Problem

Dane jest klasyczne spektrum DNA jako zbiór **S class**, długość oryginalnego DNA n, długość użytych oligonukleotydów l=k (dla k>=7) oraz procentowa liczba błędów pozytywnych i negatywnych (potencjalnie obu typów).

Rozwiązanie

→ problem został rozwiązany przy pomocy algorytmu mrówkowego w języku Python

Opis kodu

Generator instancji

Przy pomocy funkcji random.choice generujemy sekwencję o zadanej długości seq_length.

Dzielenie sekwencji na oligonukleotydy

Przy pomocy funkcji split dzielimy sekwencję na oligonukleotydy. Funkcja split polega na przechodzeniu sekwencji w dwóch pętlach for, znak po znaku, dzieląc ją na k znakowe fragmenty za pomocą yield i dodaje je na listę (program bierze pod uwagę jedynie elementy o długości k). Elementy w liście są posortowane. Następnie utworzyłyśmy nową listę unique_list, do której przepisałyśmy oligonukleotydy, które się nie powtarzały oraz pojedyncze reprezentacje powtórzeń, dzięki funkcji set użytej na elementach pierwszej listy.

Błędy negatywne oraz pozytywne

BP (błąd pozytywny) i BN (błąd negatywny) są parametrami, które służą do kontrolowania poziomu błędów w generowanej sekwencji DNA przez algorytm.

BP jest parametrem, który określa liczbę losowo generowanych fragmentów, które są

dodawane do oryginalnego zbioru fragmentów. Służy to do zwiększenia liczby dostępnych fragmentów do wyboru przez algorytm mrówkowy, co prowadzi do lepszego dopasowania sekwencji do prawdziwej sekwencji.

BN jest parametrem, który określa liczbę powtórzeń fragmentów z oryginalnego zbioru fragmentów, które są dodawane do zbioru. Służy to do wprowadzenia błędów do generowanej sekwencji. Im więcej powtórzeń, tym więcej błędów będzie zawartych w generowanej sekwencji.

BP i BN służą do kontrolowania poziomu błędów w generowanej sekwencji. Wysoki poziom błędów pozytywnych i negatywnych pozwala na uzyskanie bardziej zbliżonej sekwencji do prawdziwej, ale jednocześnie zwiększa liczbę błędów w sekwencji.

Tworzenie grafu

Do tworzenia grafu zostały użyte elementy z unique_list (po dodaniu błędów negatywnych oraz pozytywnych). W naszym przypadku, graf został zdefiniowany jako słownik, gdzie każdy wierzchołek jest kluczem, a wartością jest lista sąsiednich wierzchołków i pokrycia między nimi, w następujący sposób: `graph: Dict[str, List[Tuple[str,int]]]`. Tworzymy pusty słownik o nazwie graph, który będzie przechowywał informacje o wierzchołkach i krawędziach grafu.Używamy pętli for aby przeiterować przez każdy fragment i jego pokrycie w liście fragments i coverage. Zmienna fragment przechowuje aktualnie analizowany fragment, a zmienna cov przechowuje jego pokrycie. Tworzymy pustą listę neighbors, która będzie przechowywać informacje o sąsiadach aktualnie analizowanego fragmentu.

Używamy drugiej pętli for aby przeiterować przez każdy sąsiedni fragment i jego pokrycie w liście fragments i coverage. Zmienna neighbor przechowuje aktualnie analizowany sąsiedni fragment, a zmienna neighbor_cov przechowuje jego pokrycie.

Używamy trzeciej pętli for aby przeiterować od 1 do k-1, gdzie k to długość fragmentu. Sprawdzamy czy końcowa część aktualnie analizowanego fragmentu jest taka sama jak początkowa część sąsiedniego fragmentu, używając operatora porównania ==. Jeśli tak, oznacza to, że aktualnie analizowany fragment i sąsiedni fragment mają wspólny fragment. Sprawdzamy, czy sąsiedni fragment nie został już dodany do listy neighbors używając operatora not in. Jeśli nie, oznacza to, że jest to pierwsze wystąpienie tego sąsiada dla aktualnie analizowanego fragmentu.

Obliczamy ilość powtórzeń (pokrycie) między aktualnie analizowanym fragmentem a sąsiednim fragmentem jako k-i, gdzie i jest aktualnie analizowanym indeksem pętli. Dodajemy sąsiedni fragment i jego pokrycie do listy neighbors jako tuple (neighbor, overlap). Przypisujemy listę neighbors jako wartość dla klucza fragment w słowniku graph. Pętla iteruje dalej i analizuje następny fragment i jego sąsiadów. Po zakończeniu iteracji, funkcja zwraca słownik graph, który przechowuje informacje o wierzchołkach i krawędziach grafu.

Generator losowy

Funkcja generate_path przyjmuje trzy parametry: graf, wierzchołek startowy i długość sekwencji. Tworzy pustą listę, która będzie zawierać kolejne wierzchołki w ścieżce, a następnie używa pętli for, aby iterować od 0 do długości ścieżki -1. W każdej iteracji, funkcja pobiera listę krawędzi dla aktualnie analizowanego wierzchołka z grafu, a następnie wybiera wierzchołek z największym pokryciem (wagą) i dodaje go do listy ścieżki. Tworzymy pustą listę path i dodajemy do niej wierzchołek startowy, który jest przekazywany jako parametr start. Tworzymy zmienną current_node i przypisujemy jej wartość wierzchołka startowego. Używamy pętli for, aby przeiterować od 0 do length-1 (length jest przekazywany jako parametr funkcji). Tworzymy zmienną edges i przypisujemy do niej listę krawędzi dla aktualnie analizowanego wierzchołka, która jest dostępna w słowniku graph dla klucza current_node. Tworzymy zmienną next_node i przypisujemy do niej wierzchołek z największym pokryciem (wagą), który znajduje się w liście edges. Funkcja max(edges, key=lambda x: x[1])[0] zwraca pierwszy element tuple, który jest kluczem. Dodajemy wierzchołek next_node do listy path. Przypisujemy next_node jako aktualnie analizowany wierzchołek do current_node. Pętla iteruje dalej i analizuje następny wierzchołek. Po zakończeniu iteracji, funkcja zwraca listę path, która zawiera kolejne wierzchołki w ścieżce.

Funkcja generate_seq przyjmuje trzy parametry: graf, wierzchołek startowy i długość sekwencji, którą chcemy otrzymać. Funkcja wykorzystuje funkcję generate_path, aby znaleźć najlepszą ścieżkę między wierzchołkiem startowym a końcowym. Następnie tworzy pustą sekwencję i dodaje do niej kolejne fragmenty ścieżki w taki sposób, aby otrzymać sekwencję o długości, którą chcemy otrzymać. Po zakończeniu iteracji, funkcja zwraca sekwencję. Tworzymy zmienną sequence i przypisujemy do niej pierwszy element z listy path. Używamy pętli for aby przeiterować od 1 do len(path).

Sprawdzamy czy długość sekwencji jest mniejsza niż długość sekwencji jaką chcemy otrzymać, jeśli tak to dodajemy do sekwencji fragment od końca, którego długość równa jest różnicy między długością sekwencji jaką chcemy otrzymać a długością aktualnej sekwencji. Po zakończeniu iteracji, funkcja zwraca sekwencję.

Levenstein

Funkcja levenshtein_distance przyjmuje dwa parametry: seq1 i seq2, które reprezentują dwie sekwencje dna. Funkcja korzysta z biblioteki Levenshtein i wywołuje na niej funkcję distance, przekazując jako parametry sekwencję 1 i sekwencję 2. Funkcja distance oblicza odległość Levenshteina między dwoma sekwencjami, co jest miarą podobieństwa między nimi. Wartość zwracana przez distance jest liczbą edycji, które trzeba wykonać, aby zamienić jedną sekwencję na drugą. W tym przypadku funkcja zwraca obliczoną odległość Levenshteina między sekwencją 1 i sekwencją 2

Algorytm mrówkowy

Kod tworzy słownik node_indices, który przypisuje każdemu węzłowi (node) grafu unikalny indeks. Następnie tworzony jest pheromone_matrix, który jest macierzą jedynkową o wymiarach odpowiadających liczbie węzłów. Funkcja choose_next_node jest główną funkcją algorytmu, która określa następny węzeł do odwiedzenia przez mrówkę. Przyjmuje ona jako parametry aktualny węzeł (current_node), graf, macierz feromonów, słownik node_indices, parametry alpha i beta. Funkcja ta sumuje prawdopodobieństwo przejścia do każdego z sąsiednich węzłów, a następnie losowo wybiera jeden z sąsiednich węzłów z prawdopodobieństwem proporcjonalnym do sumy. Funkcja zwraca kolejny węzeł, do którego mrówka będzie się przemieszczać.

Funkcja generate_apath generuje ścieżkę dla mrówki na podstawie grafu, aktualnego węzła, parametrów alpha i beta oraz macierzy feromonów. Funkcja ta rozpoczyna od podanego początku, a następnie wybiera kolejny węzeł do odwiedzenia przy użyciu funkcji choose_next_node i dodaje go do ścieżki, aż do momentu, gdy ścieżka

zawiera wszystkie węzły grafu.

W algorytmie mrówkowym, alpha i beta są parametrami, które kontrolują wpływ feromonów i kosztów na wybór ścieżki przez mrówki. Alpha jest parametrem, który kontroluje wpływ feromonów na decyzję mrówki. Im większa wartość alpha, tym większy wpływ ma feromon na decyzję mrówki, co oznacza, że mrówki będą bardziej skłonne do przemijania przez ścieżki z wysokim poziomem feromonów. Beta jest parametrem, który kontroluje wpływ kosztów na decyzję mrówki. Im większa wartość beta, tym większy wpływ ma koszt na decyzję mrówki, co oznacza, że mrówki będą bardziej skłonne do przemijania przez ścieżki o niskim koszcie.

Funkcja solve_dna_sequencing jest główną funkcją rozwiązującą problem. Przyjmuje ona jako parametry graf, punkt startowy, punkt końcowy, liczbę mrówek, liczbę iteracji, parametry alpha i beta, stężenie feromonów, wartość ścieżki. Funkcja ta rozpoczyna od początkowej wartości dla najlepszej ścieżki i najlepszego wyniku. Następnie przeprowadza wielokrotne iteracje, w których każda mrówka generuje ścieżkę przy użyciu funkcji generate_apath i porównuje ją z obecnym najlepszym rozwiązaniem. Kryterium jest ocena na podstawie sumy długości nakładających się fragmentów między kolejnymi węzłami ścieżki. Jeśli nowa ścieżka jest lepsza, jest ustawiana jako najlepsze rozwiązanie. Po każdej iteracji, funkcja update_pheromone_matrix jest wywoływana, aby zaktualizować macierz feromonów. W tej funkcji, dla każdego kroku w ścieżce, aktualizowane są wartości feromonów zgodnie z wzorem (1 - decay_rate) * pheromone_matrix[i][i+1] + trail_value. Na końcu, po wykonaniu odpowiedniej liczby iteracji, najlepsza ścieżka jest zwracana przez funkcję solve_dna_sequencing.

Funkcja generate_aseq jest odpowiedzialna za generowanie sekwencji DNA na podstawie grafu, punktu startowego, końcowego, liczby mrówek, liczby iteracji, parametrów alfa, beta, wartości śladu, współczynnika degradacji, macierzy feromonów oraz prawdziwej sekwencji DNA. Funkcja wykorzystuje rozwiązanie problemu sekwencjonowania DNA solve_dna_sequencing i za jej pomocą generuje najlepszą ścieżkę. Następnie przy użyciu pętli for, funkcja sprawdza nakładanie się sekwencji i generuje nową sekwencję DNA z uwzględnieniem nakładających się fragmentów. Jeśli długość generowanej sekwencji jest większa niż długość prawdziwej sekwencji, funkcja kończy swoje działanie i zwraca wygenerowaną sekwencję.

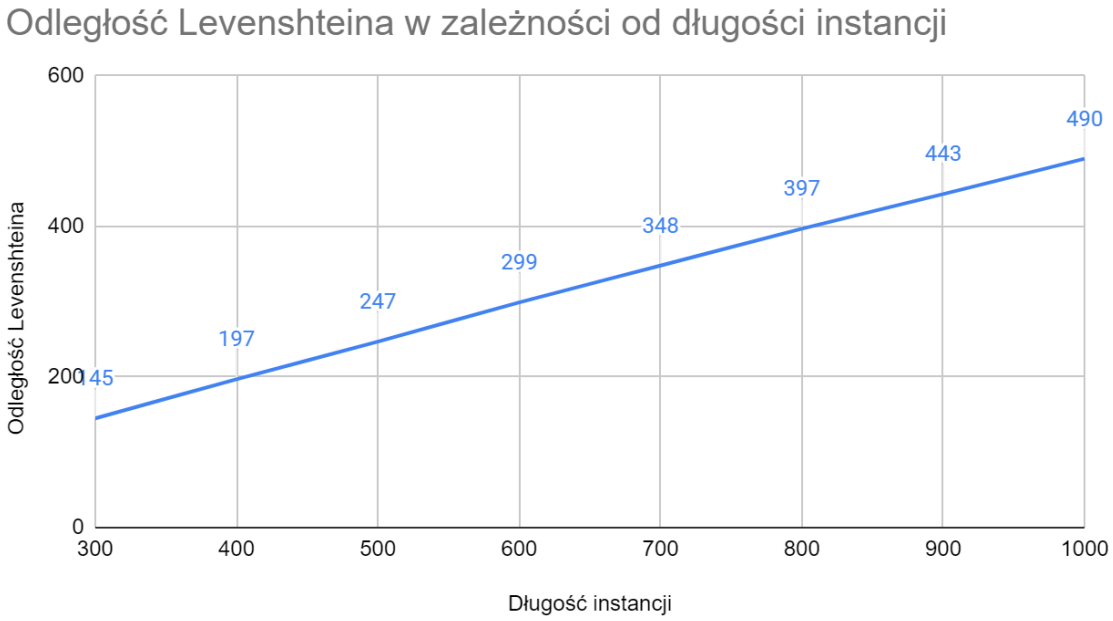
TESTY

Testy: odległość Levenshteina w zależności od długości instancji oraz czas wykonywania algorytmu w zależności od długości instancji

Przyjęte stałe parametry:

Błędy pozytywne oraz negatywne: po 3%
Długość oligonukleotydu: 7
Ilość mrówek: 20
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

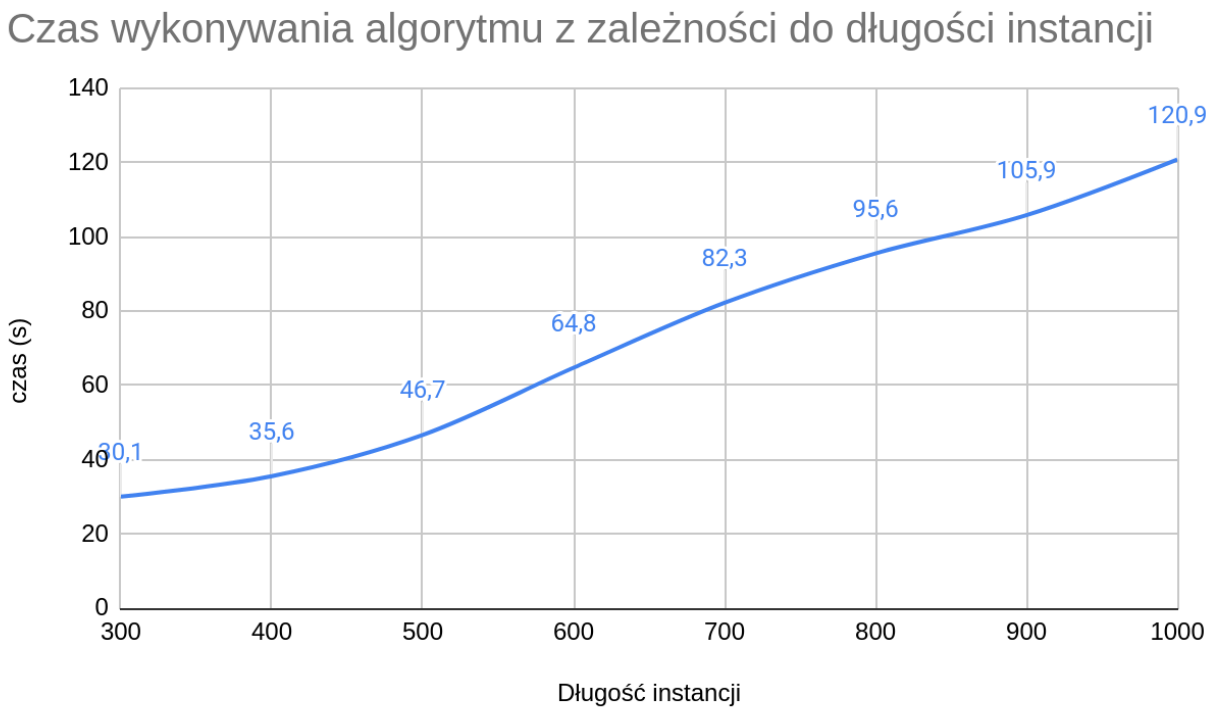
Wnioski:
Zarówno odległość Levenshteina, jak i czas, wzrastają wraz ze zwiększeniem generowanej instancji. Wzrost ten jest wprost proporcjonalny, przez co możemy uznać, że program przy zadanych stałych wartościach generuje stały stosunek odległości Levenshteina do długości sekwencji. Długość sekwencji nie ma zatem wpływu na skuteczność algorytmu.



Przyjęte stałe parametry:

Błędy pozytywne oraz negatywne: po 3%
Długość oligonukleotydu: 7
Ilość mrówek: 20
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Wnioski:
Zarówno odległość Levenshteina, jak i czas, wzrastają wraz ze zwiększeniem generowanej instancji. Wzrost ten jest wprost proporcjonalny, przez co możemy uznać, że program przy zadanych stałych wartościach generuje stały stosunek czasu do długości sekwencji. Długość sekwencji ma zatem wpływ na czas działania algorytmu.

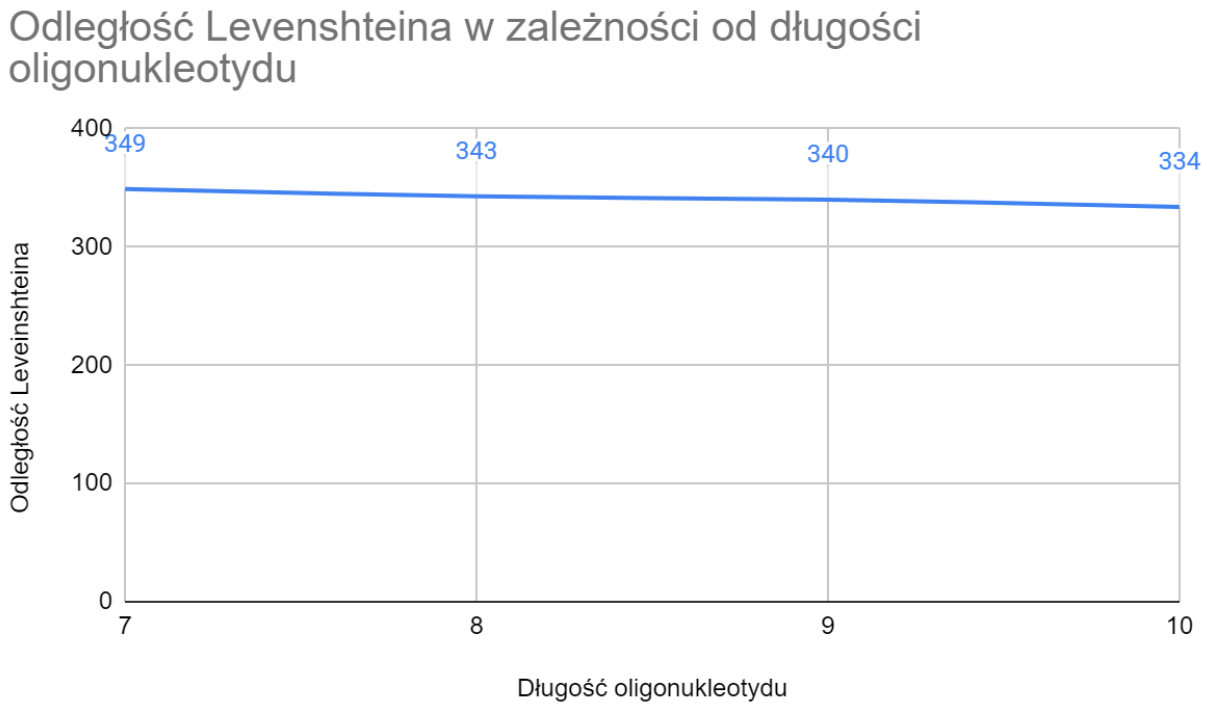


Test: odległość Levenshteina w zależności od długości oligonukleotydu

Przyjęte stałe parametry:

Błędy pozytywne oraz negatywne: po 3%
Długość oligonukleotydu: od 7 do 10
Ilość mrówek: 20
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Wnioski:
Jak możemy zaobserwować na wykresie wydłużanie długości nukleotydu ma wpływ na odległość Levenshteina. Wraz ze wzrostem długości odległość maleje, nie jest to drastyczny spadek, jednak można odczytać, że wybierając dłuższe oligonukleotydy odległość Levenshteina zmniejsza się, tym samym odtwarzana sekwencja jest dokładniejsza.



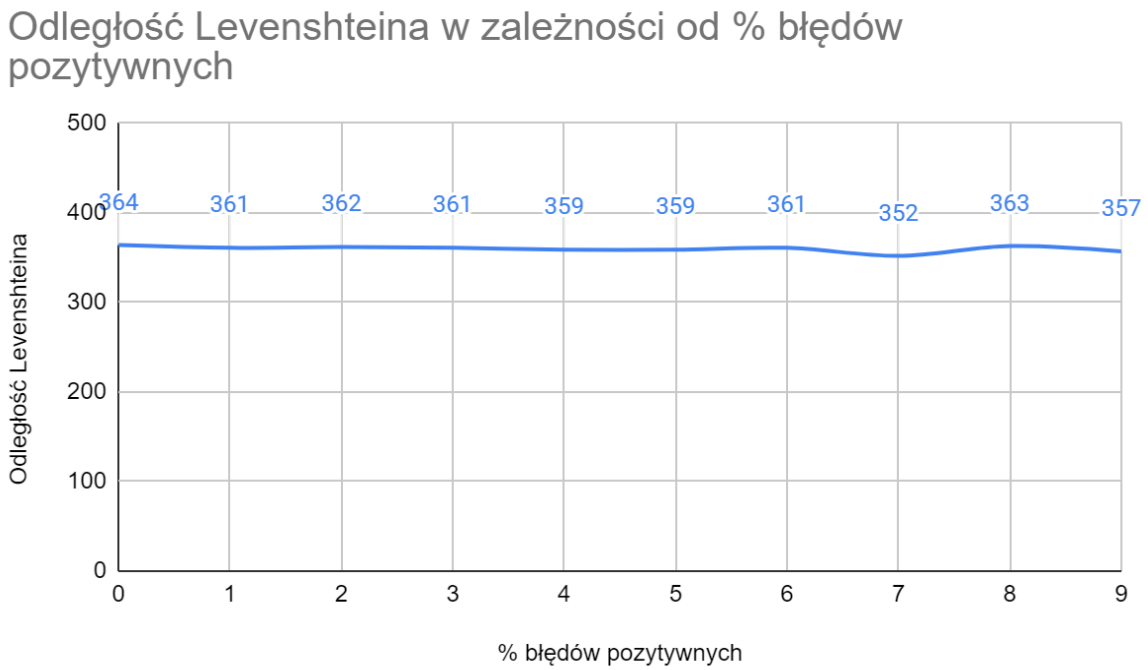
Test: odległość Levenshteina w zależności od % błędów negatywnych oraz pozytywnych (badanych oddzielnie)

Przyjęte stałe parametry:

Błędy negatywne: 0
Długość oligonukleotydu: 7
Ilość mrówek: 20
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Błędy pozytywne: 0/1/2/3/4/5/6/7/8/9%

Wnioski:
Jak możemy zaobserwować na wykresie ilość błędów pozytywnych nie ma dużego wpływu na jakość rozwiązania, uzyskane wartości niemalże generują wykres przypominający stałą.

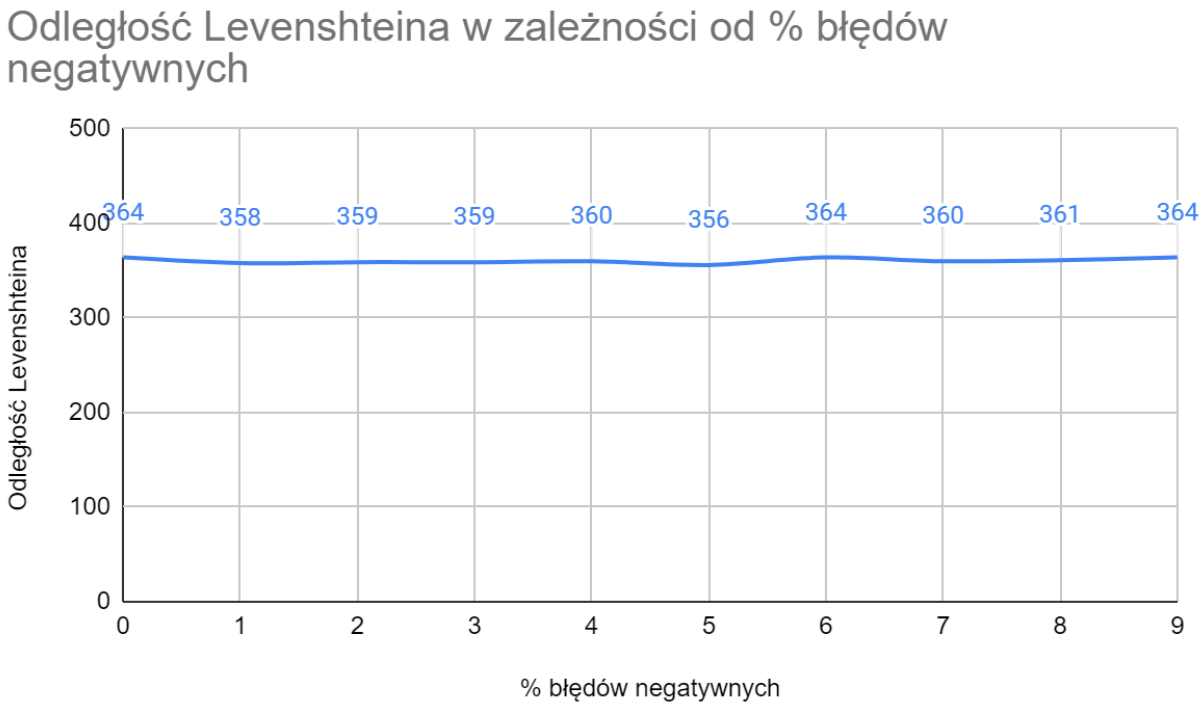


Przyjęte stałe parametry:

Błędy pozytywne: 0
Długość oligonukleotydu: 7
Ilość mrówek: 20
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Błędy negatywne: 0/1/2/3/4/5/6/7/8/9%

Wnioski:
Jak możemy zaobserwować na wykresie ilość błędów negatywnych nie ma dużego wpływu na jakość rozwiązania, uzyskane wartości niemalże generują wykres przypominający stałą.



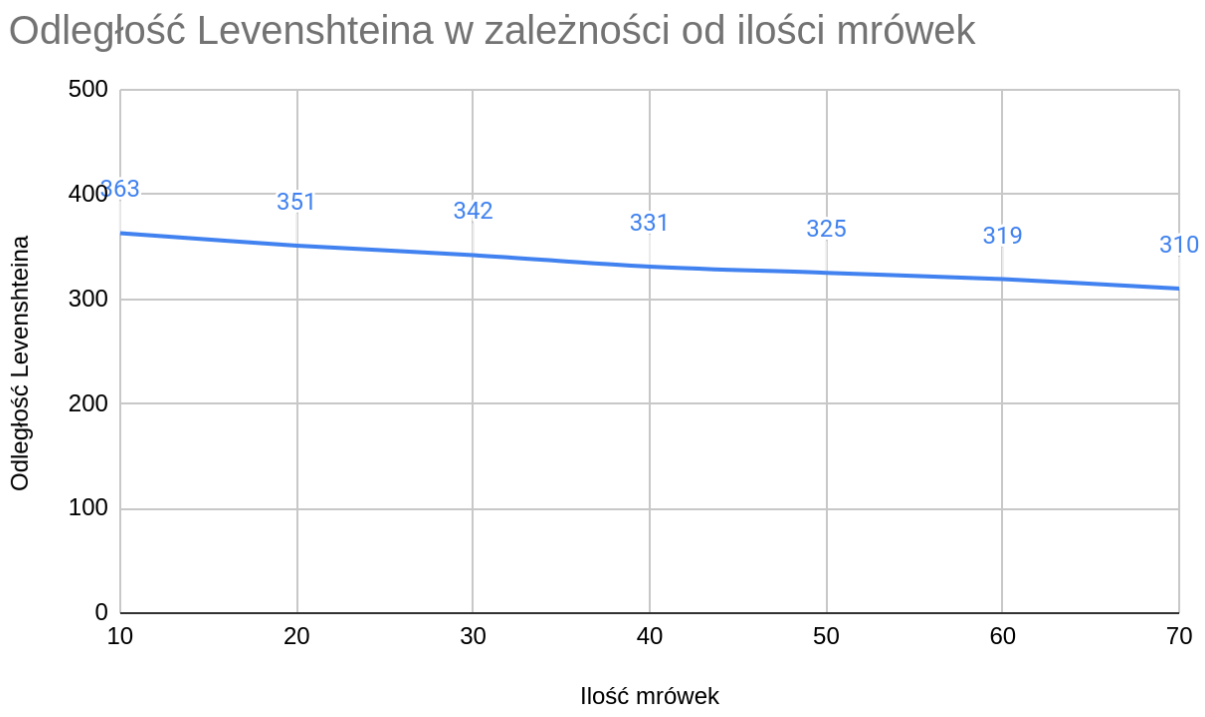
Test: odległość Levenshteina w zależności od ilości mrówek oraz test: czas wykonywania algorytmu w zależności od ilości mrówek

Przyjęte stałe parametry:

Błędy pozytywne i negatywne: po 3%
Długość oligonukleotydu: 7
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Ilość mrówek: 10/20/30/40/50/60/70

Wnioski:
Zwiększenie ilości mrówek ma wpływ na jakość tworzonego rozwiązania, jak można zobaczyć na wykresie zwiększenie ich ilości powoduje wygenerowanie rozwiązania dokładniejszego o wiele lepszej jakości.



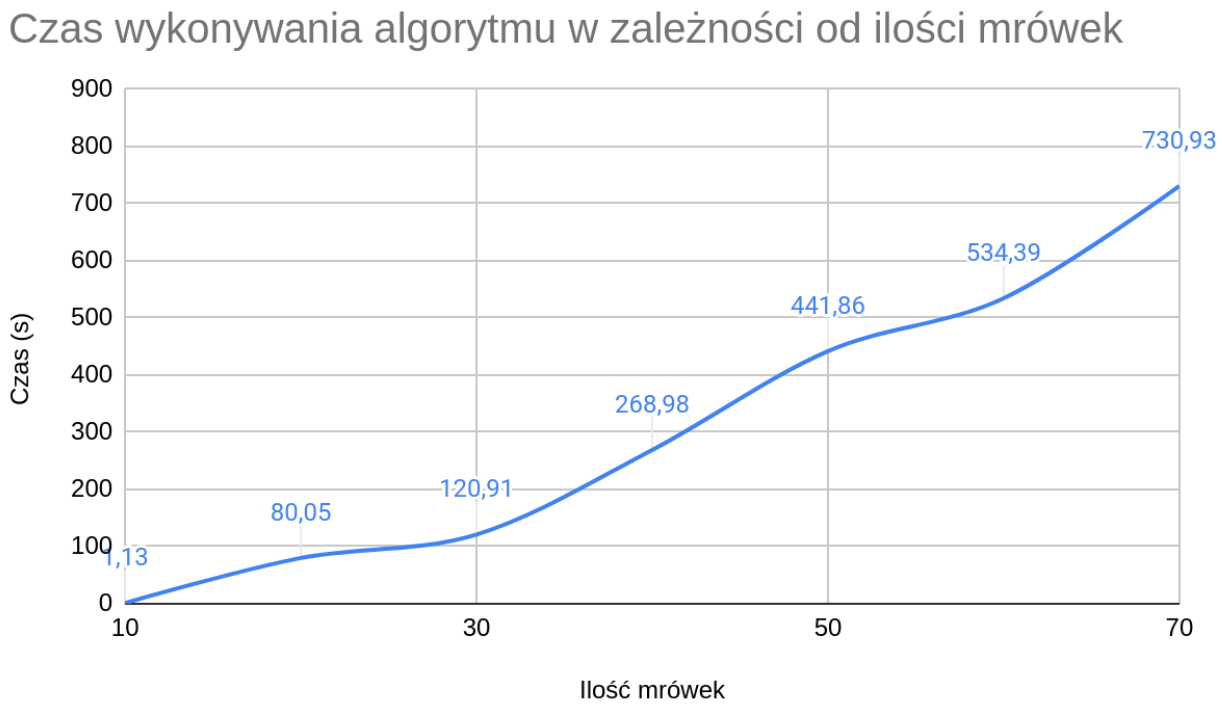
Test: odległość Levenshteina w zależności od ilości mrówek oraz test: czas wykonywania algorytmu w zależności od ilości mrówek

Przyjęte stałe parametry:

Błędy pozytywne i negatywne: po 3%
Długość oligonukleotydu: 7
Ilość kroków dla każdej mrówki: 10
Alfa: 0.5
Beta: 0.5

Ilość mrówek: 10/20/30/40/50/60/70

Wnioski:
Czas wykonywania algorytmu zwiększa się wraz z zwiększaniem ilości mrówek, wynika to z faktu, że program musi dokonać więcej przejść przez algorytm co wymaga większego nakładu pracy.

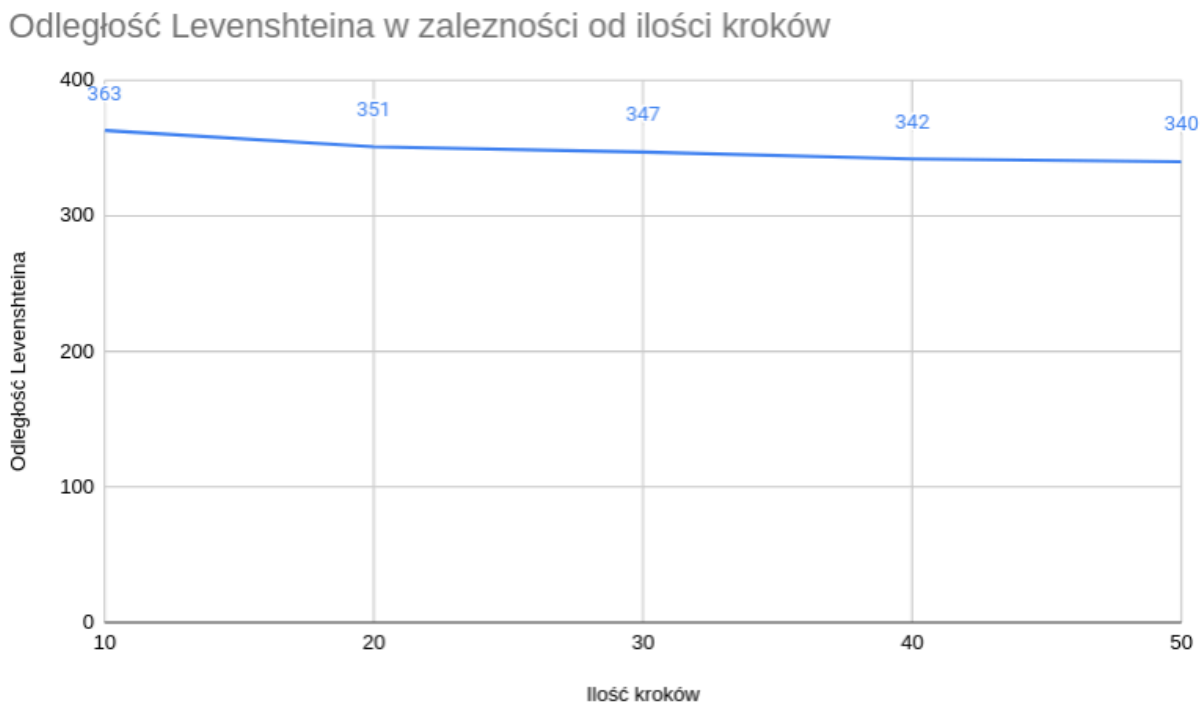


Test: odległość Levenshteina w zależności ilości kroków

Przyjęte stałe parametry:

Błędy pozytywne i negatywne: po 3%
Długość oligonukleotydu: 7
Ilość mrówek: 15
Ilość kroków dla każdej mrówki: 10/20/30/40/50
Alfa: 0.5
Beta: 0.5

Wnioski:
Zwiększenie ilości kroków wpływa na jakość rozwiązania, jak widać na wykresie dzięki większej ilości kroków generowane rozwiązanie charakteryzuje się mniejszą odległością Levenshteina czyli jest bardziej dokładne.



Wnioski ogólne

Algorytm mrówkowy jest skuteczny w rozwiązywaniu problemu sekwencjonowania DNA, ponieważ jest w stanie znaleźć optymalną ścieżkę przez graf DNA na podstawie różnych kryteriów oceny.

Algorytm mrówkowy jest dobrym rozwiązaniem dla sekwencjonowania DNA w przypadku braku informacji o prawdziwej sekwencji DNA.

W przypadku dużych grafów DNA, algorytm mrówkowy może być wolniejszy w porównaniu z innymi metodami sekwencjonowania.

W przypadku niskiej jakości danych wejściowych, algorytm mrówkowy może generować niepoprawne sekwencje DNA.

Algorytm mrówkowy jest dobrym rozwiązaniem dla problemów z wysokim poziomem niepewności, ponieważ jest w stanie generować rozwiązania z różnym poziomem jakości.

Stosowanie odpowiedniego kryterium oceny jest kluczowe dla osiągnięcia dobrych wyników.