10], [8], [9], [4], [11], [15]]
```

Wygenerowani kandydaci dla sekwencji n=30 dla sondy SW oraz RY

Możemy więc zauważyć, że pierwszy wygenerowany kandydat SSWSWSWSWSWSSSSSSSSSSSA składa się z części sond znajdujących się w pliku *.xml* po kolei z linii: 0, 2, 9 oraz 10. Po wygenerowaniu danych kandydatów funkcja sprawdzając **check_candidates** sprawdza czy wszystkie wygenerowane sekwencje są długości: $(k-1) + n$ (gdzie k oznacza długość jednej sekwencji w sondzie a n ilość wszystkich sekwencji). W przypadku stwierdzenia poprawności wygenerowanych sekwencji algorytm przystępuje do następnej fazy działania - tworzenia pierwszej populacji.

Samo wytworzenie początkowych kandydatów jest podobne na tym etapie do działania algorytmu dokładnego. Program iteracyjnie bierze sekwencje z pliku .xml i szuka sekwencji które będą miały maksymalne pokrycie z pobraną sekwencją. Czyli dla przykładu dla 10 elementowego zbioru dla pierwszej sekwencji zostanie zbadanych maksymalnie 9 kandydatów, w przypadku znalezienia maksymalnego pokrycia sekwencje są ze sobą łączone i dla następnej sprawdzanej sekwencji istniejących kandydatów jest już tylko ośmiu. Podobny schemat działania występował w algorytmie dokładnym z tą różnicą, że program ten nie jest wywoływany rekurencyjnie, aż do znalezienia końcowej sekwencji tylko raz do wygenerowania pierwszych kandydatów.

Implementacja danej funkcjonalności znajduje się w pliku **firstPopulation.pyx** pod nazwą **first_sequence_k_minus_1_for_both_poll**.

4.3.2 Permutacja powstałych kandydatów w celu zbudowania populacji

Po wygenerowaniu pierwszych kandydatów następuje faza losowego permutowania powstałych kandydatów. Od razu trzeba podkreślić fakt, że nie są tworzone wszystkie możliwe permutacje a jedynie jakaś ich część, gdyż wygenerowanie wszystkich możliwych permutacji jest niepotrzebne i bardzo czasochłonne obliczeniowo. Dla sekwencji o długości 500 korzystając z zależności, że długość końcowej sekwencji jest równa: $n + l$ (gdzie n oznacza długości sekwencji, a l ilość pokryć), to dla sekwencji o długości np. 15 potrzeba by było wygenerować $485!$ permutacji. Przy uwzględnieniu faktu, że mamy do czynienia z błędami pozytywnymi i posiadamy nadmiarowe niepasujące sekwencje końcowa ilość permutacji wyniosłaby $(485 + m)!$ (gdzie m oznacza liczbę dodatkowych błędnych sekwencji). W celu optymalizacji pracy algorytmu ilość permutacji jest ograniczona do ilości docelowej populacji określonej przez **GenethicAlgorithm**. Jedynym wyjątkiem od tej reguły jest fakt wystąpienia sytuacji gdy ilość permutacji z wygenerowanych początkowych kandydatów jest mniejsza niż wielkość populacji określona przez klasę **GenethicAlgorithm** (np. niemożliwe będzie wygenerowanie populacji równej 100 dla 3 kandydatów gdyż $3!$ jest równe 6). W przypadku wystąpienia tej sytuacji przedział populacji jest odpowiednio zmniejszany do połowy wielkości permutacji wygenerowanych kandydatów. Twórcy algorytmu staneli też przed wyborem jaki zakres permutacji powinien być generowany: czy zawsze ze stałą długością, czy też ze zmienną długością. Ostatecznie zdecydowaną się na drugą opcję gdyż stwierdzono, że sama długość powstałych wcześniej kandydatów też może być różna, a dodanie osobnej funkcji określającej długość stworzonej sekwencji nie przyniosłaby żadnego pozytywnego wpływu jeśli chodzi o jakość i czas przetwarzania algorytmu. Przykładowe wygenerowanie populacji o wielkości 20 dla sekwencji $n=30$ znajduje się poniżej.


```
ŚCIEŻKI PRZED CROSSOVER
[0, 8, 6, 3, 7, 5]
[3, 0, 2, 4]

ŚCIEŻKA PO CROSSOVER
[3, 8, 6, 3, 7, 5]
```

Przykładowe łączenie dwóch ścieżek

Sekwencja [0, 8, 6, 3, 7, 5] oraz [3, 0, 2, 4] zostały podzielone w losowych miejscach. Sekwencja pierwsza została podzielona na części: [0], [8, 6, 3], [7], [5], a sekwencja druga na [3], [0], [2], [4]. Następnie losowo ze sobą zostały połączone wybrane fragmenty. Z sekwencji drugiej została pobrana pierwsza część, a z drugiej część druga, trzecia oraz czwarta. Możemy zauważyć jednak niestety, że w wyniku tych operacji może wystąpić zjawisko wystąpienia duplikacji (co jak było wspomniane wcześniej nie jest sytuacją możliwą w przypadku gdy chcemy znaleźć prawidłowe rozwiązanie). W celu eliminacji tego zjawiska, algorytm dokonuje detekcji danych powtarzających się punktów oraz generuje listę kandydatów którzy w danej ścieżce nie byli wykorzystani. Następnie algorytm dokonuje losowego wyboru elementu z listy, usuwa wykorzystany element z listy kandydatów nie wykorzystanych i podstawia pod powtarzający się element. Proces ten powtarza się dopóki wszyscy zduplikowani kandydaci nie zostaną wyeliminowani. Ostatecznie metoda crossover może utworzyć nowy obiekt będący obiektem typu **Text**, będący potomkiem starego pokolenia, a następnie przystąpić do oceny powstałego obiektu.

4.3.5 Omówienie błędów implementacyjnych oraz sugestia polepszenia algorytmu dla metody crossover

Analizując zasadę działania **crossover** możemy zauważyć, że część pracy jest wykonywana dodatkowo i mogłaby zostać wyeliminowana. Chodzi dokładniej o typ danych które zostaje dzielony czyli ścieżkę typu PATH (posiadającą numerację kandydatów). Problem ten zaczyna być zauważalny dla bardzo dużych instancji i dużej ilości kandydatów. Jest to też jedna z przyczyn dlaczego algorytm był testowany dla stosunkowo małej ilości populacji (wynoszącej średnio do 100 osobników na populację). Po wygenerowaniu ścieżki typu PATH i wyeliminowaniu duplikatów, cała sekwencja budowana jest od początku co zajmuje niepotrzebny czas. Czyli dla 3 krzyżowań w przypadku dla 500 kandydatów i tak będziemy musieli znaleźć 499 wartości pokryć oraz wykonać 499 operacji łączeń kandydatów. Dla coraz większej ilości kandydatów problem ten staje się coraz bardziej widoczny. Proponowaną alternatywą jest dodanie dwóch dodatkowych kolumn do wartości dataframe. W oddawanej wersji typ ten dla generowanych naszych osobników posiada dwa pola: ['SEQUENCE'] oraz ['PATH']. Dodanie nowych wartości ['NEW_STRING'] oraz ['COVER'] umożliwiłoby przyspieszenie pracy danej metody. Pole ['NEW_STRING'] zbierałoby informacje oznaczającą jaka była wielkość łańcucha po każdym dołączeniu nowej sekwencji, a pole ['COVER'] jakie było między nimi pokrycie. Ostatecznie posiadając informację o: miejscu przecięcia, nowym dołączonym kandydacie, wielkości łańcucha gdzie znajduje się kandydat oraz pokryciu byłibyśmy w stanie poprawnie dokonać przecięcia oraz dołączenia nowej części sekwencji z innej populacji. Oznacza to, że liczba operacji potrzebnej do zbudowania nowego łańcucha (nie uwzględniając podstawieniach nowych wartości do [NEW_STRING] oraz [COVER]) zakładając, że nie wystąpi żadna duplikacja kandydatów zmalałaby do 3 operacji. Część pomysłów w celu przyspieszenia danego algorytmu jest wciąż w oddanym kodzie, jednak ze względu na ograniczenia czasowe została ona zakomentowana i nie pełni ona teraz żadnej funkcji.

4.3.6 Działanie mutacji

Podczas tworzenia obiektu pobierając informację o częstotliwości występowania mutacji z klasy GeneticAlgorithm self.mutation_probability, może dojść do wystąpienia mutacji na świeżo stworzonym obiekcie. Metoda _perform_mutation bierze stworzoną ścieżkę nowego osobnika i na jej podstawie tworzy ścieżkę kandydatów którzy jeszcze nie zostali wykorzystani. Następnie wybiera jeden element z danej listy i zastępuje jeden element ze stworzonej ścieżki. Twórcy programu mogli wybrać dwie ścieżki działania mutacji: dokonania zamiany losowego elementu z wygenerowanej ścieżki i dokonanie oceny wykonanej mutacji lub też zamieniania po kolei jednego elementu z wygenerowanej ścieżki i wybrania mutacji która osiągnęła najlepszy wynik dopasowania. Autorzy zdecydowali się na drugą wersję programu.

4.3.7 Omówienie błędów implementacyjnych oraz sugestia polepszenia algorytmu dla metody mutation

Sposób zaimplementowanej mutacji mimo bardzo dużego nakładu czasowego daje gwarancję, że dla losowo wybranej ścieżki (jeśli istnieje dla niego dopasowanie) to dla niego takie dopasowanie znajdzie. Zdecydowanie się na mutację każdego elementu po kolei niestety niesie ze sobą także wzrost czasu obliczeniowego potrzebnego na wykonanie funkcji _perform_mutation. Przykład rozwiązania tego problemu został już przedstawiony w sekcji **4.3.5 Omówienie błędów implementacyjnych oraz sugestia polepszenia algorytmu dla metody crossover**.

4.3.8 Ulepszanie kandydatów po wygenerowaniu nowej populacji

Każdy osobnik oprócz atrybutu fitness posiada także atrybut `self.found_max_cover`. Umieszczane tam są znalezione maksymalne dopasowania jakie udało się znaleźć podczas generowania nowej populacji. Jeśli w czasie selekcji, wyznaczeni osobnicy posiadają znalezione maksymalne dopasowania to istniejący kandydaci zostają "ulepszeni" to znaczy dwóch kandydatów zostaje ze sobą złączonych w jednego kandydata.

4.3.9 Warunek stopu

Wraz z ulepszaniem się kandydatów kiedy zostanie wygenerowany kandydat który swoją długością będzie równy szukanej sekwencji oznacza to, że szukana sekwencja została znaleziona. Gdy odpowiednio dla obu sond zostaną znalezione sekwencje, program przystępuje do dekodowania obu sekwencji do sekwencji końcowej i następuje koniec programu.

```
*****
ZNALEZIONA SEKWENCJA: SSWSSSWSSWSWSSSSWSWSSWSSWWWSSSSWSWSSWSSSSWSWSS
CZAS: 22.363138914108276
*****
*****
ZNALEZIONA SEKWENCJA: YYRRYYRRYYRRRRYYRRYYRRYYRRYYRRYYRRYYRRYYRRYYRR
CZAS: 24.68570065498352
*****
CCTGGCTTG6TCAGG6CTGAGCTTCTGGAATGGCAGACAAGGTCGACAGAGTGTGACATGCAAAGGGCTTG
26.21375322341919
```

Przykład wyjścia programu i zakodowanego wyniku

4.4 Omówienie dodatkowych zależności występujących w kodzie

W przesłanej pracy ze względu na charakter działania algorytmu genetycznego, który do poprawnej pracy nie potrzebuje komunikacji i wymiany wyników między dwoma sondami zdecydowano się podzielić pracę dla sondy SW oraz sondy RY na dwa potomne procesy. Dodatkowo, w celu polepszenia czasu działania algorytmu genetycznego w przypadku gdy przez określoną ilość populacji algorytm nie będzie w stanie znaleźć ani maksymalnego dopasowania, ani polepszyć swojego wyniku, następuje wygenerowanie całkowicie nowej populacji. Testy wykazały, że implementacja tego mechanizmu znacznie przyspiesza czas dojścia do wyniku.

4.5 Omówienie występujących błędów krytycznych w algorytmie

Analizując pseudokod zaimplementowanego algorytmu genetycznego oraz porównując go do np. zaprezentowanej przykładowej implementacji algorytmu na zajęciach możemy zauważyć, że została dodana do kodu nowa funkcjonalność. Mianowicie w przypadku znalezienia maksymalnego pokrycia następuje scalanie ze sobą kandydatów. Podejście to zostało zaproponowane przez autorów aby polepszyć szybkość i czas działania programu, gdyż bez istnienia tej funkcji, algorytm nie był w stanie dojść do prawidłowego rozwiązania. Mogła odpowiadać za to źle zaimplementowana funkcja oceny, jednak autorom nie udało się ustalić gdzie leży przyczyna aż tak długiego czasu działania programu, i po wielokrotnym przerabianiu funkcji oceny zdecydowano się na zaimplementowanie operacji scalania kandydatów gdzie znaleziono maksymalne pokrycie. Rozwiązanie to działało, dla wygenerowanych wcześniej przykładach, jednak wraz z próbą zmniejszania długości wszystkich sekwencji algorytm zaczął czasami zwracać błędne wyniki. Autorzy doszli do wniosku, że z powodu zaimplementowania scalania kandydatów algorytm nie będzie w pełni działał poprawnie dla wygenerowanych sond w których znajduje się para sekwencji składająca się z identycznych liter i różniąca się tylko ostatnią literą. Jest to spowodowane faktem, że dla obu takich sekwencji z pary znalezione dopasowanie zawsze będzie poprawne, co dla algorytmu oznacza, że nie ma znaczenia którego (losowego) kandydata weźmie do łączenia dwóch kandydatów w celu stworzenia lepszej skróconej listy kandydatów. Autorzy kodu nie przewidzieli zaitnienia tej sytuacji i przepraszają za pojawienie się tego błędu.

W celu wyeliminowania danego negatywnego zjawiska można zastosować kilka metod. Pierwszą, a zarazem najprostszą wydaje się wyeliminowanie systemu polepszania kandydatów. Jednakże nawet jeśli dany problem wyeliminujemy to w takiej sytuacji dany problem wciąż występuje i łatwo można wyobrazić sobie sytuację gdy w procesie budowania i promowania coraz to lepiej dopasowanych sekwencji zostanie dołączona sekwencja która nie powinna się tam znaleźć ale wraz z np. innymi błędami pozytywnymi tworzy bardzo długą sekwencję z maksymalnymi pokryciami. Wygenerowana w ten sposób sekwencja będzie bardzo wysoko promowana i istnieje możliwość, że generowana sekwencja też z maksymalnym pokryciem ale z poprawną literą na końcu będzie odrzucana. Ostatecznie, autorzy stwierdzają, że

mogłaby to być jedna z możliwości wyeliminowania danego błędu jednakże nie są oni do końca pewni czy ta metoda byłaby do końca poprawna.

Inną metodą proponowaną przez autorów byłoby sprawdzanie już na etapie znalezienia maksymalnego pokrycia przez jakiegoś kandydata, sprawdzenia czy w danej sondzie nie znajduje się inny kandydat, składający się z takich samych liter oprócz ostatniej. W przypadku znalezienia takich kandydatów, algorytm generowałby x ilość dołączeń do sekwencji.

Na przykład:

Posiadając sekwencję **SWWWWSSSWSWSS** i znajdując jej dopasowanie **WWWWSSSWSWSST** sprawdzamy czy nie istnieją inni kandydaci zakończeni inną literą. W przypadku gdy znajdziemy kandydatów np. **WWWWSSSWSWSSA** oraz **WWWWSSSWSWSSC** posiadamy 3 potencjalnych kandydatów którzy mają pokrycie maksymalne. Na etapie więc polepszenia kandydatów wykorzystujemy pozostałych kandydatów (których algorytm w czasie swojej realizacji bezpośrednio nie znalazł ale są podobni do znalezionej sekwencji) i generuje 3 ulepszonych kandydatów: **SWWWWSSSWSWSST**, **SWWWWSSSWSWSSA**, **SWWWWSSSWSWSSC**.

Dalsza praca algorytmu powinna ułożyć się tak, że oczywiście tylko ta sekwencja która jest poprawna doprowadzi program do prawidłowego wyniku i zakończy pomyślnie program. Pomysł ten był bliski implementacji przez autorów, jednakże znów przez ograniczenia czasowe nie został on zrealizowany.

4.6 Szacowana złożoność czasowa

Oszacowanie złożoności czasowej zaimplementowanego algorytmu autorzy szacują, że wynosi ona:

$O(G_{SW}(PM) + G_{RY}(PM))$:

- G_{SW} oraz G_{RY} oznaczają odpowiednio liczbę generacji SW oraz RY
- P oznacza liczbę populacji
- M oznacza liczbę mutacji

4.7 Testy

4.7.1 Metodologia wykonywanych testów

W testach wzięła udział wygenerowana wcześniej grupa sekwencji wspomniana w **3.1 Zbiór danych wstępnych**. W celu wyeliminowania rozbieżności wyników dla każdej sekwencji zostało wykonanych 30 testów i jako wynik została podana uzyskana wartość arytmetyczna. Algorytm został przetestowany dla różnych parametrów początkowej populacji oraz prawdopodobieństwa wystąpienia mutacji. W testach wartości te wyniosły odpowiednio:

- populacja = [10, 20, 40]
- mutacja = [0.00, 0.05, 0.1, 0.15]

4.7.2 Wyniki

```
*****
Rozmiar populacji:      10
Prawdopodobieństwo mutacji: 0.0
* Sekwencja n: 30 Czas: 1.292
* Sekwencja n: 40 Czas: 1.441
* Sekwencja n: 50 Czas: 1.49
* Sekwencja n: 60 Czas: 1.963
* Sekwencja n: 70 Czas: 1.307
* Sekwencja n: 80 Czas: 1.576
* Sekwencja n: 90 Czas: 1.898
* Sekwencja n: 100 Czas: 2.483
* Sekwencja n: 200 Czas: 3.42
* Sekwencja n: 300 Czas: 7.013
* Sekwencja n: 400 Czas: 10.808
* Sekwencja n: 500 Czas: 16.667
*****
```

```
*****
Rozmiar populacji:      10
Prawdopodobieństwo mutacji: 0.05
* Sekwencja n: 30 Czas: 1.298
* Sekwencja n: 40 Czas: 1.458
* Sekwencja n: 50 Czas: 1.798
* Sekwencja n: 60 Czas: 2.05
* Sekwencja n: 70 Czas: 1.317
* Sekwencja n: 80 Czas: 1.824
* Sekwencja n: 90 Czas: 1.941
* Sekwencja n: 100 Czas: 2.521
* Sekwencja n: 200 Czas: 4.114
* Sekwencja n: 300 Czas: 7.129
* Sekwencja n: 400 Czas: 20.559
* Sekwencja n: 500 Czas: 48.831
*****
```

```
*****
Rozmiar populacji:      10
Prawdopodobieństwo mutacji: 0.1
* Sekwencja n: 30 Czas: 1.409
* Sekwencja n: 40 Czas: 1.466
* Sekwencja n: 50 Czas: 1.474
* Sekwencja n: 60 Czas: 1.999
* Sekwencja n: 70 Czas: 1.33
* Sekwencja n: 80 Czas: 1.886
* Sekwencja n: 90 Czas: 1.99
* Sekwencja n: 100 Czas: 2.776
* Sekwencja n: 200 Czas: 5.364
* Sekwencja n: 300 Czas: 9.626
* Sekwencja n: 400 Czas: 25.382
* Sekwencja n: 500 Czas: 50.078
*****
```

```
*****
Rozmiar populacji:      10
Prawdopodobieństwo mutacji: 0.15
* Sekwencja n: 30 Czas: 1.248
* Sekwencja n: 40 Czas: 1.451
* Sekwencja n: 50 Czas: 1.486
* Sekwencja n: 60 Czas: 1.896
* Sekwencja n: 70 Czas: 1.335
* Sekwencja n: 80 Czas: 1.694
* Sekwencja n: 90 Czas: 2.022
* Sekwencja n: 100 Czas: 3.106
* Sekwencja n: 200 Czas: 5.433
* Sekwencja n: 300 Czas: 15.457
* Sekwencja n: 400 Czas: 36.22
* Sekwencja n: 500 Czas: 66.226
*****
```

```
*****
Rozmiar populacji:      20
Prawdopodobieństwo mutacji: 0.0
* Sekwencja n: 30 Czas: 1.298
* Sekwencja n: 40 Czas: 1.821
* Sekwencja n: 50 Czas: 1.872
* Sekwencja n: 60 Czas: 2.045
* Sekwencja n: 70 Czas: 1.691
* Sekwencja n: 80 Czas: 1.928
* Sekwencja n: 90 Czas: 1.951
* Sekwencja n: 100 Czas: 2.576
* Sekwencja n: 200 Czas: 4.729
* Sekwencja n: 300 Czas: 6.851
* Sekwencja n: 400 Czas: 13.095
* Sekwencja n: 500 Czas: 22.078
*****
```

```
*****
Rozmiar populacji:      20
Prawdopodobieństwo mutacji: 0.05
* Sekwencja n: 30 Czas: 1.311
* Sekwencja n: 40 Czas: 1.914
* Sekwencja n: 50 Czas: 1.854
* Sekwencja n: 60 Czas: 2.244
* Sekwencja n: 70 Czas: 1.655
* Sekwencja n: 80 Czas: 1.863
* Sekwencja n: 90 Czas: 2.014
* Sekwencja n: 100 Czas: 2.744
* Sekwencja n: 200 Czas: 5.763
* Sekwencja n: 300 Czas: 10.773
* Sekwencja n: 400 Czas: 25.411
* Sekwencja n: 500 Czas: 53.865
*****
```

```
*****
Rozmiar populacji:      20
Prawdopodobieństwo mutacji: 0.1
* Sekwencja n: 30 Czas: 1.353
* Sekwencja n: 40 Czas: 1.989
* Sekwencja n: 50 Czas: 2.17
* Sekwencja n: 60 Czas: 2.587
* Sekwencja n: 70 Czas: 1.837
* Sekwencja n: 80 Czas: 1.973
* Sekwencja n: 90 Czas: 2.11
* Sekwencja n: 100 Czas: 3.365
* Sekwencja n: 200 Czas: 5.501
* Sekwencja n: 300 Czas: 22.999
* Sekwencja n: 400 Czas: 40.357
* Sekwencja n: 500 Czas: 91.867
*****
```

```
*****
Rozmiar populacji:      20
Prawdopodobieństwo mutacji: 0.15
* Sekwencja n: 30 Czas: 1.598
* Sekwencja n: 40 Czas: 2.634
* Sekwencja n: 50 Czas: 3.892
* Sekwencja n: 60 Czas: 3.752
* Sekwencja n: 70 Czas: 2.131
* Sekwencja n: 80 Czas: 2.178
* Sekwencja n: 90 Czas: 2.569
* Sekwencja n: 100 Czas: 3.457
* Sekwencja n: 200 Czas: 8.042
* Sekwencja n: 300 Czas: 18.013
* Sekwencja n: 400 Czas: 31.429
* Sekwencja n: 500 Czas: 91.051
*****
```

```
*****
Rozmiar populacji:      40
Prawdopodobieństwo mutacji: 0.0
* Sekwencja n: 30 Czas: 1.647
* Sekwencja n: 40 Czas: 2.02
* Sekwencja n: 50 Czas: 2.034
* Sekwencja n: 60 Czas: 2.259
* Sekwencja n: 70 Czas: 1.715
* Sekwencja n: 80 Czas: 2.03
* Sekwencja n: 90 Czas: 2.298
* Sekwencja n: 100 Czas: 3.241
* Sekwencja n: 200 Czas: 5.347
* Sekwencja n: 300 Czas: 8.151
* Sekwencja n: 400 Czas: 17.146
* Sekwencja n: 500 Czas: 28.22
*****
```

```
*****
Rozmiar populacji:      40
Prawdopodobieństwo mutacji: 0.05
* Sekwencja n: 30 Czas: 1.651
* Sekwencja n: 40 Czas: 2.097
* Sekwencja n: 50 Czas: 2.07
* Sekwencja n: 60 Czas: 2.333
* Sekwencja n: 70 Czas: 1.722
* Sekwencja n: 80 Czas: 2.023
* Sekwencja n: 90 Czas: 2.426
* Sekwencja n: 100 Czas: 3.596
* Sekwencja n: 200 Czas: 7.372
* Sekwencja n: 300 Czas: 12.582
* Sekwencja n: 400 Czas: 32.178
* Sekwencja n: 500 Czas: 54.706
*****
```

```
*****
Rozmiar populacji:      40
Prawdopodobieństwo mutacji: 0.1
* Sekwencja n: 30 Czas: 1.763
* Sekwencja n: 40 Czas: 2.055
* Sekwencja n: 50 Czas: 2.452
* Sekwencja n: 60 Czas: 2.482
* Sekwencja n: 70 Czas: 1.719
* Sekwencja n: 80 Czas: 2.083
* Sekwencja n: 90 Czas: 2.432
* Sekwencja n: 100 Czas: 3.39
* Sekwencja n: 200 Czas: 7.123
* Sekwencja n: 300 Czas: 16.941
* Sekwencja n: 400 Czas: 40.638
* Sekwencja n: 500 Czas: 99.289
*****
```

```
*****
Rozmiar populacji:      40
Prawdopodobieństwo mutacji: 0.15
* Sekwencja n: 30 Czas: 1.918
* Sekwencja n: 40 Czas: 2.275
* Sekwencja n: 50 Czas: 2.264
* Sekwencja n: 60 Czas: 2.669
* Sekwencja n: 70 Czas: 1.952
* Sekwencja n: 80 Czas: 3.024
* Sekwencja n: 90 Czas: 2.771
* Sekwencja n: 100 Czas: 6.404
* Sekwencja n: 200 Czas: 14.491
* Sekwencja n: 300 Czas: 23.542
* Sekwencja n: 400 Czas: 44.447
* Sekwencja n: 500 Czas: 124.556
*****
```

Analizując uzyskane wyniki możemy zauważyć, że wspomniane przez autorów problemy wydajnościowe jeśli chodzi o operacje **crossover** oraz zwłaszcza **mutation**. Wzrost występowania mutacji znacznie zwiększa ilość operacji która w rezultacie nie przyspiesza dojścia do rozwiązania. Zmienienie zasady działania mutacji (na mutowanie jednego zamiast wszystkich i brania najlepszej mutacji) mogłoby przyspieszyć operacje związane z mutacją. Jednak wciąż odsetek występowania mutacji powinien być nie wielki i dalsze testy pokazały, że mutacja na poziomie nie przekraczającej 3% jest dosyć jeszcze dobrą wartością dobraną empirycznie, która nie ma tak dużego wpływu na wynik. Testy wykazały także, że wzrost ilości populacji nie ma też jakiegoś dużego znaczenia jeśli chodzi o czas dojścia do danego wyniku. Jednak trzeba mieć na względzie fakt, że populacje które wzięły udział w teście były niewielkie i że dla bardzo dużych sekwencji z błędami pozytywnymi wzrost populacji mógłby mieć lepszy wpływ na polepszenie wyników obliczeń.

5 Oznaczenie zmiennych w algorytmie oraz sposób uruchamiania programu

Aby uruchomić odpowiednio algorytm dokładny należy uruchomić plik *main_iter.py* natomiast aby uruchomić algorytm genetyczny należy uruchomić *setup.py*. W algorytmie dokładnym możemy zmienić plik wejściowy poprzez podstawienie odpowiedniego argumentu dla funkcji *fun_iter*. W algorytmie genetycznym do zmiany w pliku *main.pyx* mamy takie zmienne jak:

- **population_number** - liczba osobników na populację
- **mutation_probability** - prawdopodobieństwo mutacji
- **size_sequence** - numer wczytanego pliku z folderu **sample_cases/positive_numbers_max_k_max_sqt_max_pos/**
- **selection_percent** - procent wybrania osobników do nowej populacji
- **TIMEOUT** - czas po jakim program jest przerywany
- **number_of_average** - liczba testów wykonywana w testach
- **run_tests** - wybór czy chcemy uruchomić testy czy też raz wykonać program

Dodatkowo w pliku *log.pyx* znajdują się zmienne za włączenie debugowania działania programu.

- **DEBUG** - wyświetlanie podstawowych informacji na temat analizowanej sekwencji
- **DEBUG_GENERATION** - wyświetlanie informacji o nowych generacjach

6 Podsumowanie wyników oraz omówienie pracy działania algorytmów

Analizując najlepsze osiągnięte wyniki pomiędzy dwoma algorytmami możemy zauważyć, że algorytm genetyczny osiąga podobne czasy (jednak zawsze wolniejsze) czasy w porównaniu do algorytmu dokładnego. Wyższy czas wynika z zastosowanych większych struktur danych oraz większej potrzebnej do wykonania wstępnych operacji które rzutują na dłuższy czas wykonania. Widać jednak, że dla coraz co większych sekwencji czas wykonywania algorytmu dokładnego rośnie o wiele szybciej w porównaniu do algorytmu genetycznego. Oznacza to, że dla większych sekwencji powinniśmy zastosować algorytm genetyczny.