Bioinformatyka Sprawozdanie nr. 2

Adam Tomczyk 141327, Jakub Różycki 141105

Badanie przez hybrydyzację z zastosowaniem rozwiązania problemu komiwojażera

Dane wejściowe: N l-nukleotydów

Kroki algorytmu:

1. Plik zawierający badane spektrum jest wczytywany do programu
2. Po wczytaniu każdego l-nukleotydu tworzymy strukturę grafu skierowanego
3. Utworzony przez nas graf jest grafem pełnym co pozwala nam na stworzenie macierzy o wielkości NxN
4. Każdy l-nukleotyd zostanie przypisany do jednego wierzchołka grafu (miasta w problemie komiwojażera)
5. Koszt dotarcia do następnego wierzchołka będzie liczony na podstawie pokrycia się "nazw" miast. Przykładowo wierzchołki "ACGGT" i "CGGTC" pokrywają się czterema z pięciu nukleotydów więc koszt dotarcia z jednego wierzchołka do drugiego wynosi 1, wierzchołki "AAAAC" oraz "ACGGT" pokrywają się tylko dwoma z pięciu nukleotydów. Przy braku pokrycia jakiekolwiek literki koszt wynosi długość l-nukleotydu. Koszt dla wierzchołków takich samych został ustalony na nieskończoność(nigdy więc nie pojawią się obok siebie)
6. Koszt taki liczony jest dwukrotnie, "od przodu" oraz "od tyłu". Daje nam to dwie trochę inaczej zbudowane macierze, co zwiększa losowość algorytmu i skutkuje lepszymi wynikami.
7. Dane potrzebne do poprawnego rozwiązania algorytmu (np. maksymalna długość ciągu, optymalna ilość mrówek) zostaną pobrane z nazwy akurat wykonywanego pliku.
8. Przed uruchomieniem algorytmu w sposób najprostszy szukane są potencjalne wierzchołki startowe. Wybierany jest wierzchołek, który w macierzy kosztów średnio najgorzej łączy się ze swoimi poprzednikami (co może oznaczać, że nie powinien się on łączyć z żadnym poprzednikiem bo jest wierzchołkiem początkowym). Krok ten jednak nie jest jednak bardzo ważnym krokiem i został on stworzony jako trochę lepsza alternatywa wybierania pierwszego wierzchołka z listy jako wierzchołka startowego.
9. Algorytm uruchamiany jest dwukrotnie dla macierzy "od przodu" oraz dla macierzy "od tyłu".
10. Parametry wejściowe algorytmu to:
    a. Macierz odległości/kosztów
    b. Ilość mrówek - połowa ilości wierzchołków
    c. Ilość mrówek zostawiających feromon - 0.3 ilości mrówek
    d. Ilość iteracji - 10 iteracji
    e. Współczynnik pozostawania feromonu - 0.8
    f. Współczynnik ważności feromonu - 5
    g. Współczynnik ważności odległości/kosztu - 3
    h. wierzchołek początkowy - wybierany w kroku 8
    i. Maksymalna długość ciągu - pobierana z nazwy pliku
    j. Parametr określający czy algorytm idzie "od przodu" czy "od tyłu"

k. **Komentarz odnośnie parametrów:**
Ilość iteracji została mocno zmniejszona w celu poprawy czasu wykonywania algorytmu co niestety negatywnie wpłynęło na wartość wyników. Wartości parametrów e, f i g została dobrana doświadczalnie (po przeprowadzeniu kilkunastu prób szukania odpowiedniej wartości) oraz po zasięgnięciu wiedzy z artykułów naukowych

11. Algorytm rozpoczyna się od funkcji run, która jest główną funkcją wykonującą. Ustala ona potrzebne parametry i odpala potrzebne funkcje. Większość operacji wykonywana jest tyle razy ile jest iteracji. Chyba najważniejszą zmienną definiowaną na początku algorytmu jest wartość zmiennej "self.feromon", która dla każdego wierzchołka ustalana jest na wartość $1/ilość\ wierzchołków$
12. Najważniejszą funkcją wywoływaną w "run" jest funkcja "wygenerujWszystkieSciezki". Jak sama nazwa mówi funkcja ta służy do wygenerowania ścieżek dla wszystkich mrówek. W pętli, dla i w przedziale od 0 do ilości mrówek, odpalana jest więc funkcja "wygenerujSciezke", która dla każdej mrówki generuje trasę (dokładniejszy opis w kolejnym punkcie), następnie liczona jest odległość tej ścieżki/sumaryczny koszt(za pomocą funkcji "podajOdelgloscSciezki", co jest wyznacznikiem jak dobrą ścieżkę udało się znaleźć. Ścieżka oraz jej długość dodawana jest do listy ze wszystkimi trasami oraz zwracana do funkcji "run"
13. Funkcja "wygenerujScizeke" działa na zasadzie wybierania wierzchołka i jego następnika na podstawie losowego wyboru ze zmiennym prawdopodobieństwem zależnym od ilości feromonu na ścieżce oraz odległości do następnego wierzchołka. Za wybór odpowiada funkcja "coDalej", która podejmuje decyzje o wybraniu następnika. Przebiega to na podstawie najpierw policzenia zmiennej "row" ze wzoru: $feromon\ wierzchołka^{współczynnik\ alpha} * wartość\ odległości\ do\ wierzchołka^{współczynnik\ beta}$. Zmienna ta zawiera prawdopodobieństwa wybrania wszystkich wierzchołków. Wartość zmiennej "row" jest następnie normalizowana i wykorzystywana (z użyciem np_choice z biblioteki numpy) do wyboru następnego wierzchołka. Wierzchołek jest dodawany do listy odwiedzonych i nie będzie już wykorzystywany.

```
row = feromon ** self.alpha * ((1.0 / dist) ** self.beta)

norm_row = row / row.sum()
move = np_choice(self.ileWierzcholkow, 1, p=norm_row)[0]
return move
```

14. Następnie wywoływana jest funkcja "wypuscFeromon", która służy do nałożenia dodatkowego feromonu na trasa znalezioncych przez k najlepszych mrówek. Ilość nałożonego feromonu to $0.5/odległość\ między\ wierzchołkami$ co sprawia, że im lepsze połączenie tym większa ilość feromonu. Na ten moment najlepsza trasa nadal jest definiowana tylko po najlepszym koszcie, a nie po największej ilości wierzchołków, ponieważ każda trasa składa się ze wszystkich wierzchołków.
15. Następna funkcja stara się rozwiązywać ten problem. Na początku próbowaliśmy zastosować dwa podejścia:
    a. podejście zatrzymywania algorytmu kiedy, przekroczył maksymalną długość ciągu. Niestety nie dawało to to najlepszych wyników przez to, że algorytm mrówkowy na początku działania daje słabe wyniki i polepsza się z czasem, a

przez ucinanie tras nie był wstanie zaznaczać dobrych połączeń co skutkowało słabymi wynikami.

b. podejście określenia ile jeszcze wierzchołków można dobrać do podanego ciągu, któremu brakuje jeszcze x długości do n. Niestety natomiast mieliśmy problem z wyborem kiedy ciąg powinien się zatrzymać i skupić się na szukaniu swoich wyrazów końcowych oraz tworzyło to duży problem optymalizacyjny.

16. Finalnie zdecydowaliśmy się na inną, prostszą lecz czasami dłuższą funkcję. Nazwana została “utnijKonceDoN” i polega na tym, że pozwalamy algorytmowi tworzyć tak długie ciągi jak mu pasuję z naciskiem aby dobre połączenia pojawiały się jak najwcześniej (przez zawyżony koszt słabych połączeń - pojawiały się tylko kiedy naprawdę musiały), a następnie skracaniu tych ciągów tak długo aż mieściły się w wyznaczonym N. Wydaje nam się, że rozwiązanie to działa naprawdę nieźle, pojawia się natomiast problem przy problemach z błędami pozytywnymi gdzie znalezione ciągi są zazwyczaj dużo dłuższe od N (przez dodatkowe wierzchołki) więc skracanie tych ciągów wydłuża czas przetwarzania.
17. Po tym kroku najlepsza ścieżka jest już wybierana inaczej, wybierana jest ścieżka z największą ilością wierzchołków, a jeżeli jest takich kilka wybierana jest ścieżka z długością najbardziej zbliżona do N (w celu szukania optimum a nie innych ścieżek, które czasem np zawierają wszystkie słowa ale mają mniejszą długość od N)
18. Po wykonaniu tych kroków wartość feromonu na wszystkich krawędziach zmmniejszana jest według ustalonego współczynika korzystając ze wzoru $feromon = feromon * współczynnik\, pozostawania\, feromonu$
19. Jeżeli natomiast najlepsze rozwiązanie nie zmienia się od wartości $0.15 * ilość\, iteracji$ wykonywana jest dodatkowa instrukcja. Wartości feromonów przywracane są do wartości początkowych oraz wywoływana jest funkcja “znajdzNajlepszyPoczatek”. Funkcja ta jako parametr przyjmuje najlepsza trasę z danej iteracji, wybiera wierzchołek o najgorszym połączeniu z poprzednikiem i najlepszym połączeniu z następnikiem i ustawia go jako nowy wierzchołek startowy. Sprawia to, że dynamicznie ustalane są nowe wierzchołki startowe co poprawia jakość wyników.

Wyniki eksperymentów na instancjach testowych:

1. Instancje z błędami negatywnymi losowymi:

Nazwa pliku	Długość ciągu	Ilość słów	Odchylenie od optimum	Czy znaleziono optimum	Czas przetwarzania w sekundach
10.500-100	509	395	0,988	NIE	253
10.500-200	508	295	0,983	NIE	161
18.200-40	206	157	0,981	NIE	63
18.200-80	205	119	0,992	NIE	41
20.300-120	309	180	1,000	TAK	75
20.300-60	309	240	1,000	TAK	112
25.500-100	509	400	1,000	TAK	251
25.500-200	506	291	0,970	NIE	159
35.200-40	209	160	1,000	TAK	63
35.200-80	207	120	1,000	NIE	42
53.500-100	503	396	0,990	NIE	252
53.500-200	509	292	0,973	NIE	158
55.300-120	309	180	1,000	TAK	74
55.300-60	306	238	0,992	NIE	112
55.400-160	409	234	0,975	NIE	112
55.400-80	409	320	1,000	TAK	179
58.300-120	307	179	0,994	NIE	75
58.300-60	307	238	0,992	NIE	112
62.400-160	405	235	0,979	NIE	113
62.400-80	408	316	0,988	NIE	175
68.400-160	406	236	0,983	NIE	114
68.400-80	409	320	1,000	TAK	174
9.200-40	207	160	1,000	NIE	63
9.200-80	207	119	0,992	NIE	42

2. Instancje z błędami negatywnymi wynikającymi z powtórzeń:

Nazwa pliku	Długość ciągu	Ilość słów	Odchylenie od optimum	Czy znaleziono optimum	Czas przetwarzania w sekundach
113.500-8	509	490	0,996	NIE	351
144.500-12	508	488	1,000	NIE	342
28.500-18	509	481	0,998	NIE	341
34.500-32	507	468	1,000	NIE	320
59.500-2	509	498	1,000	TAK	360

3. Instancje z błędami pozytywnymi losowymi

Nazwa pliku	Długość ciągu	Ilość słów	Odchylenie od optimum	Czy znaleziono optimum	Czas przetwarzania w sekundach
10.500+200	508	499	0,998	NIE	723
18.200+80	209	200	1,000	TAK	151
20.300+120	302	293	0,977	NIE	297
25.500+200	508	493	0,986	NIE	721
35.200+80	209	200	1,000	TAK	151
53.500+200	509	500	1,000	TAK	721
55.300+120	309	300	1,000	TAK	671
55.400+160	406	397	0,993	NIE	487
58.300+120	308	299	0,997	NIE	296
62.400+160	409	400	1,000	TAK	488
68.400+160	405	396	0,990	NIE	491
9.200+80	208	199	0,995	NIE	150

4. Instancje z błędami pozytywnymi, przekłamania na końcach oligonukleotydów

Nazwa pliku	Długość ciągu	Ilość słów	Odchylenie od optimum	Czy znaleziono optimum	Czas przetwarzania w sekundach
10.500+50	509	439	0,878	NIE	488
18.200+20	208	192	0,960	NIE	103
20.300+30	309	293	0,977	NIE	197
25.500+50	509	450	0,900	NIE	484
35.200+20	209	193	0,965	NIE	103
53.500+50	509	450	0,900	NIE	475
55.300+30	306	286	0,953	NIE	194
55.400+40	407	373	0,933	NIE	316
58.300+30	303	274	0,913	NIE	196
62.400+40	407	370	0,925	NIE	335
68.400+40	409	375	0,938	NIE	318
9.200+20	208	199	0,995	NIE	103

Wnioski:

1. Problem implementowaliśmy w dwóch językach - python oraz c++. Z rozwiązania w c++ zrezygnowaliśmy dosyć szybko przez dużo większą prostotę Pythona w wykonywanych przez nas działaniach, natomiast już na najprostszych wersjach algorytmu widać było znaczną przewagę w czasie przetwarzania języka c++. Pierwszą więc rzeczą do poprawy byłaby zmiana języka programowania na język szybciej przetwarzający dane jak właśnie np. c++
2. Zastosowanie zmiennej ilości mrówek zależnej od ilości wierzchołków sprawia, że czas wykonywania dla dużych instancji rośnie względem jakości rozwiązań. Wydaję nam się, że byłoby możliwe lepsze dopasowanie ilości mrówek względem ilości iteracji w celu zmniejszenia czasu przetwarzania i pozostawienia lub nawet polepszenia jakości wyników.

3. Zastosowanie funkcji odcinającej końce ciągów działa bardzo dobrze i bardzo szybko dla instancji z błędami negatywnymi, natomiast zdecydowanie wydłuża czas przetwarzania zbiorów z błędami pozytywnymi. Można by zastosować dodatkową funkcjonalność oceniającą czy jest sens skracać dany ciąg do N, jeżeli np ma bardzo słabo dobrany początek, naprawdę dobrze dobrany koniec który i tak będzie usunięty lub zastosować jeden z dwóch pomysłów wymienionych w punkcie 15 w krokach algorytmu.