Spis treści

1. Wprowadzenie do problemu dopasowania wielu sekwencji (MSA) 2
2. Teoria 3
   2.1 Multiple Sequence Alignment dla sekwencji DNA 3
   2.2 MSA dla aminokwasów 5
   2.3 Algorytm 8
3. Metaheurystyki 11
   3.1. Algorytm mrowkowy (ACO) w problemie MSA 11
   3.2 ACO - konstrukcja 12
      A.1. Co właściwie buduje mrówka? 12
      A.2. Pozycja mrówki i dopuszczalne ruchy 13
      A.3. Jak ocenić nową kolumnę? 13
   A.4. Jak zamienić ujemne punkty na dodatnią atrakcyjność? 14
      A.5. Gdzie zapisać feromon i jak wylosować ruch? 15
      A.6. Dlaczego mrówka nie powinna wybierać zawsze najlepszej kolumny? 16
      A.7. Jak połączyć składniki w całe ACO? 17
   3.2 Algorytm genetyczny 18
   3.2.2 Konstrukcja algorytmu – uproszczona :) 19
      B.1. Osobnik i populacja początkowa 19
      B.2. Fitness i wybór rodziców bez problemu ujemnych ocen 20
      B.3. Dlaczego zwykłe cięcie w tej samej kolumnie jest problematyczne? 20
      B.4. Operator krzyżowania (bez wspólnego cięcia) 21
      B.5. Mutacja 22
      B.6. Jak z tego powstaje kolejne pokolenie? 23
      B.7. Parametry, zakończenie i pierwsze kontrole 24
      Jak odróżnić błąd od słabej jakości rozwiązania? 24
      Co ulepszać dopiero po uruchomieniu podstawy? 24
5. Specyfikacja projektu/sprawozdanie 25
   4.1. Opis problemu i algorytmu 25
   4.2. Część eksperymentalna 25
      4.2.1 Wariant dla DNA. 27
      4.2.2 Wariant białkowy. 27
   4.3 Klasy danych testowych 27
      4.3.1. Klasy podobieństwa sekwencji 27
      4.3.2. Klasy o różnej strukturze luk 27
   4.4 Jak powinny wyglądać wykresy i tabele 28
      4.5 Inne metaheurystyki 28

Multiple Sequence Alignment

wersja 2.0.1 (20.10.2026)

1. Wprowadzenie do problemu dopasowania wielu sekwencji (MSA)

Analiza i porównywanie sekwencji pozwalają wnioskować o funkcji genów i białek, pokrewieństwie organizmów, konserwowanych fragmentach ewolucyjnych, a także o możliwych mutacjach związanych z chorobami. Z tego powodu dopasowywanie sekwencji należy do najważniejszych problemów obliczeniowych w bioinformatyce.

Najprostszą wersją tego zadania jest dopasowanie pary sekwencji. Jego celem jest takie ustawienie dwóch napisów względem siebie, aby podobne lub identyczne pozycje znalazły się w tych samych kolumnach. W praktyce wolno przy tym wstawiać znaki przerwy (gaps), najczęściej oznaczane jako „-", które reprezentują insercje lub delecje powstałe w procesie ewolucji. Dopasowanie ocenia się przy użyciu funkcji punktowej: zgodność znaków zwykle zwiększa wynik, niezgodność go obniża, a wstawienie luki jest karane. Już na tym poziomie widać wyraźnie informatyczny charakter problemu: trzeba zdefiniować reprezentację rozwiązania, funkcję celu oraz znaleźć algorytm wyznaczający rozwiązanie najlepsze lub przynajmniej bardzo dobre.

W wielu zastosowaniach porównanie tylko dwóch sekwencji nie wystarcza. Wtedy naturalnym rozszerzeniem jest **MSA (Multiple Sequence Alignment)** czyli dopasowanie wielu sekwencji jednocześnie. Celem nie jest już tylko stwierdzenie, czy dwie sekwencje są podobne, ale znalezienie wspólnego układu kolumn dla zbioru danych.

Ważne jest spojrzenie na MSA jako na problem optymalizacyjny. Rozwiązaniem jest tutaj pewne uporządkowanie sekwencji z dopuszczonymi lukami, natomiast jakość rozwiązania mierzy się za pomocą funkcji oceny. Jedną z najczęściej stosowanych jest metoda sum-of-pairs, w której całe dopasowanie ocenia się jako sumę ocen wszystkich dopasowań parami w każdej kolumnie. Taki model jest intuicyjny i dobrze nadaje się do analizy algorytmicznej, ale szybko okazuje się, że przestrzeń możliwych rozwiązań rośnie bardzo gwałtownie wraz z liczbą sekwencji i ich długością. O ile dla dwóch sekwencji istnieją klasyczne algorytmy dynamiczne wyznaczające rozwiązanie optymalne, o tyle dla wielu sekwencji problem staje się znacznie trudniejszy obliczeniowo i w praktyce wymaga stosowania heurystyk oraz metod przybliżonych.

W dalszej części materiału problem MSA będzie traktowany formalnie jako zadanie optymalizacyjne: zostanie zdefiniowana reprezentacja dopasowania, funkcja oceny oraz sposób obliczania wartości rozwiązania na konkretnym przykładzie. Takie ujęcie pozwala przejść od intuicyjnego porównywania sekwencji do ścisłego modelu, który można analizować, implementować i rozwiązywać metodami algorytmicznymi.

2. Teoria

2.1 Multiple Sequence Alignment dla sekwencji DNA

Załóżmy, że mamy dopasować takie 4 sekwencje (*w1-w4 : wiersze od 1 do 4).*

```
AGTCGTAG                                  w1   A G T C G T A G
ATCGTCG   ➔ dopasowujemy, np. tak: ➔      w2   A - T C G T C G
GTAG                                      w3   - - - - G T A G
GTAGAG                                    w4   - G T A G - A G
```

Należy teraz jakoś ocenić jakość dopasowania. Potrzebujemy częściowych punktów wg skali:

- **match**, +1 punkt, *np. A z A, G z G, itd.*
- **mismatch**, -1 punkt, *np. A z G*
- **gap$_{open}$, g$_{open}$** = -2 punkty – *„otwierający" znak – , np. wiersze 3 i 4 z pierwszej kolumny, ale też **druga** kolumna wiersza w2 oraz **szósta** kolumna dla wiersza w4 – to są „otwarcia" przerwy pomiędzy nukleotydami*
- **gap$_{extend}$, g$_{ext}$** = -1 punkt, *w przykładzie to druga, trzecia i czwarta kolumna wiersza w3*
- gap vs gap: 0 (zero) punktów.

Ocena za całe dopasowanie to suma ocen częściowych każdy-z-każdym, określa ją dwumian Newtona, w naszym przypadku „4 na 2" (= 6), gdzie 4 to liczba wierszy. Musimy ocenić dopasowania w1-w2 potem w1-w3, w1-w4, w2-w3, w2-w4 i ostatecznie w3-w4.

2.1.1 Przykład

Ocena za dopasowania w1 z w2:

```
A G T C G T A G
A - T C G T C G
```

match A-A: +1
g$_{open}$ G „–" : -2 punkty
match T-T: +1
match C-C: +1
match G-G: +1
match T-T: +1
mismatch A-C: -1
match G-G: +1

Czyli suma w1-w2 to: +1 –2 +1 +1 +1 +1 -1 +1 = 3.

Ocena za dopasowania w1 z w3:

```
A G T C G T A G
- - - - G T A G
```

g$_{open}$ A „–": -2 punkty
g$_{ext}$ G „–": -1 punkt
g$_{ext}$ T „–": -1
g$_{ext}$ C „–": -1
match G-G : +1
match T-T : +1
match A-A : +1
match G-G : +1

Suma w1-w3: -2 -1 -1 -1 +1 +1 +1 +1 = -5 + 4 = -1

Ocena za dopasowania w1 z w4:

A G T C G T A G
– G T A G – A G

g_{open} A i „–” : -2 punkty
match G-G: +1
match T-T: +1
mismatch C-A: -1
match G-G: +1
g_{open} T i „–” : -2 punkty
match A-A: +1
match G-G: +1

Suma w1-w4: -2 +1 +1 -1 +1 -2 +1 +1 = 0

Ocena za dopasowania w2 z w3:

A – T C G T C G
– – – – G T A G

g_{open} A i „–” : -2 punkty
porównanie „–” z „–” : 0 punktów
g_{open} T i „–” : -2 punkty
g_{ext} C i „–” : -1 punkt
match G-G: +1
match T-T: +1
mismatch C-A: -1
match G-G: +1

Suma w2-w3: -2 +0 -2 -1 +1 +1 -1 +1 = -3

Ocena za dopasowania w2 z w4:

A – T C G T C G
– G T A G – A G

g_{open} A i „–” : -2 punkty
g_{open} „–” i G : -2 punkty
match T-T: +1
mismatch C-A: -1
match G-G: +1
g_{open} T i „–” : -2 punkty
mismatch C-A: -1
match G-G: +1

Suma w2-w4: -2 -2 +1 -1 +1 -2 -1 +1 = -5

Ocena za dopasowania w3 z w4:

– – – – G T A G
– G T A G – A G

porównanie „–” z „–” : 0 punktów
g_{open} „–” i G : -2 punkty
g_{ext} „–” i T : -1 punkt
g_{ext} „–” i A : -1 punkt
match G-G: +1
g_{open} T i „–” : -2 punkty
match A-A: +1
match G-G: +1

Suma w3-w4: +0 -2 -1 -1 +1 -2 +1 +1 = -3

Ocena całkowita dopasowania: **SP** = 3 – 1 + 0 – 3 – 5 – 3 = **-9**

Czy to dużo czy mało? **Nie wiemy, możemy co najwyżej stworzyć inne dopasowania i porównywać wyniki**. Naszym celem jest oczywiście maksymalizacja, czyli najlepsze dopasowania to te z największą liczbą, nawet gdyby była ona ujemna.

2.2 MSA dla aminokwasów

W przypadku sekwencji białkowych ocena dopasowania jest zwykle bardziej realistyczna niż dla prostego schematu „zgodność/niezgodność”, ponieważ nie wszystkie zamiany aminokwasów są biologicznie jednakowo prawdopodobne. Z tego powodu stosuje się macierze podstawień takie jak **BLOSUM62**, które przypisują punktację każdej parze aminokwasów. Wysokie wartości dodatnie odpowiadają zamianom częstym i biologicznie „akceptowalnym”, natomiast wartości ujemne oznaczają podstawienia rzadsze lub mniej prawdopodobne. Kary za luki nie są częścią samej macierzy i muszą zostać określone osobno. W poniższym przykładzie pokazano sposób oceny dopasowania sekwencji białkowych z użyciem fragmentu macierzy BLOSUM62 oraz przyjętego modelu kar za luki, tak aby zilustrować, jak zmienia się punktacja w porównaniu z prostszymi przykładami dla sekwencji DNA.

2.2.1 Przykład: 5 sekwencji, przykładowe dopasowanie.

BLOSUM62 – uproszczona tabela dla przykładu z **kilkoma** aminokwasami (pełna jest większa):

	A	C	G	N	P	S	T	–
A	4	0	0	-2	-1	1	0	**-4**
C	0	9	-3	-3	-3	-1	-1	**-4**
G	0	-3	6	0	-2	0	-2	**-4**
N	-2	-3	0	6	-2	1	0	**-4**
P	-1	-3	-2	-2	7	-1	-1	**-4**
S	1	-1	0	1	-1	4	1	**-4**
T	0	-1	-2	0	-1	1	5	**-4**
–	**-4**	**-4**	**-4**	**-4**	**-4**	**-4**	**-4**	**0**

AGTCGTAGNPST **w1: A G T C G T A G N P S T**

ASTCGTAGPST **w2: A S T C G T A G – P S T**

GTPGAGNST **w3: – G T P G – A G N – S T**

ANCGTNPT **w4: A – N C G T – – N P – T**

GATAPS **w5: – G – – A T A – – P S –**

Pierwsze cztery dopasowania z $\binom{5}{2} = 10$ (każdy wiersz z każdym):

Ocena za dopasowanie w1 z w2:

A G T C G T A G N P S T
A S T C G T A G – P S T

A-A: +4
G-S: 0
T-T: +5
C-C: +9

G-G: +6
T-T: +5
A-A: +4
G-G: +6

gap N i „–” : -4 punkty
P-P: +7
S-S: +4
T-T: +5

Suma w1-w2: +4 +0 +5 +9 +6 +5 +4 +6 -4 +7 +4 +5 = **51**

Dopasowanie w1 z w3:

A G T C G T A G N P S T
– G T P G – A G N – S T

gap A i „–” : -4 punkty
G-G: +6
T-T: +5
C-P: -3
G-G: +6
gap T i „–” : -4 punkty
A-A: +4
G-G: +6
N-N: +6
gap P i „–” : -4 punkty
S-S: +4
T-T: +5

Suma w1-w3: -4 +6 +5 -3 +6 -4 +4 +6 +6 -4 +4 +5 = **27**

Dopasowanie w1 z w4:

A G T C G T A G N P S T
A – N C G T – – N P – T

A-A: +4
gap G i „–” : -4 punkty
T-N: 0
C-C: +9
G-G: +6
T-T: +5
gap A i „–” : -4 punkty
gap G i „–” : -4 punkty
N-N: +6
P-P: +7
gap S i „–” : -4 punkty
T-T: +5

Suma w1-w4: +4 -4 +0 +9 +6 +5 -4 -4 +6 +7 -4 +5 = **26**

Dopasowanie w1 z w5:

A G T C G T A G N P S T
– G – – A T A – – P S –

gap A i „–” : -4 punkty
G-G: +6
gap T i „–” : -4 punkty
gap C i „–” : -4 punkty
G-A: 0
T-T: +5
A-A: +4
gap G i „–” : -4 punkty
gap N i „–” : -4 punkty
P-P: +7
S-S: +4
gap T i „–” : -4 punkty

Suma w1-w5: -4 +6 -4 -4 +0 +5 +4 -4 -4 +7 +4 -4 = **2**

Pozostałe pary wierszy:

- suma **w2-w3** = **11**
- suma **w2-w4** = **16**
- suma **w2-w5** = **0**
- suma **w3-w4** = **-14**
- suma **w3-w5** = **-14**
- suma **w4-w5** = **-20**

Wszystkie “części” dopasowania (dla 5 sekwencji mamy łącznie 10 par):

- w1-w2 = 51
- w1-w3 = 27
- w1-w4 = 26
- w1-w5 = 2
- w2-w3 = 11
- w2-w4 = 16
- w2-w5 = 0
- w3-w4 = -14
- w3-w5 = -14
- w4-w5 = -20

Zatem wartość całego dopasowania wynosi: **SP = 51 + 27 + 26 + 2 + 11 + 16 + 0 - 14 - 14 - 20 = 85**

A teraz przyjmiemy: g_{open} = -12, g_{ext} = -1

Ocena za dopasowanie w1 z w2:

A G T C G T A G N P S T
A S T C G T A G – P S T

A-A: +4
G-S: +0
T-T: +5
C-C: +9

G-G: +6
T-T: +5
A-A: +4
G-G: +6

g_{open} N i „–” : -12 punktów
P-P: +7
S-S: +4
T-T: +5

Suma w1-w2: +4 +0 +5 +9 +6 +5 +4 +6 -12 +7 +4 +5 = **43**

Dopasowanie w1 z w3:

A G T C G T A G N P S T
– G T P G – A G N – S T

g_{open} A i „–” : -12 punktów
G-G: +6
T-T: +5
C-P: -3

G-G: +6
g_{open} T i „–” : -12 punktów
A-A: +4
G-G: +6

N-N: +6
g_{open} P i „–” : -12 punktów
S-S: +4
T-T: +5

Suma w1-w3: -12 +6 +5 -3 +6 -12 +4 +6 +6 -12 +4 +5 = **3**

Dopasowanie w1 z w4:

A G T C G T A G N P S T
A – N C G T – – N P – T

A-A: +4
g_{open} G i „–” : -12 punktów
T-N: +0
C-C: +9

G-G: +6
T-T: +5
g_{open} A i „–” : -12 punktów
g_{ext} G i „–” : -1 punkt

N-N: +6
P-P: +7
g_{open} S i „–” : -12 punktów
T-T: +5

Suma w1-w4: +4 -12 +0 +9 +6 +5 -12 -1 +6 +7 -12 +5 = **5**

Dopasowanie w1 z w5:

A G T C G T A G N P S T
– G – – A T A – – P S –

g_{open} A i „–” : -12 punktów
G-G: +6
g_{open} T i „–” : -12 punktów
g_{ext} C i „–” : -1 punkt

G-A: +0
T-T: +5
A-A: +4
g_{open} G i „–” : -12 punktów

g_{ext} N i „–” : -1 punkt
P-P: +7
S-S: +4
g_{open} T i „–” : -12 punktów

Suma w1-w5: -12 +6 -12 -1 +0 +5 +4 -12 -1 +7 +4 -12 = **-24**

Pozostałe pary wierszy:

- suma **w2-w3** = **-21**
- suma **w2-w4** = **-13**
- suma **w2-w5** = **-29**
- suma **w3-w4** = **-59**
- suma **w3-w5** = **-48**
- suma **w4-w5** = **-73**

Wszystkie elementy częściowe dopasowania:

- w1-w2 = 43
- w1-w3 = 3
- w1-w4 = 5
- w1-w5 = -24
- w2-w3 = -21
- w2-w4 = -13
- w2-w5 = -29
- w3-w4 = -59
- w3-w5 = -48
- w4-w5 = -73

Zatem: **SP** = 43 + 3 + 5 - 24 - 21 - 13 - 29 - 59 - 48 - 73 = **-216**. To jest formalnie poprawne dla konwencji BLAST/NCBI.

2.3 Algorytm

Ok, tak możemy sobie to rozpisywać samodzielnie na kartce. Ale trzeba to jakoś zrobić w programie. Wracamy do DNA. Spójrzmy jeszcze raz na dopasowanie:

```
w1    A G T C G T A G
w2    A - T C G T C G
w3    - - - - G T A G
w4    - G T A G - A G
```

Macierz binarna:

```
1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1
0 0 0 0 1 1 1 1
0 1 1 1 1 0 1 1
```

Czyli widać teraz, że nasze dopasowanie może być opisane macierzą, w której jedynka oznacza literę, a zero oznacza kreskę. Tylko tyle i aż tyle. Ale uwaga, z tego wynika jednak genialnie prosty i trywialny sposób generowania rozwiązań losowych:

- musimy wygenerować **tyle wierszy** zer i jedynek **ile jest sekwencji** do dopasowania,
- wiersz opisujący daną sekwencję wejściową **musi mieć dokładnie tyle jedynek, ile dane sekwencja ma liter** (ale może je mieć gdziekolwiek i jakkolwiek rozdzielone zerami).

Kolumny w tej macierzy możemy sobie nazywać jak chcemy, przy czym często, choćby w kontekście algorytmów metaheurystycznych nazywa się je *ruchami* (w sensie: ruch który zmienia jedno rozwiązanie w drugie, względnie pewna liczba znanych już ruchów (ułożeń) tworzy nowe rozwiązanie). W tym znaczenie ruch to coś, co prowadzi nas dalej, do konsktrukcji rozwiązania dopuszczalnego. Rozwiązanie dopuszczalne jest zdefiniowane tymi dwoma powyższymi punktami nad tym akapitem, to jest, **musi mieć tyle wierszy, ile sekwencji** oraz **liczba jedynek w wierszu musi się równać liczbie liter w danej sekwencji** (reszta to zera). Należy tutaj zwrócić uwagę na to, że kolumna składająca się z samych zer nie ma wielkiego sensu – można by ją dodać wiele razy, ale poza obniżaniem wyniku funkcji celu nie służyłaby ona do niczego pozytywnego…

Należy też zauważyć, że każdy ruch można oceniać (*prawie*) niezależnie. „*Prawie*” ponieważ dla określania gap_{open} / gap_{extend} w aktualnie analizowanej kolumnie, musimy odwołać się do tego co jest w wierszu w kolumnie wcześniejszej, czyli tej po lewej stronie od aktualnej.

- Innymi słowy: w wierszy X w kolumnie Y natrafiamy na kreskę, czyli gap. Czy ona kosztuje ocenę -1 punkt czy -2 punkty, **zależy od tego co jest w kolumnie Y-1 tego samego wiersza**.
    - Jak jest tam już kreska, to teraz mamy -1, bo to „tylko” gap_{extend}.

- A jeżeli w kolumnie Y-1 jest jakakolwiek litera, to znaczy, że teraz, w kulumnie Y, ta nasza kreska to gap_{open}, czyli -2 punkty do oceny.

Rysunek raz jeszcze:

```
w1    A G T C G T A G
w2    A - T C G T C G
w3    - - - - G T A G
w4    - G T A G - A G
```

Macierz binarna:

```
w1    1 1 1 1 1 1 1 1
w2    1 0 1 1 1 1 1 1
w3    0 0 0 0 1 1 1 1
w4    0 1 1 1 1 0 1 1
```

Np. pierwsza kolumna to „1 1 0 0". Jest tam +1 punkt za prawidłowe dopasowanie nukletydów A z pierwszego i drugiego wiersza. A także cztery raz po -2 punkty za „gap_{open}" (wiersz w1 z w3 (-2 pkt.) i i w4 (-2 pkt.), oraz wiersz w2 z w3 (-2 pkt) i w2 z w4 (-2). Razem: +1 – 2 – 2 – 2 – 2 = -7 (jest jeszcze porównanie w3 z w4, ale to jest „kreska z kreską" czyli zero punktów które wyniku nie zmienia!)

W drugiej kolumnie „1 0 0 1" mamy „gap_{open}" (w1 vs. w2, w2 vs. w4 oraz w3 vs. w4 (**! patrz niżej**), każdy karany -2 pkt.), dwa razy „gap_{extend}" (w1 z w3 oraz w3 z w4, każde karane -1 pkt), 1 match (w1 z w4, +1 pkt). Porównanie gap'ów (w2 z w3) jest za zaro punktów. Czyli -2 -1 +1 +0 -2 – 2 = -6 (w1-w2; w1-w3; w1-w4; w2-w3, w2-w4, w3-w4). Każdy ruch/kolumnę można więc oceniać niezależnie kolumna po kolumnie, tak jak na poprzedniej stronie było pokazane, jak wiersz po wierszu liczyliśmy funkcję celu/jakość dopasowania.

Uwaga, ważne: w3 vs w4 to gap_{open}, pomimo, że to druga kreska w w3 – chodzi o to, że dopiero w tej kolumnie pojawia się pierwsza litera (G, po kresce) w w4! Czyli jeżeli mamy dwa wiersze, i w poprzedniej kolumnie dla nich były kreski (w obu, w tym przykładzie jest tak w pierwszej kolumnie) a w aktualnej kolumnie jest litera/kreska (w przykładzie: kreska w w3, litera w w4) to jest to wciąż gap_{open} pomimo faktu, że to już druga kolejna kreska dla w3! Jeszcze innymi słowy: kreska w wierszu **w3**, w drugiej kolumnie, to gap_{open} względem litery **G** w wierszu **w4**, ale równocześnie $gap_{exptend}$ względem litery **G** w wierszu w1.

Wracając do przykładu z aminokwasami:

```
AGTCGTAGNPST        w1: A G T C G T A G N P S T
ASTCGTAGPST         w2: A S T C G T A G - P S T
GTPGAGNST           w3: - G T P G - A G N - S T
ANCGTNPT            w4: A - N C G T - - N P - T
GATAPS              w5: - G - - A T A - - P S -
```

Macierz binarna:

```
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 1 1 1
0 1 1 1 1 0 1 1 1 0 1 1
1 0 1 1 1 1 0 0 1 1 0 1
0 1 0 0 1 1 1 0 0 1 1 0
```

Wersja 1: prosta kara za gap to „stałe" minus cztery (-4) punkty.

Dla przykładu aminokwasowego rozważmy kolumny siódmą i ósmą. W kolumnie siódmej mamy układ **„A A A – A"**. Występuje tu sześć porównań typu **A** – **A** (w1 z w2, w1 z w3, w1 z w5, w2 z w3, w2 z w5, w3 z w5), każde punktowane **+4**, co daje razem **+24 punktów**. Ponadto są cztery porównania aminokwasu z luką (w1 z w4, w2 z w4, w3 z w4 oraz w4 z w5), każde po **-4 punkty**, czyli razem **-16 punktów**.

Suma dla całej kolumny wynosi więc **+24 - 16 = +8**. W kolumnie ósmej mamy układ **„G G G – –"**. Są tu trzy porównania **G** – **G** (w1 z w2, w1 z w3, w2 z w3), każde po **+6 punktów**, czyli razem **+18 punktów**. Dodatkowo występuje 6 porównań aminokwasu z luką (w1 z w4, w1 z w5, w2 z w4, w2 z w5, w3 z w4, w3 z w5), każde po **-4 punkty**, co daje **-24 punkty**, a porównanie dwóch luk (w4 z w5) ma wartość **0**.

Ostatecznie dla tej kolumny otrzymujemy **+18 - 24 + 0 = -6**. W tym uproszczonym modelu każdą kolumnę można więc oceniać niezależnie, ponieważ kara za przerwę nie zależy od tego, co znajdowało się w kolumnie poprzedniej.

Wersja 2: gap_{open}= –12, gap_{extend}= –1

Dla tego samego przykładu aminokwasowego, ale przy punktacji z rozróżnieniem na **gap_{open} = -12** oraz **gap_{extend} = -1**, sytuacja jest bardziej złożona, ponieważ do oceny danej kolumny trzeba znać także kolumnę poprzednią (tak jak na przykładzie z DNA na początku tego pliku. Rozważmy ponownie kolumny szóstą i siódmą:

```
T A  |
T A  |
- A  |
T -  |
T A  |
```

W kolumnie siódmej znów mamy sześć porównań **A** – **A**, każde po **+4 punkty** czyli razem **+24 punkty**. Cztery porównania **–** w w4 (**w1** z w4, **w2** z w4, **w3** z w4 oraz **w5** z w4) są tutaj traktowane jako **gap_{open}**, ponieważ w kolumnie szóstej nie było dla tych par kreski w wierszu czwartym, tylko litera **T** (tutaj specjalnie dla wyróżniania zaznaczona na niebiesko).

Każde z tych czterech porównań daje więc **-12 punktów**, łącznie **-48 punktów**, a zatem suma dla kolumny siódmej wynosi **+24 - 48 = -24**. W kolumnie ósmej mamy układ **„G G G – –"**. Trzy porównania G-G (w1 z w2, w1 z w3, w2 z w3) dają razem **+18 punktów**. Dla par w1 z w4, w2 z w4 oraz w3 z w4 luka w w4 trwa już od kolumny siódmej, więc są to trzy przypadki **gap_{extend}**, po **-1 punkcie** każdy. Natomiast dla par w1 z w5, w2 z w5 oraz w3 z w5 luka pojawia się dopiero teraz, więc są to trzy przypadki **gap_{open}**, po **-12 punktów** każdy. Porównanie w4 z w5 to znowu przerwa z przerwą, czyli **0 punktów**. Ostatecznie dla kolumny ósmej otrzymujemy więc **+18 - 1 - 1 - 1 - 12 - 12 - 12 + 0 = -21**. Widać zatem, że przy modelu gap_{open}/gap_{extend} kolumn nie można oceniać całkowicie niezależnie: trzeba pamiętać, czy w danej parze sekwencji przerw właśnie się zaczyna, czy jest już tylko przedłużana.

3. Metaheurystyki

3.1. Algorytm mrowkowy (ACO) w problemie MSA

Algorytm mrowkowy, czyli Ant Colony Optimization (ACO), jest metaheurystyką inspirowaną zachowaniem prawdziwych mrówek poszukujących pożywienia. W wersji obliczeniowej mamy zbiór prostych agentów, nazywanych mrówkami, które budują kolejne rozwiązania problemu. Każda mrówka konstruuje rozwiązanie krok po kroku, poruszając się po odpowiednio zdefiniowanym grafie (lub innej strukturze danych opisującej sposoby konstrukcji rozwiązania). Wybór kolejnego ruchu nie jest całkowicie losowy i zależy z jednej strony od śladu feromonowego, czyli informacji zgromadzonej przez wcześniejsze dobre rozwiązania, a z drugiej strony lokalnej oceny atrakcyjności danego ruchu. Po zakończeniu iteracji feromon na mniej obiecujących decyzjach częściowo zanika (parowanie), a na elementach należących do lepszych rozwiązań zostaje wzmocniony (bo rozwiązania w kolejnych iteracjach zawierają ruchy, które wcześniej były użyte i zostawiły ślad feromonowy). Dzięki temu algorytm stopniowo uczy się, które decyzje warto preferować.

W problemie MSA (Multiple Sequence Alignment) ideę tę trzeba przełożyć na język dopasowania sekwencji. Tutaj rozwiązaniem nie jest trasa komiwojażera ani ścieżka w klasycznym grafie, lecz pełne dopasowanie wielu sekwencji, czyli **układ kolumn** zawierających symbole biologiczne oraz ewentualne luki. Z punktu widzenia ACO trzeba więc odpowiedzieć na trzy pytania:

- jak reprezentować rozwiązanie,
- jaki ruch wykonuje mrówka,
- na czym odkładać feromon.

Tutaj pojawia się ważna cecha ACO dla MSA: nie istnieje jeden kanoniczny wariant algorytmu. W literaturze spotyka się kilka sposobów modelowania problemu. Jedne podejścia budują dopasowanie bezpośrednio jako ścieżkę w wielowymiarowym grafie, który uwzględnia możliwe wstawienia luk i kolejne decyzje wyrównania. Inne traktują ACO jako metodę udoskonalania już istniejącego dopasowania, na przykład wygenerowanego wcześniej przez heurystykę. Są też warianty hybrydowe, w których ACO działa jako lokalne przeszukiwanie osadzone w większym algorytmie, na przykład genetycznym.

Najbardziej intuicyjna interpretacja wydaje się następująca. Wyobraźmy sobie, że mrówka buduje dopasowanie kolumna po kolumnie. W każdej chwili musi zdecydować, które sekwencje „przesunąć" dalej o jeden znak, a w których wstawić lukę. Taki wybór odpowiada zbudowaniu następnej kolumny dopasowania. Dla k sekwencji każda kolumna może być opisana jako wektor binarny długości k: jedynka oznacza pobranie kolejnego symbolu (litery oznaczającej nukleotyd lub aminokwas) z danej sekwencji, a zero oznacza wstawienie luki. Nie wszystkie wektory (kolumny, zwłaszcza pod koniec) są oczywiście dopuszczalne, ale każdy dopuszczalny wektor można potraktować jako lokalny ruch w przestrzeni rozwiązań. Mrówka powtarza taki wybór wielokrotnie, aż wszystkie sekwencje zostaną w pełni wykorzystane, a my otrzymujemy kompletne dopasowanie. To ujęcie łączy się z wcześniejszą reprezentacją macierzy binarnej dla MSA, bo ACO można po prostu rozumieć jako metodę inteligentnego konstruowania kolejnych kolumn tej macierzy.

W praktyce decyzja mrówki nie jest czysto losowa. Każdy możliwy ruch ma dwa składniki oceny. Pierwszy to feromon: jeżeli w poprzednich iteracjach pewne decyzje często występowały w dobrych dopasowaniach, to ich atrakcyjność rośnie. Drugi składnik to informacja heurystyczna: na przykład lokalne podobieństwo znaków,

przewidywany koszt luki, zgodność z macierzą podstawień albo poprawa funkcji celu. Właśnie połączenie tych dwóch mechanizmów daje równowagę między eksploracją a eksploatacją. **Eksploracja** oznacza sprawdzanie nowych układów kolumn i nowych miejsc wstawiania luk, natomiast **eksploatacja** oznacza częstsze wybieranie decyzji, które już wcześniej doprowadzały do dobrych wyników (czyli większe bazowanie na feromonach). Po każdej iteracji najlepsze albo wszystkie zbudowane dopasowania wpływają na aktualizację feromonu, zaś proces parowania feromonu zapobiega temu, aby algorytm zbyt wcześnie „uwierzył" w jedną, być może tylko lokalnie dobrą, strukturę rozwiązania (problem optimum lokalnego).

W MSA jakość rozwiązania mierzy się zwykle przy użyciu pewnej funkcji oceny, ewentualnie z karami za luki i macierzą podstawień dla białek. Po skonstruowaniu dopasowania przez mrówkę można obliczyć jego wartość i na tej podstawie zdecydować, ile feromonu należy dopisać do decyzji, które do tego dopasowania doprowadziły. Najważniejsze jest to, że ACO dla MSA nie zagwarantuje optimum globalnego. MSA ma ogromną przestrzeń rozwiązań, a liczba możliwych układów luk i kolumn rośnie bardzo szybko wraz z liczbą sekwencji i ich długością.

3.2 ACO - konstrukcja

Poniżej ustalimy dokładnie, co mrówka pamięta, jakie decyzje może podejmować, jak je losuje i czego uczy się cała ich „kolonia". Przez cały przykład używamy punktacji DNA: match = +1, mismatch = −1, $\text{gap}_{\text{open}} = -2$, $\text{gap}_{\text{extend}} = -1$, gap–gap = 0. Maksymalizujemy SP (czyli np. wynik −3 jest lepszy niż −8).

A.1. Co właściwie buduje mrówka?

Niech k oznacza liczbę sekwencji, a n_i długość sekwencji S_i. Gotowe dopasowanie zapisujemy jako macierz binarną B o k wierszach i L kolumnach. Liczba kolumn L nie musi być znana na początku, ale powstaje podczas budowania dopasowania.

Jedynki w wierszu odczytujemy od lewej do prawej. Pierwsza jedynka oznacza pierwszą literę sekwencji, druga - drugą literę itd. Zera zamieniamy na kreski. Dzięki temu nie przestawiamy liter i nie wybieramy ich na nowo.

Przykład. Dla sekwencji $S_1 = \mathbf{ACG}$, $S_2 = \mathbf{AG}$ i $S_3 = \mathbf{ACG}$:

Macierz B		Dopasowanie	
w1	**1 1 1**	w1	**A C G**
w2	**1 0 1**	w2	**A - G**
w3	**1 1 1**	w3	**A C G**

Poprawna macierz musi mieć odpowiednią liczbę jedynek w każdym wierszu **oraz przynajmniej jedną jedynkę w każdej kolumnie** (kolumna z samymi zerami jest **bez sensu**, jeżeli się choć chwilę zastanowimy; przy czym: w toku działania metaheurystyki może powstać, więc trzeba pilnować, żeby takie artefakty potem usuwać).

$$\sum_{c=1}^{L} B_{i,c} = n_i \qquad \text{oraz} \qquad \sum_{i=1}^{k} B_{i,c} \geq 1.$$

Czyli: wykorzystaj wszystkie litery i żadnej nie dodawaj, oraz: nie twórz pustej kolumny. Wszystkie wiersze muszą mieć tę samą długość L. W szczególności:

$$\max_i n_i \leq L \leq \sum_{i=1}^{k} n_i.$$

Dolna granica wynika z długości najdłuższej sekwencji, a górna z tego, że każda kolumna zużywa co najmniej jedną literę.

A.2. Pozycja mrówki i dopuszczalne ruchy

Mrówka przechowuje wektor $p = (p_1, \dots, p_k)$, gdzie p_i jest **liczbą już wykorzystanych liter** sekwencji S_i (czyli indeks „i" od 1 – pierwsza sekwencja, do k – ostatnia). Na początku $p = (0, \dots, 0)$. Gdy $p_i < n_i$, następną dostępną literą jest litera numer $p_i + 1$ tej sekwencji; numerację liter zaczynamy od 1.

Ruch $d = (d_1, \dots, d_k)$ jest **kolumną** zer i jedynek. Jeśli $d_i = 1$, to do nowej kolumny trafia następna litera z sekwencji S_i, a licznik p_i zwiększa się o jeden. Jeśli $d_i = 0$, trafia tam luka, a licznik się nie zmienia (bo liczy tylko litery!). Krótko zapisujemy to jako:

$$p' = p + d.$$

Są tylko dwa ograniczenia, tj. ruch nie może składać się z samych zer, a w sekwencji już zakończonej musi mieć zero. Jeszcze innymi słowy, oczywista oczywistość: nie można pobrać litery, której już nie ma :) Zbiór ruchów spełniających te warunki oznaczymy przez $D(p)$.

Jeżeli wszystkie trzy sekwencje (z przykładu) mają jeszcze litery, możliwe są ruchy: 111, 110, 101, 011, 100, 010 i 001. Jeśli zakończyła się druga sekwencja, pozostają tylko 101, 100 i 001. **Najpierw ustalamy ruchy dopuszczalne, dopiero potem je oceniamy i losujemy.**

Dla naszego przykładu jedna z mrówek może zbudować dopasowanie następująco. W zapisie kolumny znaki są podane w kolejności wierszy. Sekwencje to: $S_1 =$ **ACG**, $S_2 =$ **AG** i $S_3 =$ **ACG**

Krok	Liczniki przed ruchem	Ruch	Nowa kolumna	Liczniki po ruchu
1	(0, 0, 0)	111	**A**, **A**, **A**	(1, 1, 1)
2	(1, 1, 1)	101	**C**, **–**, **C**	(2, 1, 2)
3	(2, 1, 2)	111	**G**, **G**, **G**	(3, 2, 3)

Ostatni wektor jest równy (n_1, n_2, n_3), więc dopasowanie jest gotowe (n_i – dla przypomnienia – długość każdej sekwencji i, czyli n_1=3, n_2=2, n_3=3). Zauważmy, że ruch 101 nie oznacza zawsze tych samych liter, bo zależą one od aktualnego indeksu kolejnej wolnej litery: p_i.

Mrówka pamięta także poprzednią kolumnę binarną, potrzebną do rozróżnienia gap_{open} i $\text{gap}_{\text{extend}}$. Na samym początku nie ma żadnej „poprzedniej" kolumny, więc każde porównanie litery z luką jest otwarciem. Ponadto zapisuje wybierane pary (p, d), bo będą potrzebne przy aktualizacji feromonu.

A.3. Jak ocenić nową kolumnę?

Nową kolumnę oceniamy tak samo jak kolumnę gotowego dopasowania: porównujemy każdą parę wierszy i sumujemy punkty. Przy k wierszach mamy $k(k-1)/2$ takich porównań. W poniższej tabeli 1 oznacza literę, a 0 lukę. „Taki sam układ" oznacza, że luka pozostaje po tej samej stronie porównywanej pary.

Układ w parze teraz	Co było w tej parze poprzednio?	Punkty teraz
(1, 1)	Bez znaczenia	+1 za te same litery; −1 za różne
(0, 0)	Bez znaczenia	0
(0, 1)	Dokładnie 0, 1	−1: $\text{gap}_{\text{extend}}$
(1, 0)	Dokładnie 1, 0	−1: $\text{gap}_{\text{extend}}$
(0, 1) lub (1, 0)	Inny układ albo początek dopasowania	−2: gap_{open}

To doprecyzowuje przyjętą tutaj konwencję: po układzie 0, 0 kolejne porównanie litery z luką dostaje karę za gap_{open}. Również zmiana z (0, 1) na (1, 0) otwiera nową lukę. **Poprzednia kreska w jednym wierszu nie wystarcza do stwierdzenia, że mamy** $\text{gap}_{\text{extend}}$**.**

Niech $q(d)$ oznacza ocenę rozważanej kolumny. To zapis skrócony, bo ocena zależy również od liczników p i poprzedniej kolumny. Jeżeli $v_{i,j}$ jest wynikiem porównania wierszy i i j, to:

$$q(d) = \sum_{1 \le i < j \le k} v_{i,j}.$$

Przykładowo kolumna **A**, **A**, **A** ma ocenę $1 + 1 + 1 = 3$. Kolumna **C**, **–**, **C** po kolumnie **A**, **A**, **A** ma ocenę: $-2 + 1 - 2 = -3$ (2 x -2 punkty za gap_{open} **C** vs. **–** oraz +1 pkt. za match **C** vs.**C**). W każdym przypadku są to trzy porównania: w1–w2, w1–w3 oraz w2–w3.

Ocena całego dopasowania jest sumą ocen kolejno zbudowanych kolumn. Podczas sprawdzania programu można jeszcze policzyć ją również niezależnie, parami całych wierszy. **Oba sposoby muszą dać ten sam wynik.** Dla czterech sekwencji z wcześniejszego przykładu DNA w materiale wyniki par to 3, −1, 0, −3, −5 i −3, a ich suma wynosi −9.

A.4. Jak zamienić ujemne punkty na dodatnią atrakcyjność?

Oceny kolumn bywają ujemne. Nie można więc użyć ich bezpośrednio jako prawdopodobieństw ani jako dodatnich wag losowania. Wprowadzimy osobną wielkość: **atrakcyjność heurystyczną** $\eta(d)$ (gracka litera jako „*eta*"). Ma ona podpowiadać, który ruch wygląda dobrze teraz, w danym momencie.

Najpierw oceniamy wszystkie ruchy z $D(p)$ i znajdujemy najmniejszy wynik $q_{\min}$. Następnie przyjmujemy:

$$\eta(d) = q(d) - q_{\min} + 1.$$

Najgorszy ruch otrzymuje atrakcyjność 1, a lepsze odpowiednio większą. Dodanie tej samej liczby do wszystkich ocen nie zmienia ich kolejności. Minimum wyznaczamy na nowo przy każdym wyborze kolumny - nie dla całego uruchomienia.

Przykład. Po pierwszej kolumnie **A**, **A**, **A** mamy $p = (1,1,1)$. Następne litery to **C**, **G** i **C**. Wszystkie ruchy są jeszcze dopuszczalne (patrz opis pod tabelą, pomoże zrozumieć oceny).

Ruch	Rozważana kolumna	Ocena $q(d)$	Atrakcyjność $\eta(d)$
111	**C**, **G**, **C**	−1	5
101	**C**, **–**, **C**	−3	**3**
110	**C**, **G**, **–**	−5	1
011 (**najgorzej**)	**–**, **G**, **C**	**–5**	1
100	**C**, **–**, **–**	−4	2
010	**–**, **G**, **–**	−4	2
001	**–**, **–**, **C**	−4	2

Tutaj $q_{\min} = -5$ (to ten najmniejszy wynik dla **011**). Dla ruchu 101 otrzymujemy zatem $\mathbf{-3 - (-5) + 1 = 3}$ (pierwsze –3 to jego ocena q(d)= –3, –5 to nasze minimum atrakcyjności (q_{min}), a ostatnie +1 wynika ze wzoru: $\eta(d) = q(d) - q_{\min} + 1$). Gdy wszystkie dopuszczalne ruchy mają tę samą ocenę, każdy dostaje atrakcyjność 1. Najlepszy ruch 111 ma $\eta(d) = 5$, ponieważ $\eta(d) = q(d) - q_{\min} + 1 = -1 - (-5) + 1 = -1 + 5 + 1 = 5$.

Należy pamiętać, że $\eta(d)$ to tylko pomoc przy podejmowaniu decyzji przez algorytm. **Wynik dopasowania nadal liczymy z oryginalnych punktów, nie z wartości η.** Nie sumujemy atrakcyjności, aby wyznaczyć SP i nie porównujemy za jej pomocą gotowych rozwiązań.

A.5. Gdzie zapisać feromon i jak wylosować ruch?

Nieodmiennie dla przypomnienia: sekwencje to: $S_1 = \mathbf{ACG}$, $S_2 = \mathbf{AG}$ i $S_3 = \mathbf{ACG}$

Przyjmijmy, że feromon jest liczbą $\tau(p, d)$ przypisaną do pary: (**liczniki przed ruchem, wybrany ruch**). Symbol τ odczytujemy jako „*tau*”. Na początku każda taka wartość wynosi 1.

Przykładowo $\tau\big((\mathbf{1,1,1}), \mathbf{101}\big)$ opisuje doświadczenie związane z wybraniem kolumny C, –, C w naszym przykładzie (jest to druga kolumna). Dla przypomnienia: ***p = (1,1,1)*** oznacza, że w poprzedniej, czyli pierwszej kolumnie wstawiliśmy po jednej literze na wiersz (literki **A**), więc licznik ***p = (1,1,1)*** oznacza, że każda pierwsza litera każdej z trzech sekwencji została już użyta. **101** w $\tau\big((\mathbf{1,1,1}), \mathbf{101}\big)$ oznacza natomiast **ruch** opiany binarnie jako **101**, to: stawienie litery w pierwszym wierszu (litera **C**, pierwsza jedynka), przerwy w drugim (zero), litery w trzecim (litera **C**, druga jedynka). ***(Już zupełnie na marginesie: wykonanie tego ruchu zmieni liczniki użytych do tej pory liter p na p = (2,1,2).***

$\tau\big((\mathbf{1,1,1}), \mathbf{101}\big)$ nie jest tym samym co feromon ruchu 101 przy licznikach (2, 2, 2), gdzie pobierane byłyby inne litery (tj., odpowienio: trzecia z sekwencji 1eszej, czyli litera **G** i 3ciej – też litera **G**. No i kreska w drugim wierszu, bo tam już nie ma liter (sekwencja S_2 ma dwie litery). Sam numer kolumny również nie wystarcza, bo np. w dwóch dopasowaniach o tej samej liczbie kolumn można już wykorzystać różne liczby liter.

Dla uproszczenia nie umieszczamy poprzedniej kolumny w kluczu feromonu. Dwa dojścia do tego samego licznika p współdzielą więc tę część doświadczenia. Poprzednia kolumna nadal jest jednak uwzględniana w ocenie $q(d)$, a zatem i w heurystyce. Dla każdego dopuszczalnego ruchu obliczamy dodatnią wagę:

$$w(d) = \tau(p, d)^{\alpha}\, \eta(d)^{\beta}.$$

Parametr α określa wpływ feromonu, a β wpływ heurystyki. Dla wersji startowej przyjmiemy $\alpha = 1$ oraz $\beta = 2$: feromon bierzemy bez zmiany, a atrakcyjność podnosimy do kwadratu. Prawdopodobieństwo otrzymujemy przez podzielenie wagi przez sumę wag:

$$P(d) = \frac{w(d)}{\sum_{u \in D(p)} w\,(u)}$$

Litera u w mianowniku oznacza kolejno każdy dopuszczalny ruch. Ruchy niedopuszczalne nie uczestniczą w tej sumie i mają prawdopodobieństwo zero.

W przykładzie z poprzedniej tabeli, którą tu jeszcze raz wklejam, przy **każdym** feromonie równym $\tau(p, d) = 1$, wagi wynoszą kolejno 25, 9, 1, 1, 4, 4 i 4. Ich suma wynosi 48. Otrzymujemy wagi (ze wzoru $w(d) = \tau(p, d)^{\alpha}\, \eta(d)^{\beta}$. Czyli 25 to wynik $w(111) = \tau(p, d)^{\alpha}\, \eta(d)^{\beta} = (1)^{\alpha} \cdot (5)^{\beta} = (1)^{1} \cdot (5)^{2} = 1 \cdot 25 = 25$.

Ruch	Rozważana kolumna	Ocena $q(d)$	Atrakcyjność $\eta(d)$	Waga	Prawdopodobieństwo
111	**C**, **G**, **C**	−1	5	25	25/48 ≈ 52,08%
101	**C**, –, **C**	−3	3	9	9/48 = 18,75%
110	**C**, **G**, –	−5	1	1	1/48 ≈ 2,08%
011	–, **G**, **C**	−5	1	1	1/48 ≈ 2,08%
100	**C**, –, –	−4	2	4	4/48 ≈ 8,33%
010	–, **G**, –	−4	2	4	4/48 ≈ 8,33%
001	–, –, **C**	−4	2	4	4/48 ≈ 8,33%

Załóżmy **tylko dla przykładu**, że po aktualizacja wartości feromonu to byłoby $\tau\big((\mathbf{1,1,1}), \mathbf{101}\big) = 2$.

Wtedy$w(\mathbf{101}) = \tau((1,1,1), \mathbf{101})^{\alpha}\, \eta(\mathbf{101})^{\beta} = (2)^{\alpha} \cdot (3)^{\beta} = (2)^{1} \cdot (3)^{2} = 2 \cdot 9 = 18$ (czyli 18, zamiast wagi 9 w tabelce).

Jak wykonać takie losowanie? Ustawiamy ruchy w kolejności z tabeli. Losujemy liczbę rzeczywistą r z przedziału [0, 48). Ruch 111 zajmuje odcinek [0, 25), ruch 101 odcinek [25, 34), kolejne dwa odcinki mają długość 1, a trzy ostatnie długość 4. Dla $r = 30$ wybierzemy więc 101. Korzystamy z niezaokrąglonych wag, nie z procentów przepisanych z tabeli. **Komentarz:** łatwe? Łatwe. A ten operator **selekcji** nazywa się **ruletką**. **Do zapamiętania.**

Nie wybieramy zawsze największej wagi. Gdybyśmy tak robili, zniknęłaby możliwość sprawdzania mniej oczywistych rozwiązań (to jest z kolei **selekcja rankingowa**). Jeśli pozostał tylko jeden dopuszczalny ruch, jego prawdopodobieństwo automatycznie wynosi 1.

A.6. Dlaczego mrówka nie powinna wybierać zawsze najlepszej kolumny?

Porównajmy dwa dopasowania tych samych trzech sekwencji:

Dopasowanie X		**Dopasowanie Y**	
w1	A C G	w1	A C G
w2	A – G	w2	A C –
w3	A C G	w3	A C G

Dopasowanie	Ocena kolumny 1	Ocena kolumny 2	Ocena kolumny 3	Całe SP
X	3	−3	3	**3**
Y	3	−1	−3	**−1**

Druga kolumna Y jest lokalnie lepsza niż druga kolumna X. Zużywa jednak **G** z drugiej sekwencji za wcześnie: w ostatniej kolumnie trzeba już wstawić lukę. W dopasowaniu X chwilowo przyjmujemy gorszą kolumnę **C**, –, **C**, aby później otrzymać **G**, **G**, **G**.

Właśnie dlatego heurystyka jest tylko **wskazówką**. Feromon ma przekazywać inną informację, mianowicie, czy decyzja uczestniczyła w dobrym **całym dopasowaniu**.

Parowanie i wzmocnienie

Po zbudowaniu dopasowań przez wszystkie mrówki wybieramy to z największym SP. Przy remisie losujemy jedno z najlepszych. Tylko ono wzmacnia feromon w danej iteracji. **Nie trzeba** przyznawać nagrody proporcjonalnej do SP (ale **można**, tj. jest wiele możliwości konstrukcji metaheurystyk, w tym oczywiście ACO).

Najpierw każda wartość feromonu jest mnożona przez $1 - \rho$, gdzie ρ jest częścią, która wyparowuje. Następnie dodajemy stałą nagrodę $Q = 1$ do każdej pary (p, d) wykorzystanej w wybranym dopasowaniu. Przyjmujemy również dolną granicę $\tau_{\min} = 0{,}05$, aby żaden ruch nie tracił całkowicie szansy wyboru:

$$\tau_{\text{nowa}}(p,d) = \max\{\tau_{\min}, (1-\rho)\tau(p,d)\} + \Delta(p,d),$$

przy czym $\Delta(p,d) = Q$ dla decyzji z najlepszego dopasowania tej iteracji, a $\Delta(p,d) = 0$ dla pozostałych decyzji.

Dla $\rho = 0{,}2$ i początkowego feromonu 1 decyzja niewzmocniona otrzyma 0,8, a wzmocniona $0{,}8 + 1 = 1{,}8$. Gdy najlepsze było X, wzmacniamy dokładnie trzy pary: ((0, 0, 0), 111), ((1, 1, 1), 101) i ((2, 1, 2), 111).

W kolejnym wyborze przy $p = (1{,}1{,}1)$ waga ruchu 101 wyniesie $1{,}8 \cdot 3^2 = 16{,}2$, a waga ruchu 111 wyniesie $0{,}8 \cdot 5^2 = 20$. Suma wag wszystkich siedmiu ruchów wyniesie 47,4. Szansa ruchu 101 wzrośnie więc z 18,75% do około 34,18%. Feromon nie wymusza więc decyzji, lecz zwiększa jej szansę wyboru.

Ta reguła działa także wtedy, gdy wszystkie końcowe SP są ujemne. Spośród wyników −20, −12 i −7 wybieramy −7 i przyznajemy dodatnie Q. Czyli: nie stosujemy $1/SP$ ani nie odkładamy ujemnego feromonu.

A.7. Jak połączyć składniki w całe ACO?

Początek uruchomienia. Ustalamy parametry i początkowy feromon. Przygotowujemy też miejsce na najlepsze dopasowanie znalezione w całym uruchomieniu. Nie wpisujemy tam sztucznego wyniku 0, bo prawdziwe wyniki mogą być wyłącznie ujemne. Najprościej zapisać pierwsze ukończone dopasowanie.

Jedna iteracja. Każda z m mrówek zaczyna od pustego dopasowania oraz zerowych liczników. Buduje kolejne kolumny opisanym sposobem, aż wykorzysta wszystkie litery. Wszystkie mrówki w tej iteracji korzystają z tych samych wartości feromonu; nie zmieniamy ich w trakcie budowania pojedynczych dopasowań. Na końcu porównujemy pełne SP, aktualizujemy zapamiętany rekord i wykonujemy jedno parowanie oraz wzmocnienie.

Następna iteracja i zakończenie. Mrówki znów zaczynają budowanie od zera, ale używają już zmienionego feromonu. Po wykonaniu ustalonej liczby iteracji zwracamy rekord z całego uruchomienia, niekoniecznie zwycięzcę ostatniej iteracji. Najlepszy wynik danej iteracji może być gorszy od wcześniejszego rekordu.

Element	Wariant startowy
Liczba mrówek m	20
Liczba iteracji	100
Początkowy feromon	1
Wpływ feromonu i heurystyki	$\alpha = 1, \beta = 2$
Parowanie	$\rho = 0{,}2$
Dolna granica feromonu	$\tau_{\min} = 0{,}05$
Wzmocnienie	$Q = 1$; nagradza tylko zwycięzca iteracji

Jak nie przechowywać ogromnej tablicy?

Nie trzeba tworzyć wszystkich możliwych wektorów p i ruchów. Wystarczy **słownik**, przechowujący wartości dla par, które otrzymały wzmocnienie feromonowe. Wszystkie pozostałe pary mają wspólną wartość domyślną, początkowo równą 1.

> **Dygresja**: ***słownik***: chodzi o strukturę danych: (**klucz**, **wartość**) – są to dynamiczne struktury danych, tylko bardziej skomplikowane niż ta ukochana przez Was lista dwukierunkowa na podstawach programowania: (ノ°□°) ノ⌒┻┻.
>
> Dobra wiadomość: nie bawimy się w implementację tego samodzielnie, tylko wykorzystujemy istniejące biblioteki języka, np. dla **C++** radzę zajrzeć tutaj: https://www.geeksforgeeks.org/cpp/how-to-create-a-dictionary-in-cpp/

Przy parowaniu zmniejszamy zarówno wartości w słowniku, jak i wartość domyślną, zawsze z tą samą dolną granicą. Jeśli nagradzana para nie była jeszcze zapisana, bierzemy jej aktualną wartość domyślną po parowaniu, dodajemy Q i zapisujemy wynik. Nie przywracamy takiej parze początkowej wartości 1. Dzięki temu opis matematyczny pozostaje taki sam, a nie tworzymy ogromnej tablicy wypełnionej jednakowymi liczbami.

Co jest ograniczeniem tej wersji?

Jeśli a sekwencji ma jeszcze niewykorzystane litery, istnieje $2^a - 1$ dopuszczalnych ruchów. Dla 3 sekwencji jest ich 7, dla 8 - 255, a dla 10 - 1023. Ocena każdej kolumny wymaga ponadto porównań parami. Dlatego pełne wyliczanie ruchów jest prostym rozwiązaniem dla małych instancji, ale szybko drożeje.

Po uruchomieniu wersji podstawowej można ograniczać liczbę rozważanych ruchów albo poprawiać gotowe dopasowanie lokalnymi przesunięciami luk. Trzeba też pamiętać, że dokładne klucze (p, d) mogą rzadko powtarzać się dla większych instancji, przez co uczenie feromonu bywa słabe. Zmiana sposobu uogólniania tego doświadczenia jest osobnym ulepszeniem, a nie warunkiem uruchomienia pierwszego algorytmu.

3.2 Algorytm genetyczny

Dla MSA algorytm genetyczny (GA) najlepiej opisywać jako GA z operatorami przystosowanymi do problemu dopasowania. To ważne, bo zwykłe bitowe mutacje i krzyżowanie bardzo łatwo psują poprawność dopasowania: zmieniają długości wierszy, zmieniają kolejność znaków w sekwencji albo tworzą pełne kolumny kresek (gaps). Najprostszy wariant można oprzeć na reprezentacji: **osobnik** = **pełne dopasowanie MSA**, czyli macierz o $k$wierszach i wspólnej długości L, gdzie każdy wiersz zawiera symbole sekwencji oraz przerwy. Chromosomem nie jest tu pojedyncza sekwencja, tylko całe dopasowanie. Funkcja fitness może być dokładnie ta sama jak z przykładów, sum-of-pairs z odpowiednią punktacją, np. match/mismatch/gap albo macierz BLOSUM + kary za przerwy.

Schemat algorytmu. Najpierw generujemy populację startową, na przykład losowo (ale można jakąś prostszą heurystyką, bo im lepsze wejście, tym lepszy wynik). Następnie w każdej generacji liczymy dopasowanie, wybieramy rodziców, tworzymy rozwiązania potomne przez krzyżowanie i mutację, naprawiamy dopasowanie, jeżeli potrzeba, usuwamy pełne kolumny przerw, a potem wybieramy osobniki do następnej generacji. W praktyce bardzo pomaga strategia elitaryzmu, czyli przepisywanie kilku najlepszych dopasowań bez zmian do kolejnej populacji.

Najważniejsza część to **krzyżowanie**. W MSA sensowny crossover powinien łączyć fragmenty całych dopasowań. Najprostsza wersja to jednopunktowe krzyżowanie (dla dopasowań): wybieramy kolumnę cięcia w rodzicu nr 1 a potem w drugim, dobieramy miejsce tak, aby po sklejeniu lewej części jednego i prawej części drugiego zachować poprawną kolejność znaków w każdej sekwencji.

Druga sensowna wersja to *uniform/block crossover*, w którym rozwiązanie potomne składa się z naprzemiennie wybieranych bloków pochodzących od różnych rodziców, ale tylko między pozycjami „zgodnymi", czyli takimi, gdzie połączenie nie łamie spójności sekwencji. Jeszcze prościej: krzyżowanie to wtedy wymiana bloków kolumn między dwoma dopasowaniami, ale z warunkiem, że po tej wymianie każdy wiersz nadal zawiera symbole swojej sekwencji w tej samej kolejności co przed dopasowaniem. Jeżeli po sklejeniu długości wierszy się różnią, trzeba dołożyć przerwy wyrównujące; jeżeli powstaną kolumny z samych przerw, trzeba je usunąć.

Mutacja daje algorytmowi możliwość „przesuwania" luk i lokalnego poprawiania dopasowań. Najprostsza mutacja to *gap insertion*: wybieramy grupę sekwencji i wstawiamy w nich przerwę tego samego rozmiaru, a w pozostałych sekwencjach wyrównujemy długość odpowiednim wstawieniem w innym miejscu. Taki operator generuje nowe konfiguracje przerw których wcześniej nie było.

Druga dość "naturalna" mutacja to tzw. *block shuffling* albo prościej: przesuwanie bloków przerw lub bloków znaków w lewo albo w prawo. Idea jest taka, że po wstawieniu przerwy optymalny układ często różni się tylko tym, że blok powinien być przesunięty o 1–3 kolumny. Podsumowując taki sposób: mutacja nie zmienia biologicznej treści sekwencji, tylko zmienia położenie przerw lub całych lokalnych fragmentów dopasowania.

Nadają się tutaj takie mutacje jak na przykład:

- **Shift mutation**: wybieramy jedną przerwę albo blok przerw w danym wierszu i przesuwamy go o jedną kolumnę w lewo lub prawo, jeśli nie zmienia to kolejności znaków.
- **Merge/split mutation**: dwa sąsiednie bloki przerw łączymy w jeden, albo odwrotnie: duży blok dzielimy na dwa mniejsze.
- **Column cleanup mutation**: usuwamy pełne kolumny przerw (to nawet nie mutacja, tylko korekta rozwiązania) lub kolumny prawie puste, a następnie naprawiamy długości wierszy. Taki zestaw jest dość praktyczny, bo lokalnie eksploruje przestrzeń rozwiązań bez gwałtownego niszczenia dobrych fragmentów dopasowania. Ogólna idea „gap-oriented mutations" jest zgodna z kierunkiem rozwoju

operatorów w GA dla MSA, gdzie mutacje projektuje się wokół przerw i bloków, a nie wokół pojedynczych liter.

Bardzo ważny element techniczny to naprawa rozwiązania po krzyżowaniu (lub mutacji). Po każdym operatorze powinno się sprawdzić:

1. Upewnić się, że w każdym wierszu znaki występują w poprawnej kolejności (liczba znaków).
2. Usunąć pełne kolumny przerw.

Selekcja rodziców może być zwykła: turniejowa albo rankingowa, albo jeszcze inna, np. ruletka.

GA dla MSA działa na całych dopasowaniach jako osobnikach; krzyżowanie miesza dobre bloki kolumn z dwóch rodziców, a mutacja lokalnie przesuwa i rekonfiguruje przerwy, po czym najlepsze dopasowania przechodzą do następnych generacji oceniane selekcją według funkcji celu.

3.2.2 Konstrukcja algorytmu – uproszczona :)

Tutaj nie budujemy pojedynczego rozwiązania za pomocą mrówki. Przechowujemy wiele pełnych dopasowań i tworzymy z nich kolejne. Jednym osobnikiem jest **całe MSA**, a nie pojedynczy wiersz. Poniżej ustalamy konkretny sposób tworzenia populacji, wyboru rodziców, krzyżowania i mutacji.

W przykładach nadal obowiązuje punktacja DNA: match = +1, mismatch = −1, $\text{gap}_{\text{open}} = -2$, $\text{gap}_{\text{extend}} = -1$ oraz gap–gap = 0. Rozróżnienie otwarcia i przedłużenia jest wykonywane osobno dla każdej pary wierszy, według przyjętej wcześniej reguły poprzedniej kolumny.

B.1. Osobnik i populacja początkowa

Niech k oznacza liczbę sekwencji, a n_i długość sekwencji S_i. Osobnik jest macierzą binarną o k wierszach: wiersz i ma dokładnie n_i jedynek. Wszystkie jego wiersze mają jednakową długość, ale **różni osobnicy mogą mieć różne liczby kolumn**. Nie wymuszamy jednej długości chromosomu dla całej populacji.

Populacja jest zbiorem przechowywanych osobników; jej liczność oznaczymy przez N. Można zacząć np. od $N = 40$. Każdy osobnik opisuje dopasowanie tego samego zestawu sekwencji, w tej samej kolejności wierszy.

Jak stworzyć losowe rozwiązanie MSA?

Najpierw wybieramy jego roboczą długość L_0, nie mniejszą od długości najdłuższej sekwencji. Prosty przepis polega na dodaniu niewielkiego, losowego zapasu:

$$L_0 = n_{\max} + r, \qquad n_{\max} = \max_i n_i.$$

Przykładowo losujemy całkowite r od 0 do $\min\{h, \sum_i n_i - n_{\max}\}$, gdzie $h = \max\{1, \lceil 0, 2n_{\max} \rceil\}$. Innymi słowy: dopuszczamy zapas około 20% długości najdłuższej sekwencji, z co najmniej jednym możliwym dodatkowym miejscem, ale nie przekraczamy sumy długości wszystkich sekwencji. Jest to wybór do inicjalizacji, nie ograniczenie późniejszej ewolucji.

Następnie dla wiersza i losujemy **bez powtórzeń** dokładnie n_i pozycji spośród L_0 kolumn. Na tych pozycjach umieszczamy jedynki, na pozostałych zera. Powtarzamy to niezależnie dla wszystkich wierszy, a na końcu usuwamy kolumny złożone z samych zer. Po usunięciu takich kolumn ostateczna długość może być mniejsza od L_0.

Przykład. Dla sekwencji: **A C G**, **A G** oraz **A C G** wybierzmy $L_0 = 4$. W pierwszym i trzecim wierszu wylosowano pozycje 1, 2 i 4, a w drugim 1 i 3:

Wylosowana macierz	**Odczytane dopasowanie**
w1 **1 1 0 1**	**A C – G**
w2 **1 0 1 0**	**A – G –**
w3 **1 1 0 1**	**A C – G**

To dopasowanie jest poprawne, choć jego SP wynosi −7. **Losowanie ma zapewnić poprawność i różnorodność, a nie od razu wysoką jakość.** Nie losujemy liter, tylko miejsca przeznaczone na już ustalone litery. Powtarzamy tworzenie osobników, aż uzyskamy N dopasowań. W pierwszej wersji można dopuścić powtórzenia. Wymaganie bezwzględnej unikalności przy bardzo małych danych mogłoby spowodować niekończące się próby uzupełnienia populacji.

B.2. Fitness i wybór rodziców bez problemu ujemnych ocen

Funkcja przystosowania, czyli fitness, będzie równa SP:

$$F(B) = SP(B).$$

Wybierzmy selekcję **turniejową**. Losujemy równomiernie dwa różne alignmenty z bieżącej populacji i porównujemy ich fitness. Lepszy osobnik zostaje rodzicem, a jeśli mają równy wynik, rozstrzygamy losowo. Aby wybrać drugiego rodzica, przeprowadzamy nowy turniej. Ten sam osobnik może wygrać oba turnieje – ale nie jest to najbezpieczniejsza strategia (optimum lokalne).

Przykładowo w populacji z wynikami −14, −3, −8 i −6 turniej między −14 a −8 wygrywa osobnik z −8. Nie jest on najlepszy w całej populacji, ale był lepszy w swojej wylosowanej parze. Dzięki temu także słabsze rozwiązania mają czasem potomstwo. Turniej między −3 a −6 wygrywa −3.

W takim wyborze wystarczy porównać dwie liczby. Naiwna ruletka $F(B)/\sum F$ może dawać ujemne prawdopodobieństwa przy mieszanych znakach ocen, a przy samych ujemnych wynikach preferuje osobników gorszych. Turniej nie wymaga takiego przeliczania.

B.3. Dlaczego zwykłe cięcie w tej samej kolumnie jest problematyczne?

Krzyżowanie ma połączyć początek jednego dopasowania z dalszą częścią drugiego. Nie wolno jednak zakładać, że ta sama kolumna w dwóch rodzicach odpowiada temu samemu miejscu sekwencji. Liczba wcześniejszych luk może być różna.

Rozważmy sekwencje ACGT, AGT i ACT oraz dwa poprawne dopasowania. Pionowa kreska poniżej zaznacza cięcie, nie jest znakiem dopasowania:

Rodzic P		**Rodzic Q**	
w1	A C | G − T	w1	A − C | G T
w2	A − | − G T	w2	− A − | G T
w3	A C | − − T	w3	A − C | − T

Po lewej stronie cięcia w P wykorzystano odpowiednio 2, 1 i 2 litery. Taki sam wektor liczników, $(2,1,2)$, uzyskujemy w Q dopiero po trzeciej kolumnie. Można więc bezpośrednio skleić lewą część P z prawą częścią Q:

Dziecko			
w1	A C | G T	→	A C G T
w2	A − | G T	→	A − G T
w3	A C | − T	→	A C − T

Gdybyśmy obcięli również Q po drugiej kolumnie, jego pierwszy wiersz miałby jeszcze po prawej stronie CGT. Dołączenie tego fragmentu do A C z P dałoby A C C G T, czyli powtórzenie litery C.

Dopisanie luk tego nie naprawi. Bezpieczne jest zatem łączenie po takiej samej liczbie wykorzystanych liter w każdym wierszu, a nie koniecznie po takim samym numerze kolumny. Także identyczne litery widoczne przy cięciu nie wystarczają: w sekwencji mogą się one powtarzać.

B.4. Operator krzyżowania (bez wspólnego cięcia)

W poprzednim przykładzie rodzice mieli pasujące pionowe cięcia, ale nie zawsze tak będzie. Poniższy przepis działa również wtedy, gdy w drugim rodzicu nie istnieje jedna kolumna odpowiadająca całemu wektorowi liczników.

Początek z rodzica P. Wybieramy losowo cięcie po jednej z kolumn od 1 do $L_P - 1$, gdzie L_P jest długością P. Zachowujemy cały lewy prostokąt. Jeśli P ma tylko jedną kolumnę, w tej próbie pomijamy krzyżowanie, bo nie ma w nim wewnętrznego cięcia.

Ustalenie, ile już zabraliśmy. W każdym zachowanym wierszu liczymy litery. Jeśli cięcie było po kolumnie c, oznaczmy tę liczbę przez q_i:

$$q_i = \sum_{j=1}^{c} P_{i,j}$$

Tutaj $P_{i,j}$ jest bitem macierzy rodzica P. Symbol q_i w tej sekcji jest licznikiem liter, nie oceną kolumny z opisu ACO.

Dalsza część z rodzica Q. Kopiujemy dopasowanie Q. W jego wierszu i zamieniamy pierwsze q_i liter na luki, nie usuwając przy tym ich kolumn. Te litery mamy już z P, więc nie wolno ich pobierać drugi raz. Wszystkie pozostałe litery zostają na dotychczasowych pozycjach. W reprezentacji binarnej oznacza to wyzerowanie pierwszych q_i jedynek tego wiersza. Jeśli $q_i = 0$, wiersz pozostaje bez zmian; jeśli $q_i = n_i$, staje się w całości wierszem luk.

Następnie z tak przygotowanej części Q usuwamy kolumny, które są puste we wszystkich wierszach, i doklejamy pozostały prostokąt z prawej strony zachowanego początku P. Nie dosuwamy osobno każdego wiersza w lewo - stracilibyśmy układ kolumn odziedziczony po Q.

Przykład bez wspólnej kolumny cięcia

Zostawmy ten sam początek P: **A C**, **A –** oraz **A C**.

Potrzebujemy zatem $q = (2,1,2)$. Drugi rodzic niech wygląda tak:

Rodzic Q		**Po usunięciu wykorzystanych liter**		
w1	**A C G T**	w1	**– – G T**	(pierwsze 2 litery do luk)
w2	**A G – T**	w2	**– G – T**	(pierwsza 1 litera do luki)
w3	**A C – T**	w3	**– – – T**	(pierwsze 2 litery do luk)

Pierwsza kolumna po prawej jest teraz pusta. Po jej usunięciu i dołączeniu początku P otrzymujemy:

Początek P		**Dalsza część Q**		**Gotowe dziecko**	
w1	**A C**	w1	**– G T**	w1	**A C – G T**
w2	**A –**	w2	**G – T**	w2	**A – G – T**
w3	**A C**	w3	**– – T**	w3	**A C – – T**

Po usunięciu luk z gotowych wierszy odzyskujemy ACGT, AGT i ACT.

Ogólnie w wierszu i bierzemy z P pierwsze q_i liter, a z Q pozostałe $n_i - q_i$. Łącznie jest ich dokładnie:

$$q_i + (n_i - q_i) = n_i.$$

Kolejność również jest zachowana: najpierw występuje początek sekwencji, potem jej dalsza część. **To uzasadnia poprawność operatora; nie dowodzi, że dziecko będzie lepsze.** Na granicy części mogą powstać niekorzystne luki i punkty trzeba policzyć ponownie dla całego dziecka.

W jednej próbie wystarczy tworzyć jedno dziecko. Role P i Q wynikają z kolejności wybrania rodziców. Podstawowy operator nie wymaga wyszukiwania wspólnych cięć: opisany sposób z lukami działa także w przykładzie z sekcji B.3 i daje tam to samo, krótsze dziecko.

B.5. Mutacja

Mutacja zmienia położenie luk, a nie treść sekwencji. W podstawowej wersji zastosujemy tylko dwa dokładnie określone operatory.

Mutacja 1: zamiana sąsiadującej litery i luki

Wybieramy losowo jeden wiersz i miejsce, gdzie sąsiadują litera oraz luka. Zamieniamy je miejscami. W macierzy binarnej jest to zamiana 10 na 01 albo 01 na 10. Najprościej losować spośród wszystkich takich dopuszczalnych par w całym dopasowaniu. Liczba liter i długość wiersza nie zmieniają się.

Przed mutacją		**Po mutacji drugiego wiersza**	
w1	A C G	w1	A C G
w2	A G –	w2	A – G
w3	A C G	w3	A C G

Tutaj SP rośnie z −1 do 3. Przesunięte G nie przeskoczyło innej litery, tylko lukę, więc kolejność sekwencji AG pozostała niezmieniona. Nie wolno natomiast losowo przestawiać dwóch liter wiersza ani odwracać fragmentu sekwencji.

Taka mutacja nie zawsze poprawia wynik. Dopuszczamy również pogorszenie: późniejsza selekcja i elityzm decydują, które rozwiązania będą zachowane. Przyjmowanie wyłącznie poprawiających przesunięć byłoby dodatkowym przeszukiwaniem lokalnym, a nie tą samą regułą mutacji.

Mutacja 2: wstawienie nowej luki z wyrównaniem długości

Same zamiany 10↔01 nie potrafią stworzyć nowej luki, zwłaszcza w dopasowaniu sekwencji jednakowej długości bez żadnych przerw. Potrzebujemy więc drugiego operatora.

Wybieramy losowo jeden wiersz oraz jedną z jego istniejących kolumn. Wstawiamy w tym wierszu lukę **przed wybraną kolumną**, przesuwając dalszą część o jedną pozycję w prawo. Do każdego pozostałego wiersza dopisujemy po jednej luce na końcu. Wszystkie wiersze mają teraz długość $L + 1$ i nadal zawierają swoje oryginalne litery.

Przed mutacją		**Po dodaniu luki w w2 przed kolumną 2**	
w1	A C G	w1	A C G –
w2	A – G	w2	A – – G
w3	A C G	w3	A C G –

Nie dodajemy wszystkim wierszom luki w tej samej kolumnie, bo otrzymalibyśmy wyłącznie kolumnę do usunięcia. Po wstawieniu wykonujemy porządkowanie. Możliwy jest brak rzeczywistej zmiany po porządkowaniu; nie trzeba z tego powodu ponawiać losowania bez końca.

Porządkowanie po operatorze (mutacji/krzyżowania)

Usuwamy tylko te kolumny, w których **wszystkie** wiersze mają lukę. Kolumny prawie pustej, zawierającej choć jedną literę, nie wolno po prostu skasować.

Przed porządkowaniem		**Po usunięciu kolumny 2**	
w1	A – C G	w1	A C G
w2	A – G –	w2	A G –
w3	A – C G	w3	A C G

Dzięki takim usunięciom długość osobnika może także maleć, np. gdy przesunięcia luk doprowadzą do powstania pustej kolumny. Po porządkowaniu liczymy fitness od nowa. Przy modelu otwarcia i przedłużenia usunięta kolumna może wpływać na sposób naliczenia kary w sąsiedztwie.

Jeżeli nie ma żadnej sąsiadującej pary litera–luka, zamiast pierwszej mutacji wykonujemy drugą. Operator nigdy nie zastępuje jedynki zerem bez zachowania tej litery gdzie indziej w dopasowaniu.

B.6. Jak z tego powstaje kolejne pokolenie?

1. **Elityzm.** Z bieżącej populacji wybieramy najlepszy osobnik i kopiujemy go bez zmian do nowej. Przy remisie można wybrać dowolnego zwycięzcę. Ta kopia nie podlega krzyżowaniu ani mutacji. Dzięki temu najlepsze dopasowanie nie spada między pokoleniami.
2. **Tworzenie pozostałych osobników.** Dla każdego wolnego miejsca w nowej populacji wybieramy rodziców turniejami. Z prawdopodobieństwem p_c wykonujemy opisane krzyżowanie; w przeciwnym razie kopiujemy pierwszego rodzica. Następnie, niezależnie od krzyżowania, z prawdopodobieństwem p_m mutujemy powstałe dziecko. Porządkujemy je, sprawdzamy poprawność, obliczamy fitness i umieszczamy w nowej populacji.
3. W pierwszej wersji przyjmujemy wszystkie poprawne dzieci na te wolne miejsca, także gorsze od rodziców. Nie wybieramy automatycznie wyłącznie lepszych dzieci. Ochronę najlepszego rozwiązania zapewnia elita, a nacisk na lepsze rozwiązania - turnieje w kolejnych pokoleniach jako mechanizm selecji.
4. **Wymiana pokolenia.** Nowe pokolenie jest gotowe, gdy znów liczy dokładnie N osobników. Dopiero wtedy zastępuje stare. Podczas jego tworzenia wszystkie turnieje dotyczą jeszcze starej populacji. Nie modyfikujemy oryginalnych rodziców w pamięci, ale potomkowie i elita muszą być odrębnymi kopiami dopasowań.

Przykład pełnego pokolenia

Dla czytelności przyjmijmy $N = 4$ i jednego osobnika elitarnego. Wszystkie poniższe dopasowania dotyczą sekwencji **ACG**, **AG** oraz **ACG**:

Osobnik	Wiersz 1	Wiersz 2	Wiersz 3	SP
A1	**A C G**	**A G –**	**A C G**	−1
A2	**A – C G**	**A G – –**	**A C G –**	−10
A3	**– A C G**	**A – – G**	**A C G –**	−14
A4	**A C – G**	**A – G –**	**– A C G**	−16

A1 przechodzi bez zmian jako elita. Przykładowe losowania dla pozostałych trzech miejsc mogą dać następujący przebieg:

- **Dziecko D1.** Turniej wygrywa **A1**. Krzyżowanie nie zostaje wylosowane, więc tworzymy kopię A1. Mutacja zamienia **G–** na **–G** w drugim wierszu. Dostajemy **ACG** / **A–G** / **ACG** z wynikiem 3.
- **Dziecko D2.** Pierwszym rodzicem zostaje **A2**, drugim **A1**. Krzyżowanie pobiera z **A2** pierwszą kolumnę, czyli trzy litery **A**. Dalsza część **A1** daje dziecko **ACG** / **AG–** / **ACG**. Mutacja nie zostaje wylosowana; wynik wynosi −1. Powstała kopia znanego już dopasowania i w tej podstawowej wersji jest to dozwolone.
- **Dziecko D3.** Pierwszym rodzicem zostaje **A1**, drugim **A2**. Cięcie **A1** wypada po drugiej kolumnie, co daje liczniki (2, 2, 2). Z **A2** pobieramy zatem tylko niewykorzystane końcówki. Dziecko ma wiersze **AC–G** / **AG––** / **ACG–**. Bez mutacji jego SP wynosi −6.

Nowe pokolenie składa się z elity A1 oraz D1, D2 i D3. Ich wyniki to **−1, 3, −1 i −6**. Najlepszy wynik wzrósł z −1 do 3. Nie każde dziecko jest poprawą, a D3 jest nawet gorsze od swojego lepszego, nie stanowi to błędu działania algorytmu genetycznego, raczej efekt uboczny działania.

B.7. Parametry, zakończenie i pierwsze kontrole

Poniższe wartości wystarczają do jednoznacznego ustalenia pierwszej wersji. Nie są obietnicą dobrych wyników dla każdego zestawu danych.

Element	Wariant startowy
Liczność populacji N	40
Liczba pokoleń	100
Selekcja	Turniej dwóch osobników
Elita	1 osobnik kopiowany bez zmian
Prawdopodobieństwo krzyżowania p_c	0,8 na próbę utworzenia dziecka
Prawdopodobieństwo mutacji p_m	0,3 na dziecko
Rodzaj wylosowanej mutacji	Przesunięcie z szansą 0,8; wstawienie z szansą 0,2
Gdy przesunięcie nie jest możliwe	Wstawienie nowej luki

Prawdopodobieństwo mutacji 0,3 oznacza, że każde dziecko ma 30% szans na **jedną mutację**, a nie że zmieniamy 30% jego bitów (!). Dopiero po podjęciu decyzji o mutowaniu wybieramy rodzaj operatora. Prawdopodobieństwa krzyżowania i mutacji nie muszą sumować się do 1: dziecko może przejść oba operatory albo żaden.

Po utworzeniu i ocenieniu populacji początkowej wykonujemy 100 zmian pokolenia. Zwracamy najlepsze pełne dopasowanie wraz z jego SP. Można przechowywać osobny rekord całego uruchomienia; przy prawidłowym elityzmie najlepszy wynik pozostaje również obecny w kolejnych populacjach.

Jak odróżnić błąd od słabej jakości rozwiązania?

Po każdym operatorze sprawdzamy, czy po usunięciu luk wiersz i daje dokładnie oryginalną sekwencję S_i. Kontrola samej liczby liter nie wystarcza dla reprezentacji tekstowej: tyle samo liter może występować w innej kolejności. Sprawdzamy także równe długości wierszy i brak kolumn z samych luk.

Jeżeli test treści nie przechodzi, nie próbujemy naprawiać dziecka przypadkowymi literami ani dopisywaniem luk na końcu. W fazie uruchamiania warto przerwać tę próbę i sprawdzić przykład, na którym się pojawił. Opisane tutaj operatory zachowują poprawność z konstrukcji.

Testy samej punktacji można zacząć od bardzo małych przypadków. Trzy wiersze ACG bez luk powinny dać $3 \cdot 3 = 9$ punktów. Dopasowanie **ACG** / **A–G** / **ACG** powinno dać 3, a **ACG** / **AG–** / **ACG** powinno dać −1. Warto też osobno sprawdzić początek luki, jej przedłużenie, zmianę strony luki i przypadek gap–gap.

Co ulepszać dopiero po uruchomieniu podstawy?

Można tworzyć część populacji z prostych dopasowań wyrównanych do lewej, zamiast losować wszystkie osobniki niezależnie. Można przesuwać całe bloki luk, preferować zgodne cięcia rodziców albo stosować kilka lokalnych prób poprawy po mutacji. Przy dużej liczbie kopii warto zastępować część najsłabszych osobników nowymi losowymi dopasowaniami. Każdą taką zmianę łatwiej ocenić, gdy wersja podstawowa już działa i jej operatory przechodzą testy poprawności.

Przejście do białek. Reprezentacja, krzyżowanie, mutacje i selekcja pozostają takie same. Zmieniamy ocenę par liter na odpowiednie wartości BLOSUM62 oraz wybieramy jeden z opisanych wcześniej modeli luk. Analogicznie w ACO zmienia się ocena kolumny, z której obliczana jest heurystyka. Nie trzeba od nowa projektować sposobu przechowywania dopasowania.

5. Specyfikacja projektu/sprawozdanie

Projekt wykonują zespoły dwuosobowe. Można wybrać algorytm ACO, GA albo inną sensowną metaheurystykę, ale niezależnie od wyboru trzeba pokazać, że zespół rozumie zarówno sam problem MSA, jak i działanie zastosowanej metody optymalizacyjnej.

Wariant podstawowy projektu dotyczy sekwencji DNA i pozwala uzyskać ocenę do 4–4.5. Wariant rozszerzony, pozwalający dostać ocenę do 5.0, powinien dotyczyć sekwencji białkowych, z użyciem sensowniejszej funkcji punktacji, np. BLOSUM62 + kary za luki. Należy zaimplementować program, który:

- przyjmuje zestaw sekwencji wejściowych,
- tworzy dopasowanie wielu sekwencji,
- ocenia jego jakość za pomocą z góry ustalonej funkcji celu,
- poprawia rozwiązanie za pomocą wybranej metaheurystyki,
- pozwala uruchamiać serię eksperymentów dla różnych parametrów.

(nie oczekuję, że algorytm będzie działać **bardzo dobrze** / **dobrze**, najważniejsze jest jednak, aby **działał** :) (poprawnie), był dobrze opisany, dawał się sensownie testować, pokazywał jakąkolwiek poprawę jakości rozwiązania w zadanym czasie). Sprawozdanie powinno mieć dwie główne części.

4.1. Opis problemu i algorytmu

Ta część powinna mieć około 2–3 stron i powinna być napisana tak, abym mógł ocenić, czy grupa naprawdę rozumie, co implementuje :) Nie chodzi o kopiowanie definicji z internetu, tylko o własny, techniczny opis. Powinny się tam znaleźć:

- krótkie wyjaśnienie, czym jest MSA i co oznacza jakość dopasowania,
- definicja użytej funkcji celu,
- sposób reprezentacji rozwiązania w programie,
- opis wybranej metaheurystyki,
- opis najważniejszych operatorów lub kroków algorytmu,
- warunek stopu,
- lista najważniejszych parametrów.

Dla ACO interesujące będą zwłaszcza: liczba mrówek, sposób przechowywania feromonu, parowanie, heurystyka lokalna, reguła wyboru ruchu. Dla GA: reprezentacja chromosomu, sposób tworzenia populacji początkowej, selekcja, krzyżowanie, mutacja, elityzm, itd. Ta część ma pokazać, że zespół rozumie zarówno problem biologiczny/informatyczny, jak i mechanikę zastosowanej metaheurystyki.

4.2. Część eksperymentalna

To ma być najważniejsza część sprawozdania (z punktu widzenia oceny). Interesuje mnie nie tylko to, że program „coś liczy”, ale też czy zespół potrafi przeprowadzić sensowne testy i wyciągać wnioski. W tej części powinny się znaleźć:

- opis zbiorów testowych,
- opis sposobu pomiaru wyników,

- testy parametrów algorytmu,
- testy na różnych klasach instancji,
- wykresy lub tabele,
- komentarz i wnioski.

Główna miara sprawozdania, czyli ocena, leży w eksperymentach i ich prezentacji. Podstawową miarą powinna być wartość funkcji celu, czyli np. końcowy SP-score po ustalonym limicie obliczeniowym. To jest chyba najbardziej naturalne, bo algorytm właśnie tę wartość optymalizuje. Sama końcowa wartość funkcji celu nie zawsze wystarczy, dlatego dobrze, aby w sprawozdaniu pojawiły się także:

- średni wynik końcowy z wielu uruchomień,
- odchylenie standardowe albo przynajmniej informacja o rozrzucie,
- najlepszy wynik,
- średnia poprawa względem rozwiązania początkowego,
- czas działania lub liczba iteracji / ocen funkcji celu.

Dla algorytmów tego typu pojedynczy eksperyment nie jest wiarygodny. Dlatego każdy punkt na wykresie lub każdy wpis w tabeli powinien być liczony na podstawie wielu uruchomień, np. 10 albo 20.

Jak mają wyglądać testy parametrów: w sprawozdaniu powinny znaleźć się 2–3 testy parametrów. Nie trzeba stroić wszystkiego naraz, wystarczy wybrać 2–3 najważniejsze parametry i pokazać, jak wpływają na wynik.

Przykładowo dla **Ant Colony Optimisation**:

- liczba mrówek,
- współczynnik parowania feromonu,
- relacja wpływu feromonu i heurystyki,
- liczba iteracji.

Dla **Genetic Algorithm**:

- liczność populacji,
- prawdopodobieństwo mutacji,
- prawdopodobieństwo krzyżowania,
- poziom elityzmu.

Każdy taki test powinien wyglądać mniej więcej tak:

- wybieramy **jeden parametr**,
- dla niego sprawdzamy kilka wartości, np. 4–6,
- pozostałe parametry zostają stałe w danym teście,
- uruchamiamy algorytm na **zbiorze testowym**, a nie na jednej instancji (!!!),
- każdy eksperyment powtarzamy wielokrotnie,
- raportujemy średni wynik, ewentualnie rozrzut i wyciągamy wnioski.

Bardzo ważne: nie należy robić wykresu „parametr vs wynik” na jednej instancji i wyciągać z tego daleko idących wniosków (o ile jakiekolwiek by można...). To jest za mało wiarygodne. Parametry należy oceniać na małym, ale zróżnicowanym zbiorze instancji.

Przy projektach realizowanych na zwykłych laptopach i w językach typu **C++, Java, C#** dużo lepiej sprawdza się zestaw **krótszych i średnich instancji**, uruchamianych wiele razy, niż jedna ogromna instancja liczona przez wiele godzin. Praktyczne rekomendacje:

4.2.1 Wariant dla DNA.

Do strojenia parametrów:

- **małe instancje**: 5–6 sekwencji, długość 20–40,
- **średnie instancje**: 6–8 sekwencji, długość 40–80,
- **większe instancje**: 8–10 sekwencji, długość 80–150.

Dobrze byłoby przygotować:

- po 5–10 instancji w każdej klasie,
- czyli łącznie około 15–30 instancji do eksperymentów.

Przy takim zakresie rozsądny jest czas rzędu:

- 10–20 s na małą instancję,
- 20–30 s na średnią,
- 30–60 s na większą.

To nie są wartości jedynie słuszne, tylko rekomendacja. Chodzi o to, aby algorytm miał czas wykonać pewną optymalizację, ale żeby cały pakiet testów był jeszcze realny do przeprowadzenia.

4.2.2 Wariant białkowy.

Dla wersji rozszerzonej:

- 5–8 sekwencji,
- długości około 30–100 aminokwasów,
- punktacja typu BLOSUM62 + kara za luki.

4.3 Klasy danych testowych

Jak sensownie różnicować klasy instancji? Samo zwiększanie liczby sekwencji i ich długości to trochę za mało. Dobrze, aby w sprawozdaniu pojawiły się też różne klasy trudności. Polecam dwa proste i (chyba) sensowne pomysły.

4.3.1. Klasy podobieństwa sekwencji

Można generować dane z jednej „sekwencji przodka", a potem wprowadzać mutacje i insercje/delecje z różnym nasileniem. Dzięki temu dostajemy klasy:

- **łatwe**: sekwencje bardzo podobne,
- **średnie**: umiarkowanie podobne,
- **trudne**: mało podobne.

To jest o tyle dobry pomysł, że wtedy można pokazać, jak jakość działania algorytmu spada wraz ze wzrostem „odległości ewolucyjnej" między sekwencjami.

4.3.2. Klasy o różnej strukturze luk

Można utrzymać podobną długość i podobny poziom podobieństwa, ale zmieniać charakter przerw:

- wiele krótkich przerw,
- nieliczne, ale długie bloki przerw.

To też daje bardzo sensowne porównanie, bo dla metaheurystyk przesuwanie i ustawianie długich bloków przerw bywa trudniejsze niż obsługa krótkich, rozproszonych zmian. Opcjonalnie można dołożyć trzecią klasę: sekwencje z **jednym lub dwoma silnie konserwatywnymi motywami** otoczonymi „szumem". Taka klasa jest ciekawa, bo pozwala zobaczyć, czy algorytm potrafi dobrze ustawiać wspólne bloki, a nie tylko poprawiać wynik globalny przypadkowymi przesunięciami.

4.4 Jak powinny wyglądać wykresy i tabele

Chciałbym proste i **czytelne** :) wykresy i tabele. Nie musi być ich dużo, ale powinny być sensowne. Przykłady:

- wartość funkcji celu w zależności od parametru,
- średnia poprawa względem rozwiązania początkowego,
- jakość końcowa dla różnych klas instancji,
- przebieg zbieżności w czasie lub po iteracjach,
- tabela średni wynik / najlepszy wynik / odchylenie / czas.

Dobrze, jeśli dla każdego wykresu będzie jasno napisane: jaki był zbiór instancji, ile było uruchomień, co dokładnie przedstawia oś Y, czy pokazywana jest średnia, mediana czy najlepszy wynik. Co będzie miało największy wpływ na ocenę:

1. poprawność i kompletność implementacji,
2. zrozumienie problemu i algorytmu,
3. jakość części eksperymentalnej,
4. sensowność wniosków.

Nie będę oceniał projektu po tym, czy usykano wysoki wynik funkcji celu. Ważniejsze jest, czy:

- dobrze zdefiniowano problem,
- zaprojektowano reprezentację rozwiązania,
- poprawnie zaimplementowano metaheurystykę,
- przeprowadzono wiarygodne testy,
- uzasadniono swoje decyzje.

W praktyce:

- na ocenę **3–3.5** wystarczy działający program i podstawowy raport,
- na **4–4.5** potrzebne będą już sensowne testy parametrów i poprawna analiza wyników,
- na **5.0** oczekuję wersji białkowej albo wyraźnie ambitniejszej metodologii eksperymentów, najlepiej z bardzo dobrą analizą oraz starannym porównaniem klas instancji.

Lepiej zrobić porządnie mniejszy zakres niż próbować rozwiązywać bardzo duże instancje bez sensownej analizy.

4.5 Inne metaheurystyki

Dozwolone, ale wtedy też powinny spełniać ogólne wymagania tutaj opisane, które trzeba dostosować do wybranego algorytmu.