Bioinformatyka

Semestr: 6

Projektu: Sekwencjonowanie przez hybrydyzację (SBH)

Autor: Szymon Warguła 158037

Linki:

- do kodu programów: https://github.com/Kosinir/Bioinformatyka
- do całego arkusza kalkulacyjnego: https://docs.google.com/spreadsheets/d/1j3mLXDZVZH6OMmuPjRM1Vn6ATKVlT9rWCe5lAFwuHHQ/edit?usp=sharing

1. Opis algorytmu

Algorytm rozwiązujący problem *Sequencing by Hybridization* (SBH) ma na celu rekonstrukcję oryginalnej sekwencji DNA na podstawie zbioru oligonukleotydów (k-merów) o ustalonej długości. Problem biologiczny zostaje sprowadzony do problemu grafowego, w którym wierzchołkami są k-mery, a krawędzie odpowiadają ich możliwym nakładkom. Dzięki zastosowaniu metaheurystyki kolonii mrówek (ACO), wzmacnianej lokalną optymalizacją i algorytmem Dijkstry, możliwe jest skuteczne odtworzenie sekwencji, nawet w obecności błędów.

2. Dane wejściowe

Program otwiera plik tekstowy i czyta z niego:

- **n** – długość sekwencji DNA do odtworzenia.
- **k** – długość oligonukleotydów (k-merów) w spektrum ($7 \leq k \leq 10$).
- **start_oligo** – znany początkowy k-mer (wierzchołek startowy grafu).
- **num_neg_errors** – liczba brakujących k-merów (błędy negatywne).
- **has_repeats** – flaga: czy wśród błędów są te wynikające z powtórzeń (wtedy część k-merów można odwiedzić więcej niż raz).
- **num_pos_errors** – liczba fałszywych k-merów (błędy pozytywne).
- **lista k-merów** – zbiór oligonukleotydów długości k.

3. Budowa grafu

Algorytm rozpoczyna się od przekształcenia spektrum k-merów w graf: każdy fragment długości k staje się wierzchołkiem, a między dowolnymi dwiema sekwencjami, które zachodzą na siebie przynajmniej jedną literą, wstawiana jest skierowana krawędź wyznaczona przez długość ich nakładki. Waga tej krawędzi odpowiada liczbie nowych nukleotydów, które musimy dodać do odtwarzanej sekwencji (czyli k minus długość nakładki). Do każdego istniejącego połączenia przypisywana jest początkowa wartość feromonu, co pozwala na późniejsze wzmacnianie lub osłabianie poszczególnych dróg w procesie optymalizacji.

4. Inicjalizacja

Następnie inicjalizowane są "mrówki" – każda z nich przechowuje własną trasę (kolejność odwiedzanych wierzchołków), bieżącą długość odtworzonej sekwencji, sumaryczny koszt (suma wag krawędzi) oraz licznik odwiedzin poszczególnych wierzchołków, uwzględniający limit dopuszczalnych powtórzeń węzłów wynikający z liczby błędów negatywnych i informacji o powtórzeniach. Wszystkie mrówki startują z tego samego wierzchołka początkowego, który odpowiada znanemu kawałkowi sekwencji startowej.

5. 1 etap działania algorytmu

Każda iteracja algorytmu składa się z etapu „budowy trasy" dla każdej mrówki oraz etapu aktualizacji feromonów. Podczas budowy trasy mrówka porusza się po grafie, krok po kroku dopisując do rekonstruowanej sekwencji kolejne fragmenty. W pierwszej kolejności próbuje zawsze wykorzystać łuki o wadze 1 (najczęściej, zależności od parametru Beta) (czyli maksymalne nakładanie się fragmentów), najpierw preferując te, które prowadzą do węzłów jeszcze nieodwiedzonych („świeże"), a jeśli takich zabraknie, akceptując wejście do węzła już użytego, o ile nie przekracza on wyznaczonego limitu powtórzeń. O ile jeszcze możliwe, porusza się po takich maksymalnych nakładkach aż do momentu, gdy żaden z węzłów docelowych nie spełnia tych kryteriów. Wtedy w obrębie łuków o malejącej nakładce (1, 2, 3, …, k–1) ponownie wyszukuje najpierw węzły świeże, a następnie dopuszcza re-usage, aż znajdzie jakąkolwiek krawędź prowadzącą dalej.

6. Etap 2 działania algorytmu (tzw. jak algorytm radzi sobie z pułapkami)

Gdy i to zawiedzie – czyli mrówka utknie w węźle, z którego nie wychodzą żadne dopuszczalne krawędzie o nakładce ≥ 1 – używany jest algorytm Dijkstry: z bieżącego węzła budowana jest najkrótsza ścieżka do kilku wybranych kandydatów, spośród tych jeszcze nieodwiedzonych lub dopuszczalnych do ponowienia, wybierając tę o najmniejszym koszcie sumarycznym. Znaleziona podścieżka do wybranego węzła jest w całości doklejona do trasy mrówki, co pozwala przełamać lokalne zastoje i wykorzystać krawędzie o większej wadze do przesunięcia się w nowe obszary grafu. Jeśli i ta metoda nie doprowadzi do wznowienia postępu (nie ma żadnych dopuszczalnych celów),

wreszcie wykonywany jest tzw. „skok wirtualny” – przejście do dowolnego węzła nie przekraczającego limitu powtórzeń, traktowane jak nakładka zerowa, co pozwala uniknąć zapętlenia lub przedwczesnego zakończenia budowy sekwencji.

7. Etap 3 działania algorytmu - optymalizacja

Budowa trasy dla danej mrówki trwa tak długo, aż długość odtworzonej sekwencji osiągnie docelowe n nukleotydów (liczone jako początkowe k liter plus suma wag kolejnych przejść) lub osiągnięty zostanie limit kroków równy liczbie wszystkich k-merów. Na koniec, jeżeli trasa faktycznie osiągnęła długość co najmniej n, stosowana jest lokalna optymalizacja typu 2-opt, w której losowo wybrane fragmenty trasy są odwracane i sprawdzane pod kątem tego, czy mimo zmienionej kolejności krawędzie nadal istnieją (nakładka ≥ 1) i czy łączny koszt spada. Jeśli tak, trasa zostaje zastąpiona lepszą wersją.

8. Finalizacja

Po zakończeniu budowy tras wszystkich mrówek obliczana jest statystyka ukończonych rozwiązań, a feromony na wszystkich krawędziach częściowo odparowują (mnożenie przez 1–ρ). Następnie każda mrówka, która odtworzyła pełną sekwencję, wnosi wkład w postaci deponowania feromonów proporcjonalnie do odwrotności całkowitego kosztu trasy – im lepsza (niższa suma wag), tym więcej feromonu. Dodatkowo stosowany jest depozyt elitarystyczny z najlepszej w dotychczasowej historii trasy, co pozwala przyspieszyć konwergencję. Cały cykl budowy tras i aktualizacji feromonów powtarza się wielokrotnie, aż zostanie przekroczony limit czasu lub liczba iteracji (w testach jest limit czasu co ma swoje wady i zalety ale o tym później).

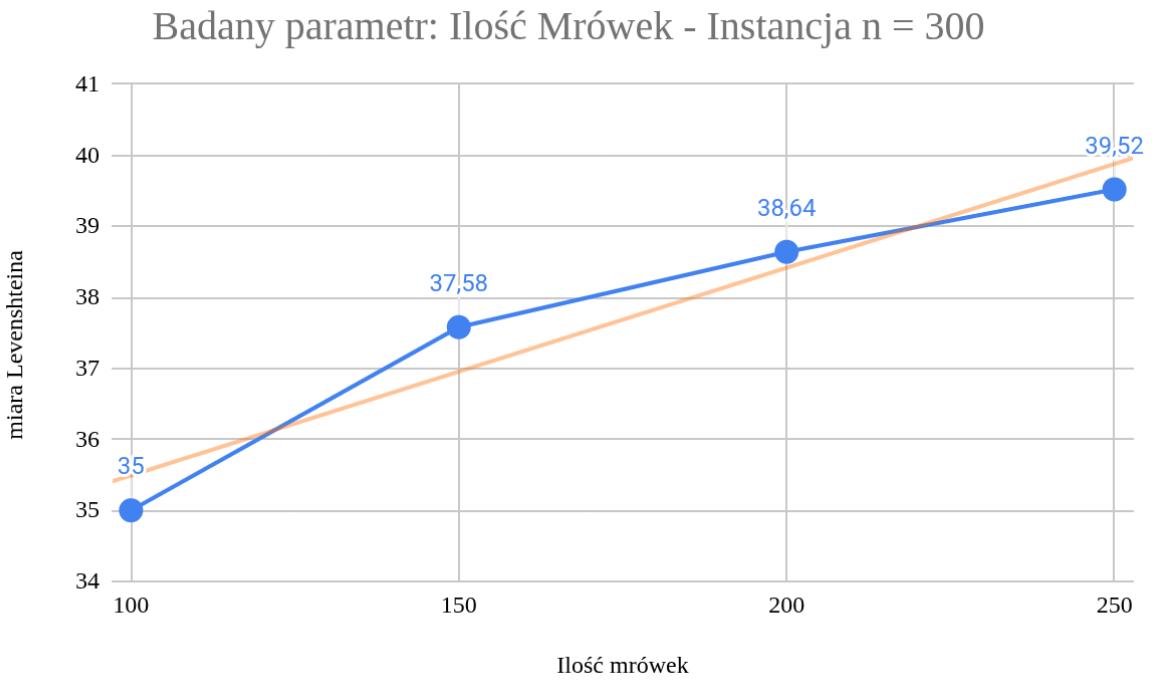
Dzięki tej wieloetapowej strategii łączącej deterministyczne przechodzenie po maksymalnych nakładkach, globalne poszukiwanie za pomocą Dijkstry, agresywne przełamywanie zastoju „skokami wirtualnymi”, heurystykę ACO oraz lokalne poprawki 2-opt, algorytm radzi sobie zarówno z idealnymi zestawami k-merów, jak i z instancjami zawierającymi zarówno braki fragmentów, jak i fałszywe dodatki, przybliżając w praktyce oryginalną sekwencję DNA.

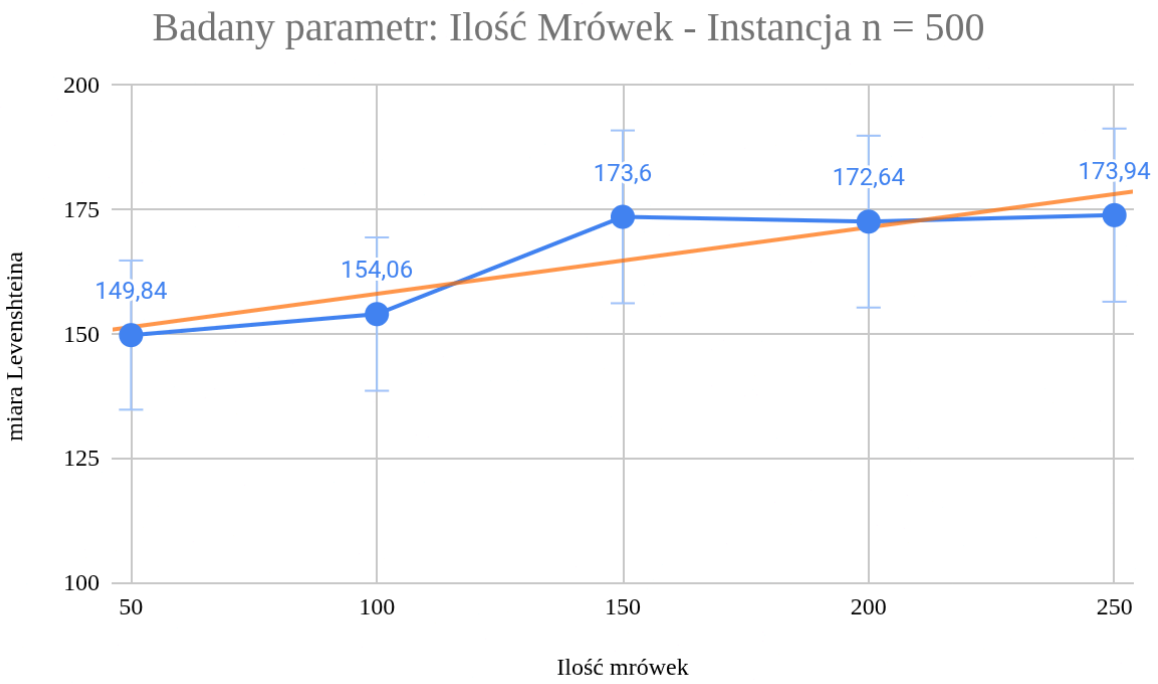
Testy i pomiary algorytmu

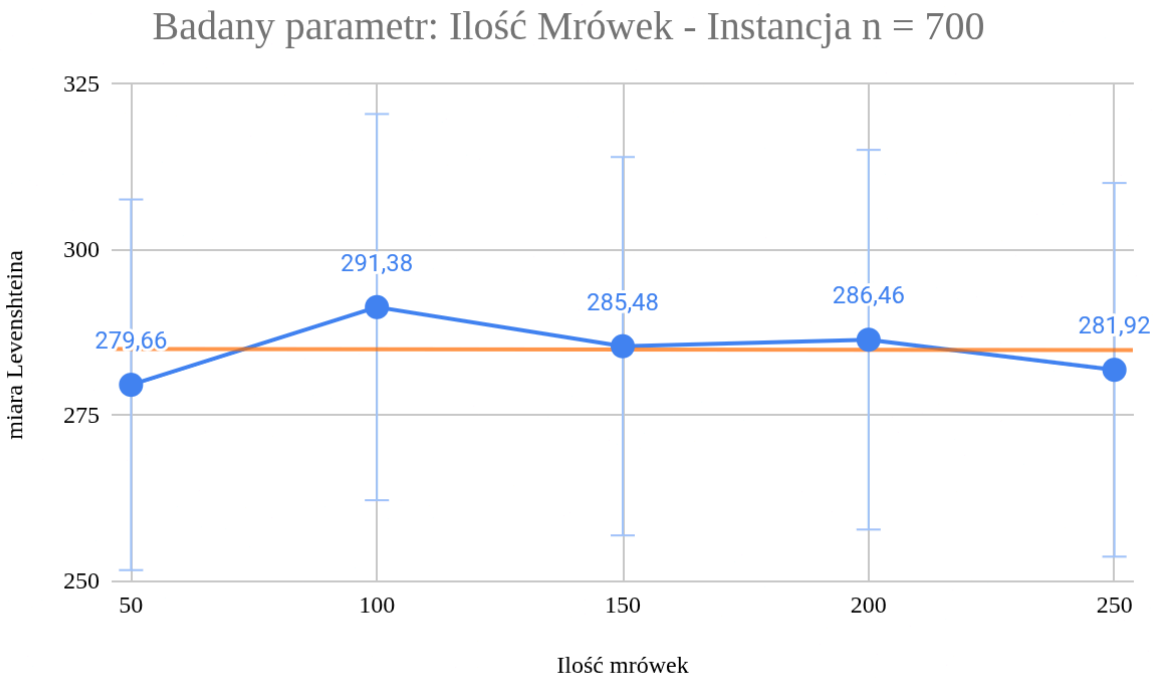
Mamy 2 rodzaje testów: Testy parametrów algorytmu i Testy parametrów instancji.
Wszystkie pomiary (danego n) to wyniki średnich 50 instancji.
Oś Y reprezentuje miarę Levenshteina, a oś X badany parametr.
Każdy test jest dla instancji n = [300, 500, 700] czyli małe, średnie i duże instancje.
Linia trendu - pomarańczowa linia.

Testowanie parametry algorytmu

Badany parametr: Ilość Mrówek (ants)						
Instancja	Input	Average Levenshtein by number of ants	Ants	300	500	700
				Levenstein		
n = 300, k = 8, num_neg = 30 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, time = 10 (s)	avg Levenshtein = 35.78 avg Levenshtein = 35.00 avg Levenshtein = 37.58 avg Levenshtein = 38.64 avg Levenshtein = 39.52	50	35,78	149,84	279,66
n = 500, k = 8, num_neg = 50 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, time = 10 (s)	avg Levenshtein = 149.84 avg Levenshtein = 154.06 avg Levenshtein = 173.60 avg Levenshtein = 172.64 avg Levenshtein = 173.94	100	35	154,06	291,38
			150	37,58	173,6	285,48
n = 700, k = 8, num_neg = 70 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, time = 10 (s)	avg Levenshtein = 279.66 avg Levenshtein = 291.38 avg Levenshtein = 285.48 avg Levenshtein = 286.46 avg Levenshtein = 281.92	200	38,64	172,64	286,46
			250	39,52	173,94	281,92


Badany parametr: Ilość Mrówek - Instancja n = 300
41
40
39
38
37
36
35
34
miara Levenshteina
35
37,58
38,64
39,52
100
150
200
250
Ilość mrówek

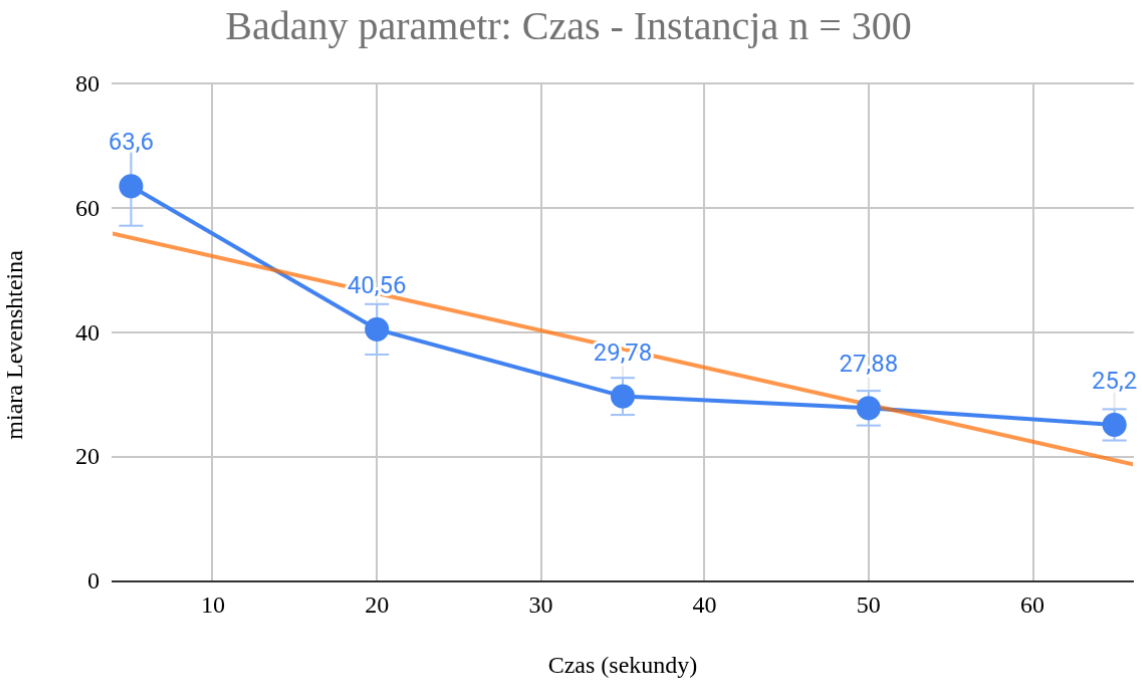

Badany parametr: Ilość Mrówek - Instancja n = 500
200
175
150
125
100
miara Levenshteina
149,84
154,06
173,6
172,64
173,94
50
100
150
200
250
Ilość mrówek

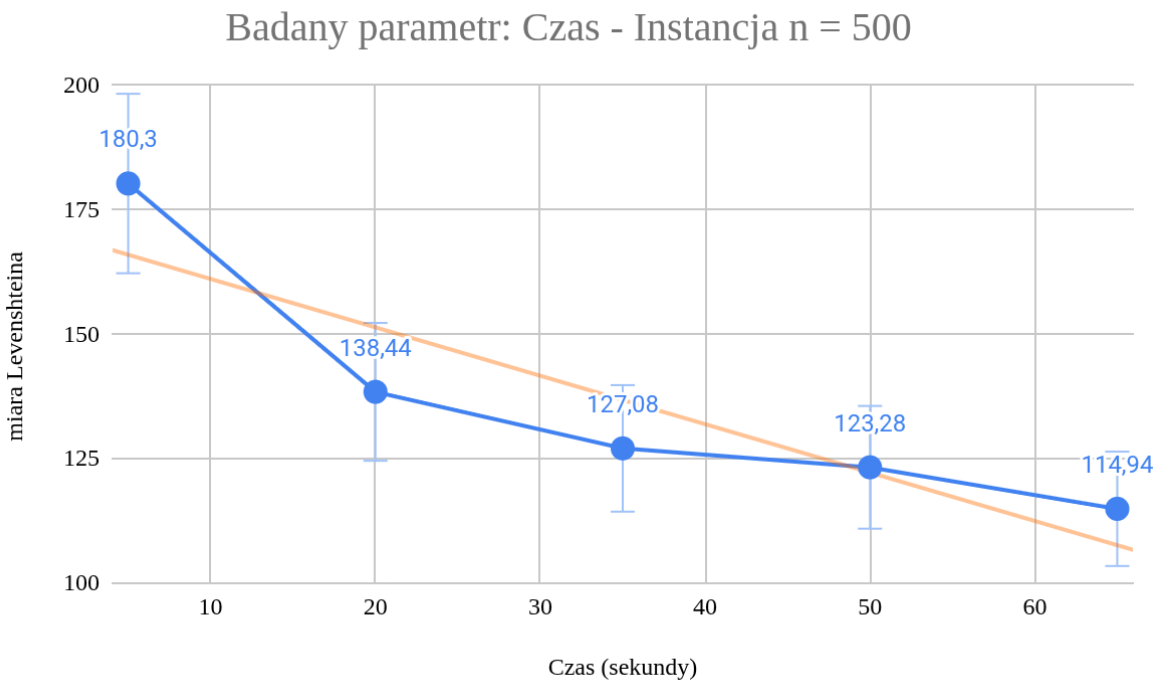


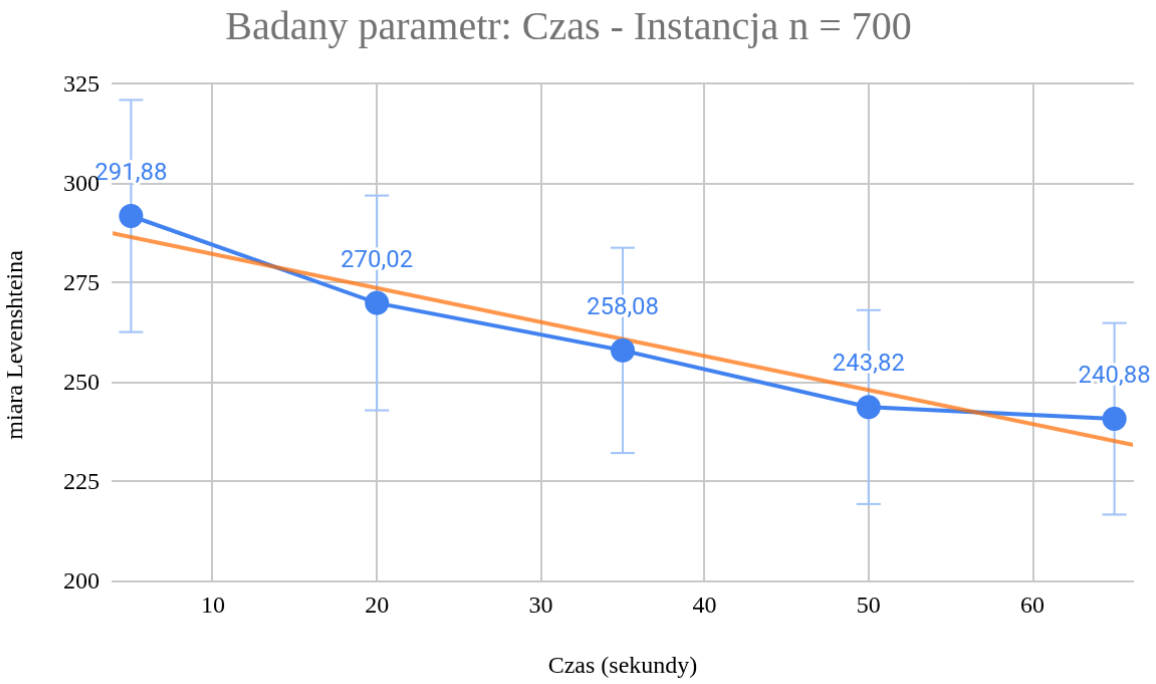
Wnioski i obserwacje:

Dla wszystkich instancji optymalna liczba mrówke to 50, im mniejsza liczba mrówek tym więcej iteracji może wykonać algorytm w ciągu 10 s. (co jest dość małym czasem) ale dla n = 700 można zauważyć że najmniejszy wyniki ma dla ants = 50 i podobny wynik dla ants = 250.
Dla ants = 50, algorytm w ciągu 10 sekund może wykonać najwięcej iteracji (prawdopodobnie do 3 iteracji), w pozostałych przypadkach tylko 1 iteracje (wina sprzętu, algorytmu i parametrów), ale dla tej 1 iteracji, 250 mrówek jest w stanie osiągnąć podobne wyniki co dla 50 mrówek.

Badany parametr: Czas (time)						
Instancja	Input	Output	Czas	300	500	700
				Levenstein		
n = 300, k = 8, num_neg = 30 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100	Time=5: avg Levenshtein = 63.60 Time=20: avg Levenshtein = 40.56 Time=35: avg Levenshtein = 29.78 Time=50s: avg Levenshtein = 27.88 Time=65s: avg Levenshtein = 25.20	5	63,6	180,3	291,88
n = 500, k = 8, num_neg = 50 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100	Time=5: avg Levenshtein = 180.30 Time=20: avg Levenshtein = 138.44 Time=35: avg Levenshtein = 127.08 Time=50: avg Levenshtein = 123.28 Time=60: avg Levenshtein = 114.94	20	40,56	138,44	270,02
			35	29,78	127,08	258,08
n = 700, k = 8, num_neg = 70 (10%), repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100	Time=5s: avg Levenshtein = 291.88 Time=20s: avg Levenshtein = 270.02 Time=35s: avg Levenshtein = 258.08 Time=50s: avg Levenshtein = 243.82 Time=65s: avg Levenshtein = 240.88	50	27,88	123,28	243,82
			65	25,2	114,94	240,88


Badany parametr: Czas - Instancja n = 300
miara Levenshteina
80
60
40
20
0
63,6
40,56
29,78
27,88
25,2
10
20
30
40
50
60
Czas (sekundy)

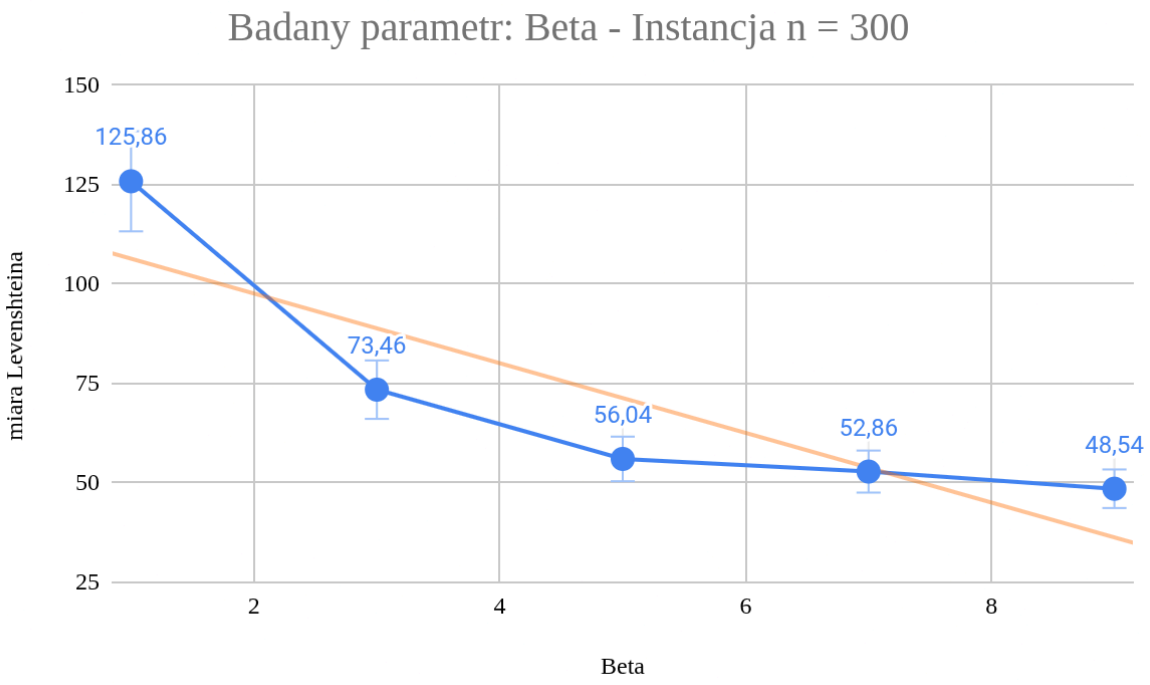

Badany parametr: Czas - Instancja n = 500
miara Levenshteina
200
175
150
125
100
180,3
138,44
127,08
123,28
114,94
10
20
30
40
50
60
Czas (sekundy)

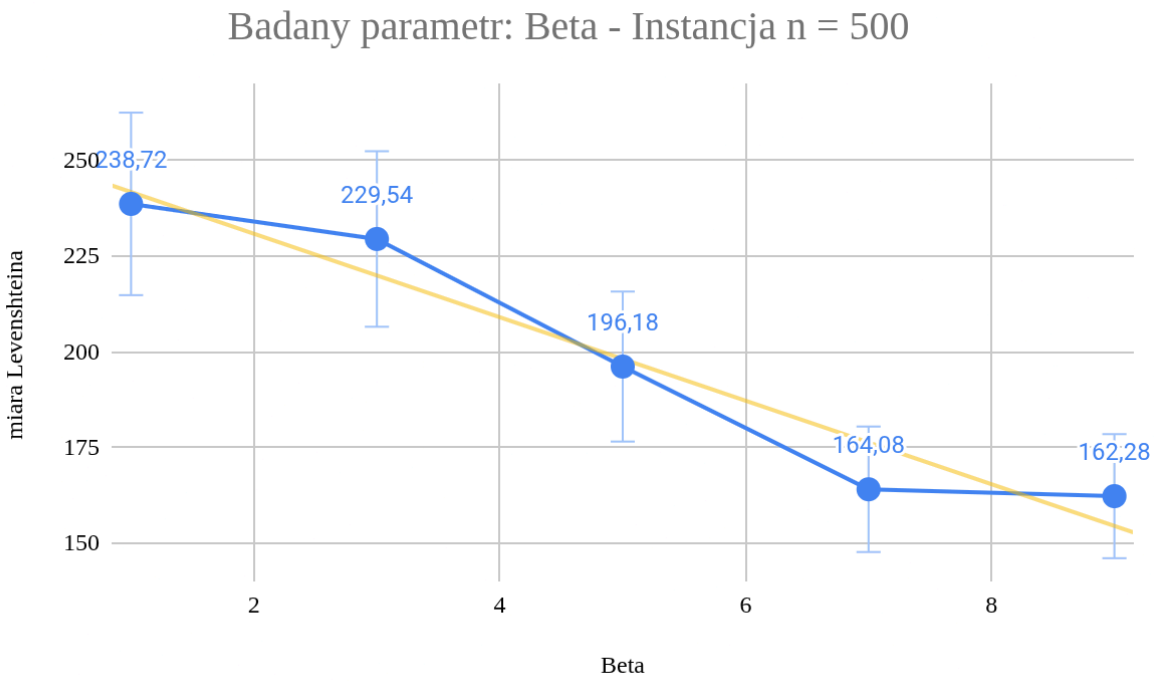


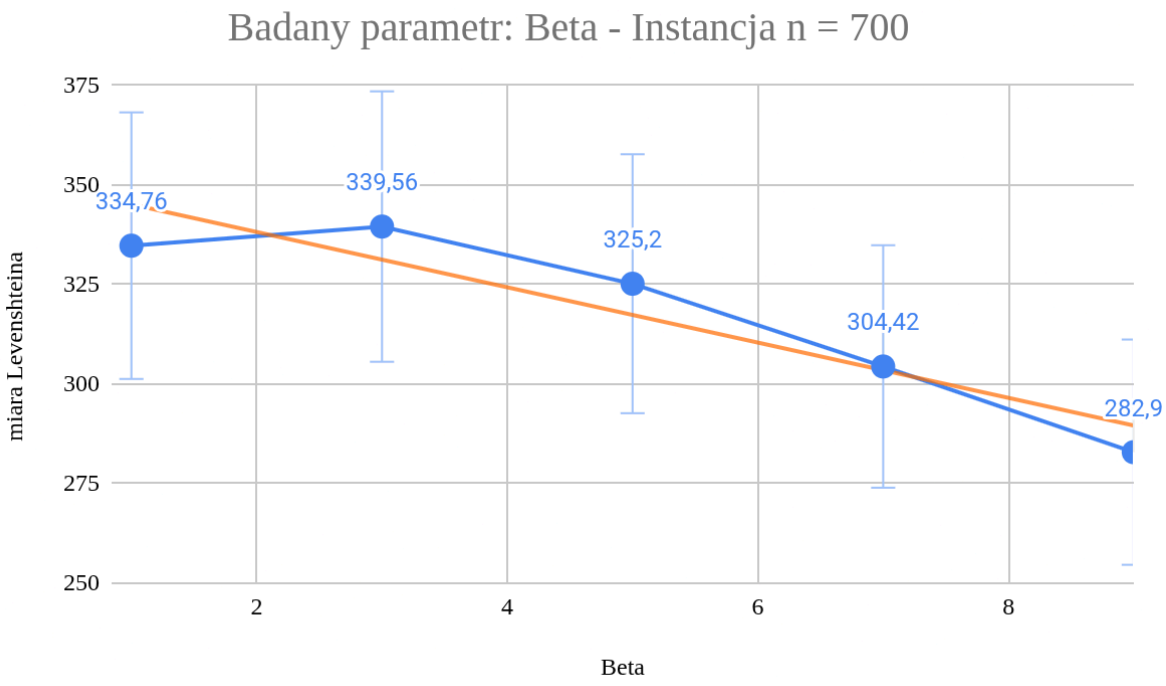
Wnioski i obserwacje:

Wniosek prosty i oczywisty - im więcej czasu tym więcej iteracji, tym lepsze wyniki, gdzie pomiędzy 50 a 65 sekund algorytm znajduje optimum lokalne.

Badany parametr: Beta						
Instancja	Input	Output	Beta	300	500	700
				Levenstein		
n = 300, k = 8, num_neg = 30 (10%), repeat = True, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 125.86 avg Levenshtein = 73.46 avg Levenshtein = 56.04 avg Levenshtein = 52.86 avg Levenshtein = 48.54	1	125,86	238,72	334,76
n = 500, k = 8, num_neg = 50 (10%), repeat = True, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 238.72 avg Levenshtein = 229.54 avg Levenshtein = 196.18 avg Levenshtein = 164.08 avg Levenshtein = 162.28	3	73,46	229,54	339,56
			5	56,04	196,18	325,2
n = 700, k = 8, num_neg = 70 (10%), repeat = True, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 343.76 avg Levenshtein = 339.56 avg Levenshtein = 325.20 avg Levenshtein = 304.42 avg Levenshtein = 282.90	7	52,86	164,08	304,42
			9	48,54	162,28	282,9





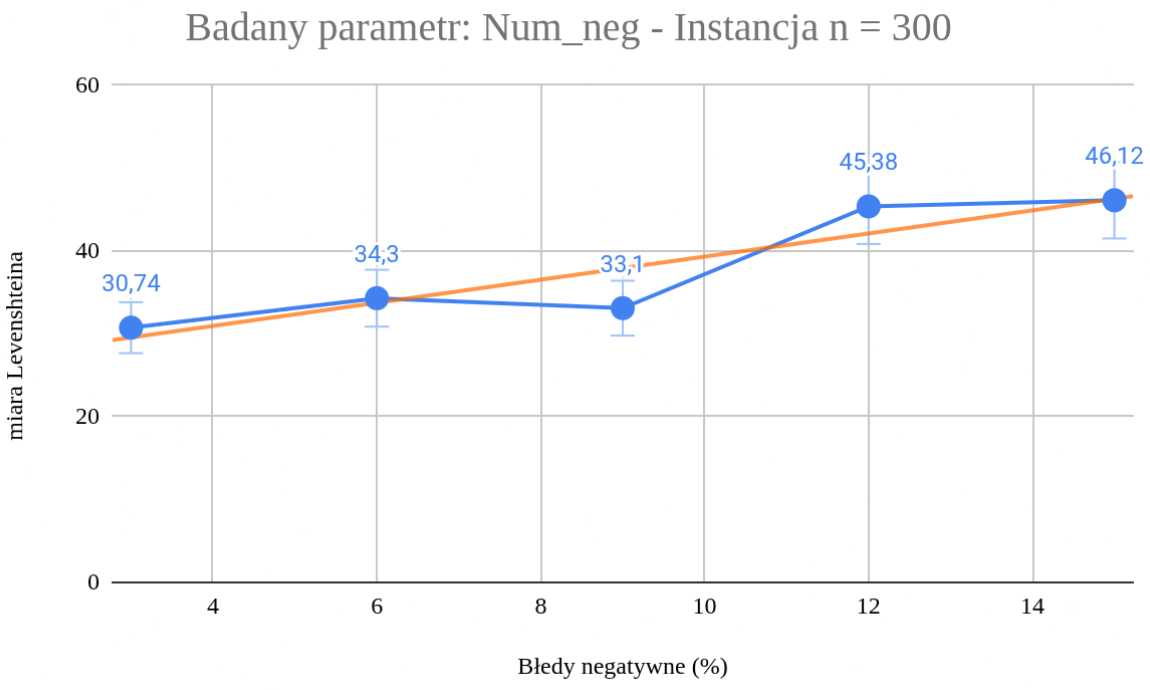


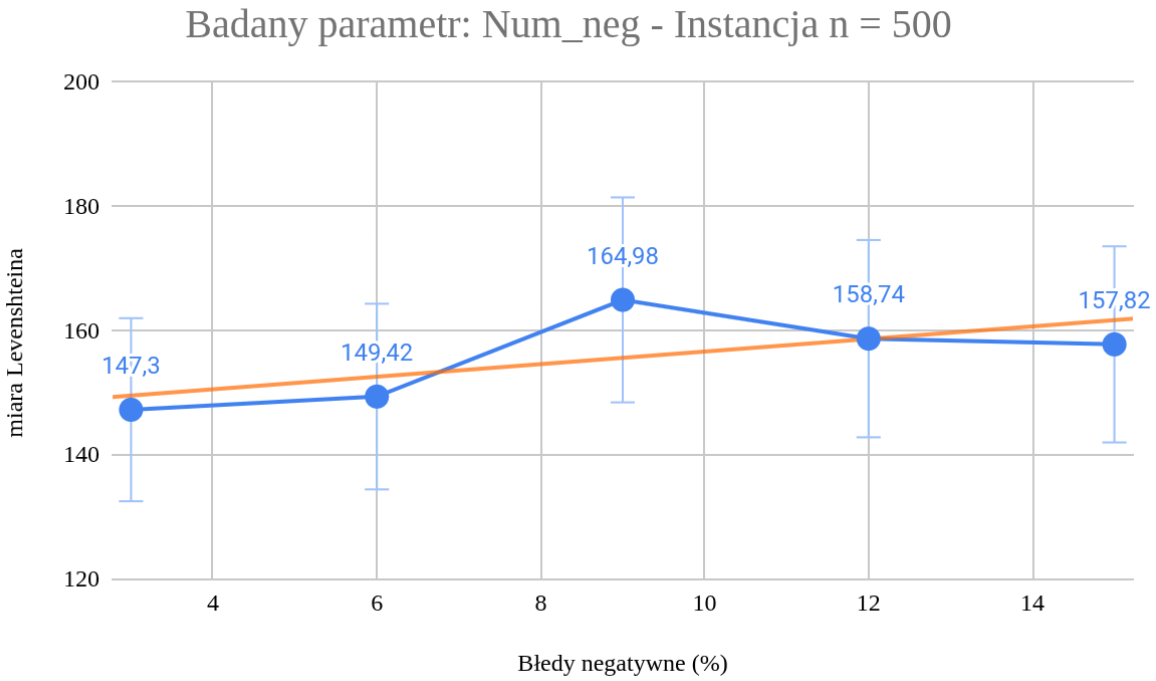
Wnioski i obserwacje:

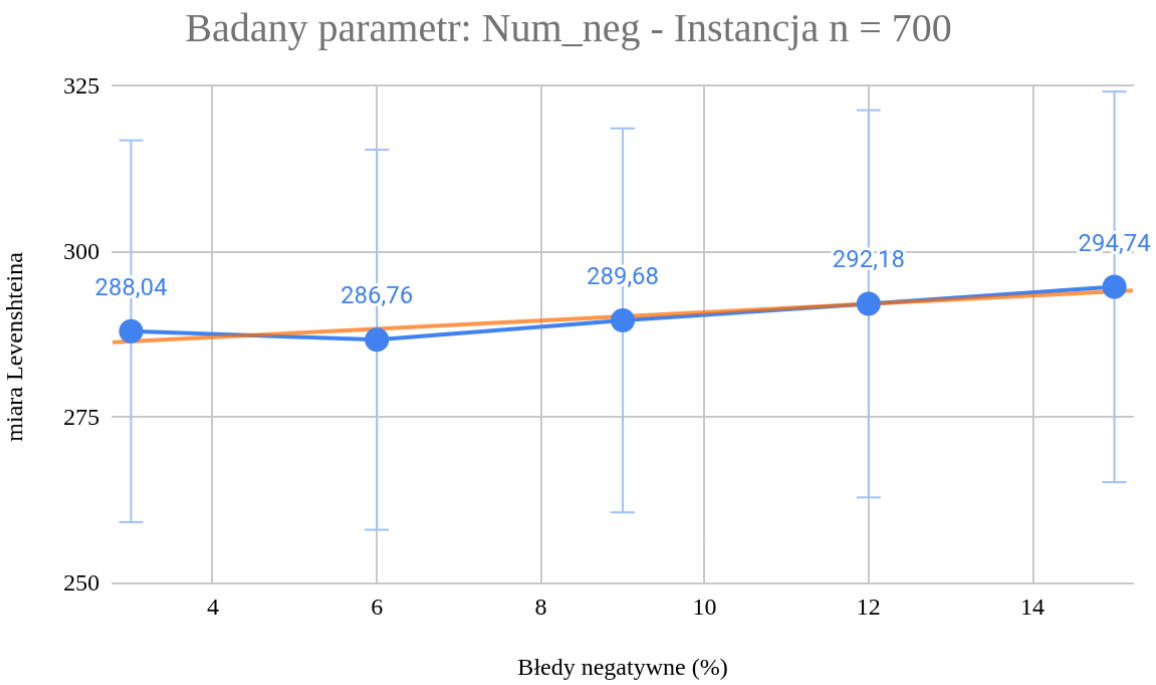
Tutaj również, im większy parametr (Beta) tym lepsze wyniki - dla małych wartości Beta, mrówki są bardziej skłonne do eksploracji najgorszych wierzchołków, ale im większa Beta tym mniej je eksplorują (te najgorsze) i jak widzimy dla Beta = 9 otrzymujemy najlepsze wyniki bo mrówki częściej wybierają wierzchołki o wadze 1 - jest to najbardziej widoczne dla instancji n = 700 bo tutaj algorytm jest w stanie 10 sekund wykonać 1-2 iteracji, gdzie dla n = [300, 500] szybko znajduje optimum lokalne.

Testowanie parametrów Instancji

Badany Parametr: błędy negatywne						
Instancja	Input	Output	num_neg (%)	300	500	700
n = 300, k = 8, repeat = True, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 30.74 avg Levenshtein = 34.30 avg Levenshtein = 33.10 avg Levenshtein = 45.38 avg Levenshtein = 46.12		Levenstein		
n = 500, k = 8, repeat = True, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 147.30 avg Levenshtein = 149.42 avg Levenshtein = 164.98 avg Levenshtein = 158.74 avg Levenshtein = 157.82	3	30,74	147,3	288,04
			6	34,3	149,42	286,76
n = 700, k = 8, repeat = True, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10(s)	avg Levenshtein = 288.04 avg Levenshtein = 286.76 avg Levenshtein = 289.68 avg Levenshtein = 292.18 avg Levenshtein = 294.72	9	33,1	164,98	289,68
			12	45,38	158,74	292,18
			15	46,12	157,82	294,74



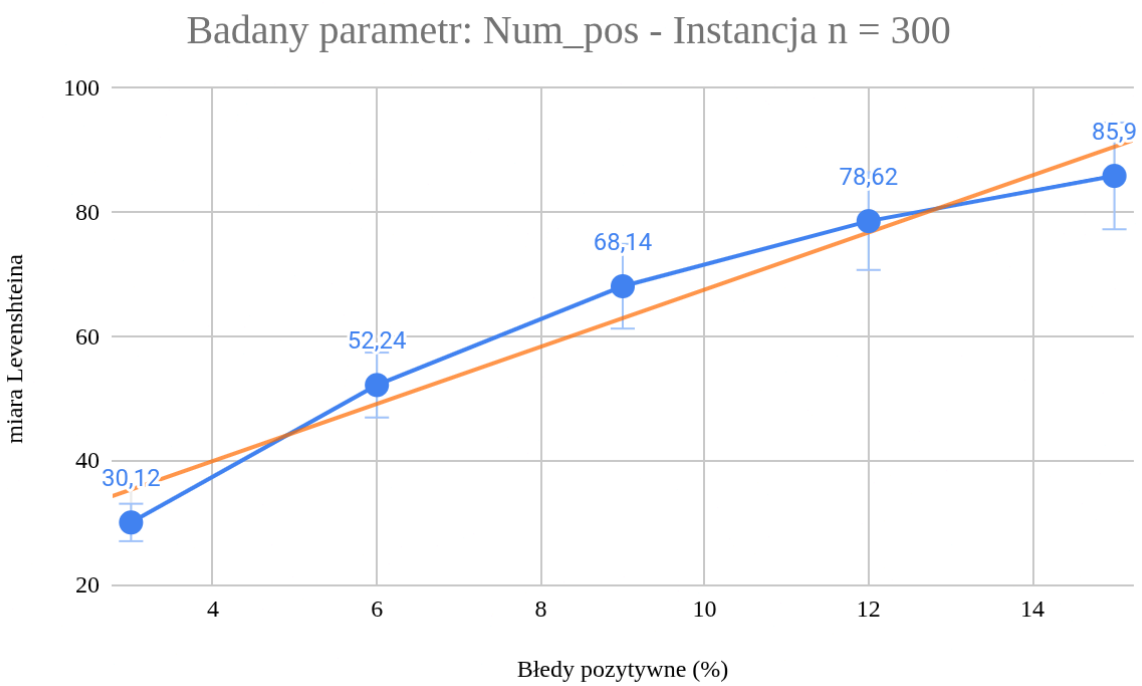


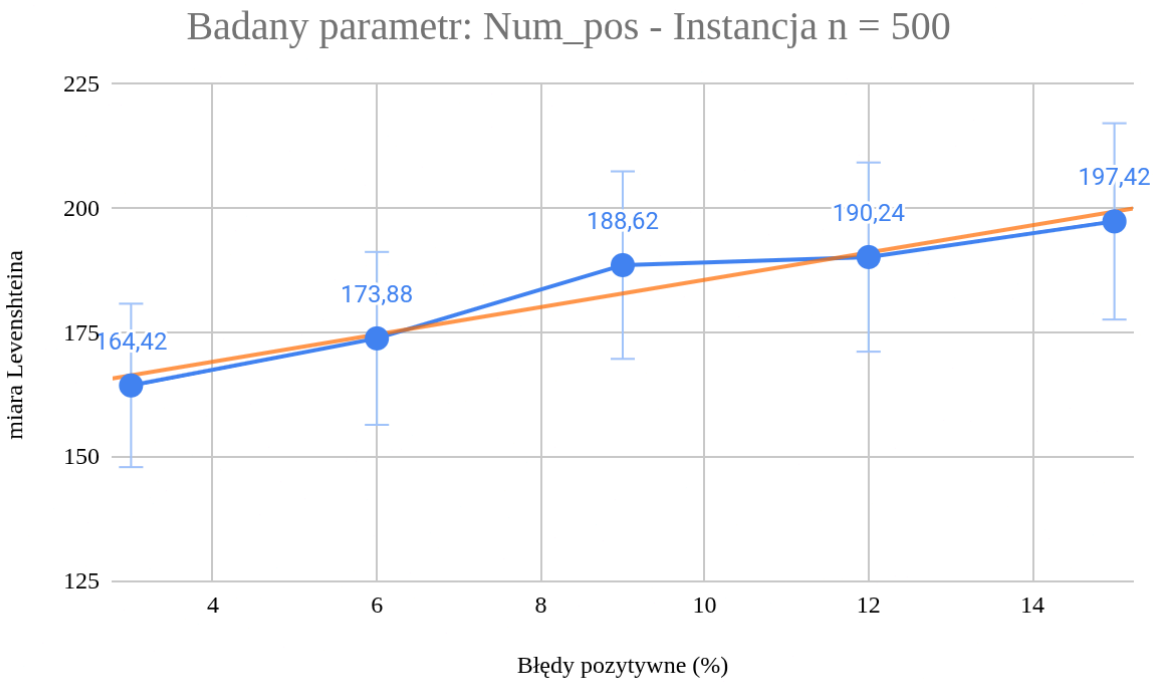


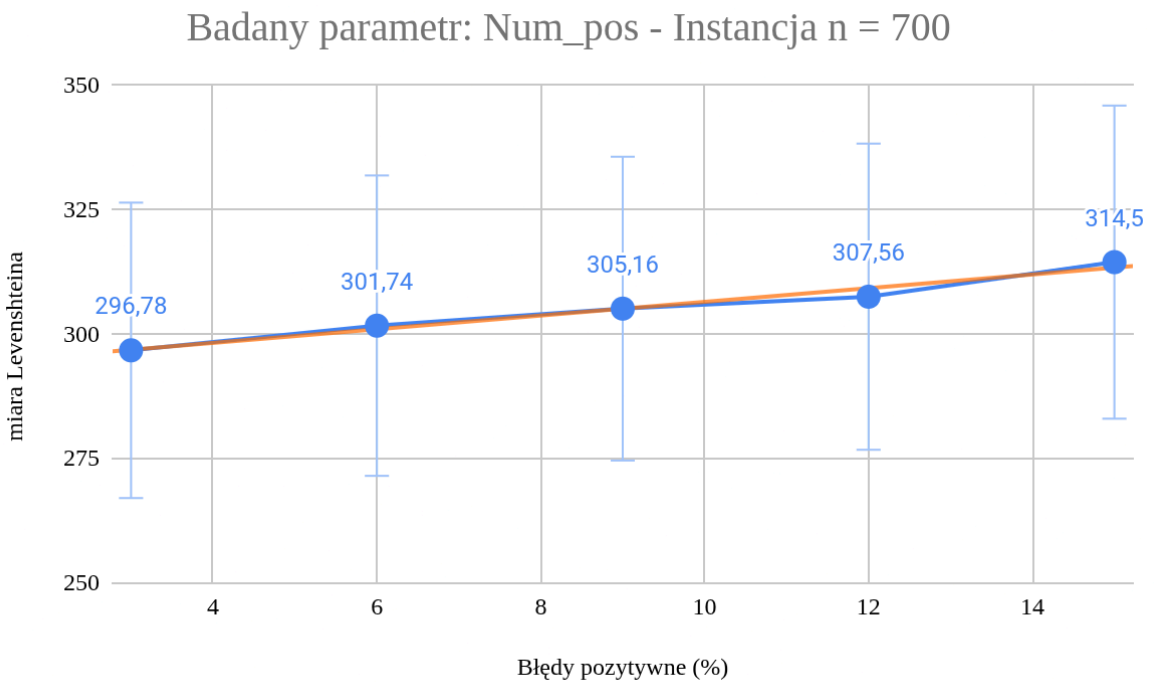
Wnioski i obserwacje:

Im więcej błędów w instacji tym gorsze wyniki - co jest oczywiste ale dlaczego wyniki podobne? Prawdopodobnie wina za wysokiego parametru Beta (= 8) który sprawia że mrówki bardzo silnie preferują najkrótsze przejścia - najbardziej oczywiste ścieżki w grafie przez co poruszają się po podobnych trasach - co oznacza że te błędy są w pewnym stopniu “ignorowane” bo mrówki trzymają się lokalnie najlepszych przejść.

Badany Parametr: błędy pozytywne						
Instancja	Input	Output	num_pos (%)	300	500	700
n = 300, k = 8, repeat = True, num_neg = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 30.12 avg Levenshtein = 52.24 avg Levenshtein = 68.14 avg Levenshtein = 78.62 avg Levenshtein = 85.90		Levenstein		
n = 500, k = 8, repeat = True, num_neg = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 164.42 avg Levenshtein = 173.88 avg Levenshtein = 188.62 avg Levenshtein = 190.24 avg Levenshtein = 197.42	3	30,12	164,42	296,78
			6	52,24	173,88	301,74
n = 700, k = 8, repeat = True, num_neg = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10(s)	avg Levenshtein = 296.78 avg Levenshtein = 301.74 avg Levenshtein = 305.16 avg Levenshtein = 307.56 avg Levenshtein = 314.50	9	68,14	188,62	305,16
			12	78,62	190,24	307,56
			15	85,9	197,42	314,5



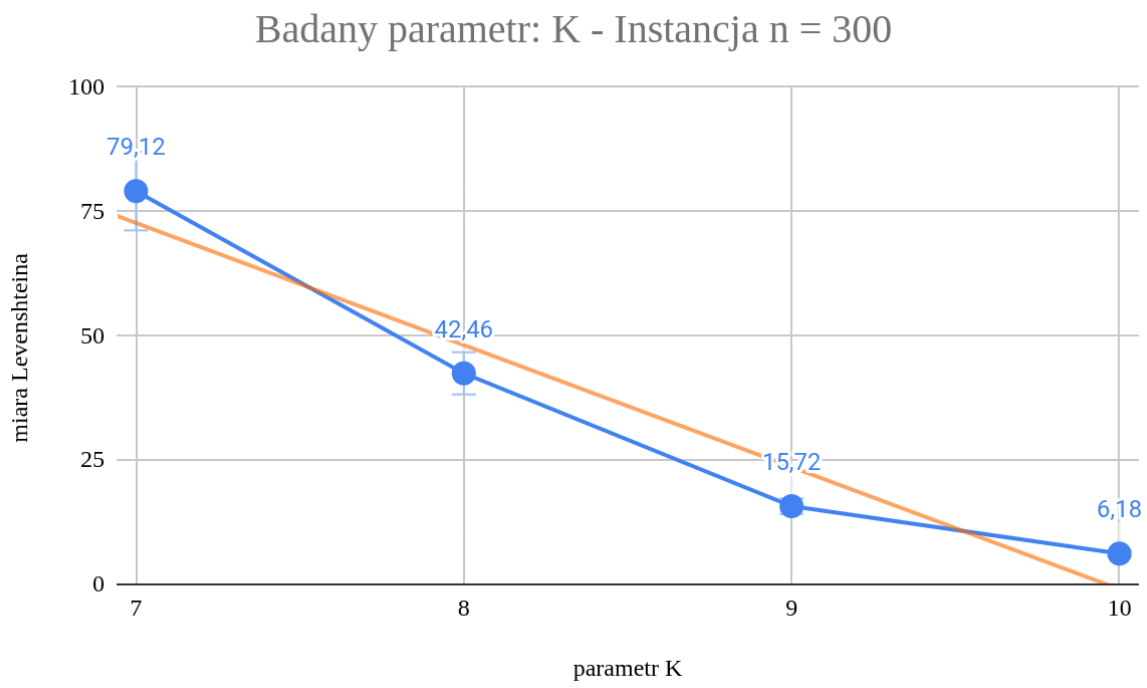


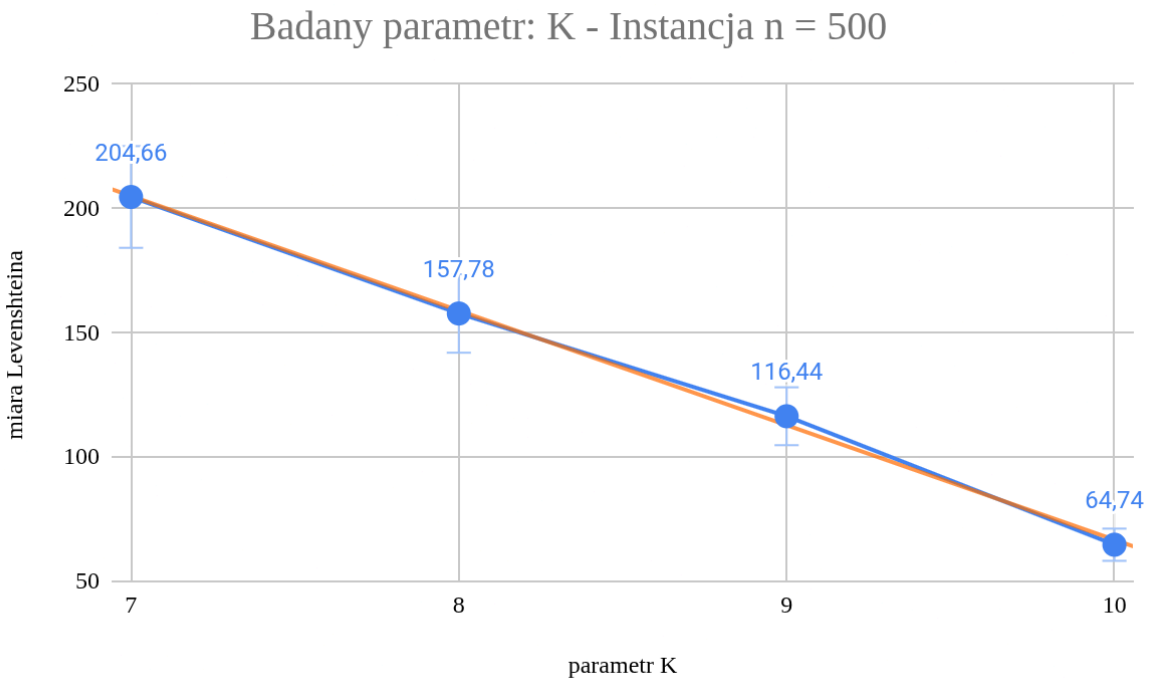


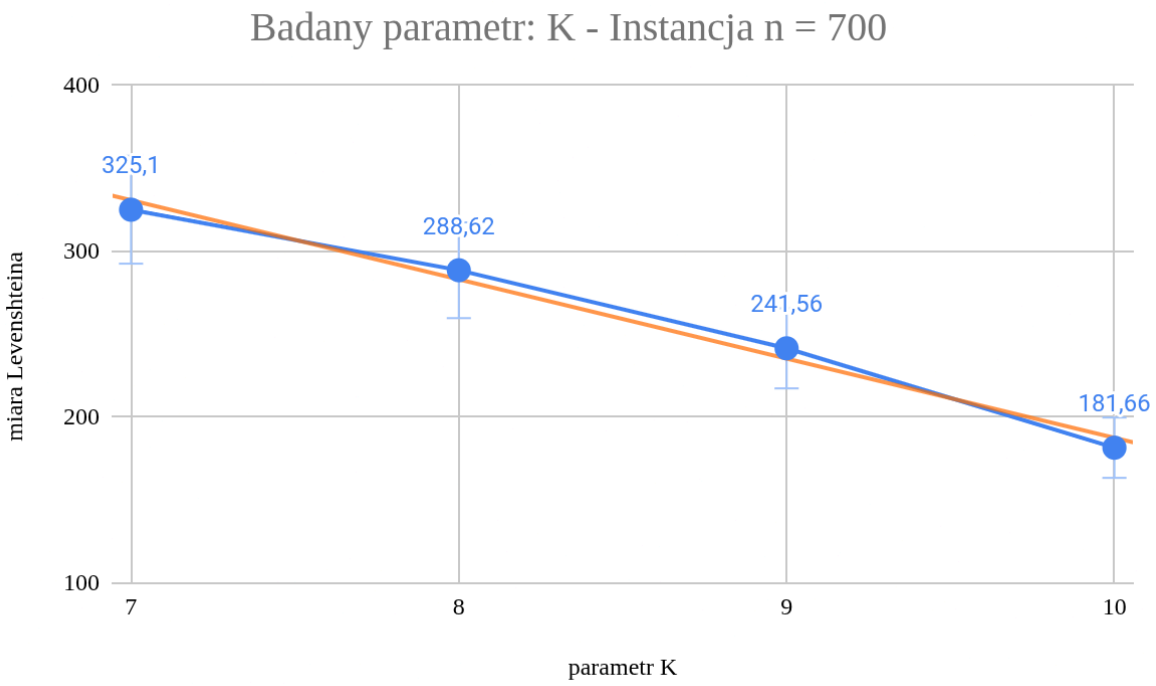
Wnioski i obserwacje:

Tutaj podobna sytuacja - im więcej błędów tym gorsze wyniki, tylko tutaj są one bardziej widoczne. Dlaczego? Fałszywe k-mery dodają szum do grafu i tworzą nieistniejące połączenia, które zmyłkowo przyciągają mrówki - zwłaszcza przy wysokiego parametru beta. Im więcej takich błędów, tym większa szansa że algorytm zbuduje niepoprawną trasę. Sam efekt jest silniejszy niż przy błędach negatywnych, ponieważ mrówka może zupełnie “odpłynąć” od poprawnej sekwencji, zamiast tylko chwilowo się zatrzymać.

Badany Parametr: K						
Instancja	Input	Output	K	300	500	700
n = 300, repeat = True, num_neg = 10%, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 79.12 avg Levenshtein = 42.46 avg Levenshtein = 15.72 avg Levenshtein = 6.18	K	Levenstein		
n = 500, repeat = True, num_neg = 10%, num_pos = 0	alpha = 0.7, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10 (s)	avg Levenshtein = 204.66 avg Levenshtein = 157.78 avg Levenshtein = 116.44 avg Levenshtein = 64.74	7	79,12	204,66	325,1
			8	42,46	157,78	288,62
n = 700, repeat = True, num_neg = 10%, num_pos = 0	alpha = 0.7, beta = 8.0, rho = 0.8, q = 20.0, tau0 = 0.7, ants = 100, time = 10(s)	avg Levenshtein = 325.10 avg Levenshtein = 288.62 avg Levenshtein = 241.56 avg Levenshtein = 181.66	9	15,72	116,44	241,56
			10	6,18	64,74	181,66







Wnioski i obserwacje

Wraz ze wzrostem długości k-merów, jakość rekonstrukcji wyraźnie się poprawia. Dlaczego?
Dłuższe k-mery są bardziej unikalne, bardziej odporny na błędy pozytywne i negatywne, więc graf ma mniej mylących połączeń i łatwiej prowadzi mrówki właściwą ścieżką.