# PRZEDMIOT OBIERALNY: BIOINFORMATYKA, LABORATORIUM

**Wersja pliku: v 1.5.1, 20 marca 2025**
**Autor:** dr inż. Marcin Radom

## PROJEKT – ZALICZENIE - ZASADY

- Praca w zespole dwuosobowym.
- Sprawozdanie z projektu:
  - Opis projektu: problem, algorytmy, testy.
  - Wyniki testów w formie tabel i wykresów.
  - Termin: koniec semestru (informacje na stronie).
- Sprawozdania można wysyłać mailem, należy też dostarczyć pełny kod projektu (też w mailu, jeżeli są problemy z przesłaniem, można wysłać link do repo).
- Język programowania: (prawie) dowolny, preferowane: C++, C#, Java, Python.

## PROBLEMY

Problem rekonstrukcji DNA ze spektrum pochodzącym z eksperymentu SBH może mieć wiele wersji i nie chodzi tutaj tylko o błędy pozytywne i negatywne. Poniżej trzy proponowane wersje problemów do wyboru, z komentarzem:

| **Nazwa problemu:** | **Dane:** |
|---|---|
| **Problem klasycznego SBH z błędami negatywnymi i pozytywnymi** | • Długość DNA (n), długość oligonukleotydów (k), wierzchołek początkowy (oligonukleotyd o długości k).<br>• Informacja o rodzaju błędów: pozytywne / negatywne / oba rodzaje. |
| ~~**SBH z informacją o położeniu + błędy**~~ (stare, w 2025 roku nie obowiązuje) | ~~• Wszystkie dane z klasycznego SBH.~~<br>~~• Z każdym oligonukleotydem związana jest informacja o tym, gdzie w DNA znajduje się dany fragment – w formie przedziału (x, y) gdzie x ≥ 0, y ≤ n.~~ |
| **SBH z informacją o powtórzeniach + błędy** | • Wszystkie dane z klasycznego SBH.<br>• Z każdym oligonukleotydem związana jest informacja o tym, czy występuje w DNA raz czy więcej razy (dla błędów pozytywnych = 1).<br>• Można przyjąć, że z każdym elementem spektrum jest związana informacja: 1, 2 lub *. 1 lub 2 oznacza, że dokładnie tyle razy dany element się powtarza w DNA. * oznacza, że występuje 3 lub więcej razy. Dla elementów będących błędami pozytywnymi – do nich dopisujemy „1”, czyli info że tylko raz każdy występuje (inaczej nie ma to sensu biologicznego). |

Podzielmy to na następujące formuły problemów do wyboru dla grup laboratoryjnych:

1. Klasyczny problem SBH z błędami negatywnymi wynikającymi z powtórzeń oraz pozytywnymi.
2. ~~Problem SBH z informacją o położeniu oraz wszystkimi rodzajami błędów.~~
3. Problem SBH z informacją o powtórzeniach oraz wszystkimi rodzajami błędów.

Algorytmy dla problemu nr 2 i 3 mają przynajmniej hipotetycznie szanse działać lepiej niż dla problemu nr 1 ze względu na dodatkowo informacje zawarte w danych wejściowych (instancji problemu SBH). Ta dodatkowa informacja, nawet w formie: albo raz albo dwa razy albo wiele razy (powtórzenia) pozwala przynajmniej teoretycznie, efektywniej rekonstruować sekwencję. W tym przypadku wiadomo, że wszystkie elementy spektrum muszą zostać użyte (a przynajmniej powinny – o tym dalej, w różnicach pomiędzy algorytmami do zastosowania).

## DANE WEJŚCIOWE

Aby mieć lepsze wyobrażenie o skali problemu i konieczności wyboru odpowiedniego algorytmu (i jego dobrej implementacji) ustalmy tutaj „typową" instancję (konkretną liczbową reprezentację) problemu.

Typowa instancja do badania powinna charakteryzować się:

- DNA do rekonstrukcji ma długość **przynajmniej** 300 i mniej niż 1000. Ale i tak wszystko powyżej 700 dla problemu 1 to już ekstrema i nie należy oczekiwać dobrych wyników rekonstrukcji.
- Jeżeli chodzi o długość oligonukleotydów w czasie symulowania eksperymentu SBH (czyli generowania fragmentów spektrum), to stosujemy przedział: przynajmniej 7, nie więcej niż 10. W przypadku prawdziwych mikromacierzy do SBH oznaczałoby to odpowiednio: $4^7$ = 31768 komórek mikromacierzy DNA, $4^9$ = 262144 komórek, $4^{10}$ = 1048576. Jak się można domyślić, większych mikrochipów się raczej nie stosowało, bo upchnięcie na małej płytce ponad miliona komórek każda z innym oligonukleotydem przestawało być zabawne, a mówiąc poważniej: opłacalne / realistycznie.
- Informacja, ile jest błędów i jakich. Algorytm ma prawo znać dokładną liczbę błędów negatywnych i dokładną liczbę błędów pozytywnych. W przypadku istnienia w zbiorze błędów negatywnych *błędów negatywnych wynikających z powtórzeń* algorytm może mieć o tym informację (że są takie), ale oczywiście nie wie które to i ile ich jest.
    - Chyba, że w instancji nie miało być błędów negatywnych które polegają na usuwaniu pewnych elementów spektrum, ale są tylko te wynikające z powtórzeń. To wtedy wiemy, ile ich jest, będzie to różnica pomiędzy n-k+1 a licznością spektrum. Oczywiście zakładamy, że nie ma wtedy błędów pozytywnych. Bo jeśli są, to wzór n-k+1 już do niczego się nie przydaje – wiemy, że są błędy wynikające z powtórzeń, ale znowu nie wiadomo, ile ich jest, bo doszły fałszywki, czyli błędy pozytywne.

**Błędy pozytywne**

Teraz dość ważny wątek. Jak tworzyć SENSOWNE błędy pozytywne? Pierwsze podejście, na które każdy może wpaść, to po prostu wygenerowanie odpowiedniej liczby fałszywych oligonukleotydów o długości k (tylu, ile ma być błędów pozytywnych) i wrzucenie ich do spektrum. Problem z tym podejściem jest taki, że tak wygenerowane fałszywki będą mieć nikły wpływ na rekonstrukcję. Dlaczego? Ano dlatego, że bardzo rzadko będą tworzyć łuki z prawdziwymi elementami spektrum, bo nie będą się z nimi nakładać końcówkami. A już niemal na pewno będą mieć bardzo mało, jeżeli w ogóle, łuki o wadze „1". W końcu zostały wygenerowane sztucznie, to czego oczekiwać?

Dlatego też błędy pozytywne powinny być tworzone sensowniej, to jest tak, aby wykazywały pewne podobieństwo z już istniejącymi elementami spektrum. Jak to zrobić?

1. Algorytm bierze jakiś element ze spektrum, taki, który nie jest sam błędem pozytywnym. Dla przykładu załóżmy, że będzie to np. AGTCGTCG (czyli elementy w spektrum mają k=8 liter).
2. Następnie tworzymy dwie kopie tego elementu. Kopie! Czyli oryginał zostawiamy w spokoju w spektrum skąd go skopiowaliśmy, służył nam on tu tylko jako *matryca* do stworzenia na jego bazie dwóch błędów pozytywnych w punkcie 3.
3. W jednej kopii zmienia ostatni nukleotyd oraz (osobno w drugiej kopii) środkowy.
    a. Dla pierwszego: mając AGTCGTC**G**, zamieniamy to ostatnie **G** na coś innego, na **A** lub **C** lub **T**. W przykładzie dla AGTCGTCG powiedzmy, że powstał na jego bazie AGTCGTC**A** jako pozytywny błąd.
    b. Druga kopia: bierzemy oryginalny AGT**CG**TCG i zmieniamy w nim **albo** środkowe **C** **albo** **G** na jakąś inną literę z alfebetu {A, C, G, T}. W tym przypadku musimy się zdecydować, czy zmienimy środowe **C** lub **G**, **ponieważ k=8 jest parzyste**. Gdyby np. było k=9, czyli nieparzysta liczba liter, a mielibyśmy element CAGT**C**GTCG (ze spektrum, ta *matryca* do tworzenia błędów), to wiadomo, że środkowa litera jest wtedy tylko jedna, jest to **C**, i to ją musimy zmienić na **A** lub **G** lub **T** żeby powstał drugi błąd pozytywny.
4. Schemat z 1-3 powtarzamy tak długo, aż otrzymamy odpowiednią liczbę błędów pozytywnych. Czyli jak już mamy dwa pierwsze błędy pozytywne z przykładu, to losowo wybieramy kolejny (inny) element z oryginalnego spektrum DNA, tworzymy jego dwie kopie, itd. Jak opisałem w puntach 2 i 3 powyżej.

## ALGORYTMY

Biorąc to wszystko pod uwagę widzimy, że graf będzie mieć kilkaset wierzchołków i całą masę łuków z wagami 1, 2 i 3. W ogólności próba stosowania algorytmu dokładnego (i/lub brute force) skończy się tym, że dla niewielkich fragmentów DNA (< 300nt) i największych bibliotek oligonukleotydów (czyli k=10) dostaniemy zapewne rozwiązanie przy założeniu, że w spektrum nie będzie błędów, będzie ich naprawdę naprawdę mało, albo będą to tylko błędy pozytywne (one, jeżeli nie towarzyszą im błędy negatywne, nie mają aż tak złego wpływu na jakość rekonstrukcji, o ile oczywiście nie będzie ich naprawdę dużo).

Pomysł na algorytm (jeden z tysięcy albo więcej możliwych, więc jest tu ogromne pole dla waszej inwencji twórczej) który podaję na laboratorium jest następujący (z grubsza):

1. Mając już graf i oznaczony wierzchołek początkowy, przechodzimy łukami o wadze 1 z wierzchołka do wierzchołka tak daleko jak się da. A precyzyjniej: zatrzymujemy się z tym szybkim i radosnym odwiedzaniem wierzchołków, gdy:
    a. Nie ma w wierzchołku, do którego dotarliśmy, żadnego łuku o wadze 1.
    b. Jest jakiś, ale prowadzi do wierzchołka już odwiedzonego.
        i. ***** który jednak możemy odwiedzić takim łukiem o wadze 1 **tylko jeżeli wybraliśmy problem nr 3**, tj. ten z informacją o powtórzeniach i wiemy, że ten akurat nowy wierzchołek docelowy **może być odwiedzony więcej niż raz**. To wtedy wciąż można kontynuować podróż takim łukiem o wadze 1. W innym wypadku: stop i patrzymy dalej na ten pseudokod, który właśnie czytacie.
    c. Na potrzeby dalszego opisu nazwijmy sobie taki wierzchołek **wierzchołkiem X**, tj. ten w którym się właśnie zatrzymaliśmy, bo z jakiegoś powodu brak dobrych łuków o wadze 1.
2. Jak się już zatrzymaliśmy z powodów opisanych w punkcie 1 powyżej, to wtedy:
    a. Robimy listę 5-10 (jest to pewne zmienna, parametr naszego algorytmu) wierzchołków grafu spełniających dwa kryteria:

i. Po pierwsze dany wierzchołek nie został jeszcze odwiedzony.
ii. Po drugie jego najbliższe sąsiedztwo, czyli wszystkie wierzchołki do których prowadzą z niego łuki (o dowolnej wadze: 1 , 2 lub 3, nieistotne) też nie zostały jeszcze odwiedzone.
iii. Jak napisałem w podpunkcie a) powyżej, takich wierzchołków szukamy kilka, np. 5 do 10. Nie trzeba skanować tak w tym kroku całego grafu, jak już znaleźliśmy kilka spełniających powyższe kryteria.

b. To teraz mamy następujące dane: **wierzchołek X** w którym się zatrzymaliśmy (punkt 1 tego pseudokodu) oraz kilka potencjalnych wierzchołków w grafie w których: *„fajnie byłoby się znaleźć, ponieważ wtedy moglibyśmy znowu iść ścieżką zbudowaną z łuków o wadze 1”*. Te wierzchołki nazwę tutaj wspólnie: **zbiorem Y**.

c. Uruchomiamy algorytm pomocniczy (np. alg. Dijkstry czy jakiekolwiek inny podobny) i każemy mu odnaleźć ścieżki od **wierzchołka X** do każdego z wierzchołków ze **zbioru Y**.
i. Wybieramy **najkrótszą** ze znalezionych ścieżek, aktualizujemy naszą ścieżkę od wierzchołka początkowego grafu o właśnie znalezioną podścieżkę z **X** do której z wierzchołków ze **zbioru Y**, i wracamy do punktu 1 pseudokodu, kontynuując podróż łukami o wadze 1.
ii. Jeżeli ktoś tu wciąż nie rozumie, przykład. Załóżmy, że mamy ścieżkę (numery to indeksy wierzchołków): 1 (wierzchołek startowy grafu) -> 56 -> 13 -> 178 -> 201 -> 189 -> **X_199** (załóżmy, że X ma numer 199). Do tego momentu wszystkie przejścia są po łukach o wadze 1. Teraz, Dijkstra/inny algorytm pomocniczy znalazł nam coś takiego: **X_199** -> 34 -> 301 -> 145 -> 187 -> **wierzchołek_ze_zbioru_Y_o_indeksie_96.** To teraz nasza pełna ścieżka wygląda tak: **1** -> 56 -> 13 -> 178 -> 201 -> 189 -> **X_199** -> 34 -> 301 -> 145 -> 187 -> **Y_96**. Od 96-ego znowu idziemy jedynkami jak dalego się da.
iii. Należy tu zaznaczyć, że to właśnie podświeżka **X_199** -> 34 -> 301 -> 145 -> 187 -> **Y_96** ma pewnie w sobie łuki o wagach 2 lub 3 (choć pewnie też i jakieś 1, może). Czyli w skrócie: używanie łuków o wyższych wagach zostawiamy algorytmowi pomocniczemu, szukania ścieżki od **X** do **Y**. I tak by to pewnie zrobił, w końcu zdefiniowaliśmy **wierzchołek X** jako taki, w których właśnie nie ma łuków 1 (punkt 1 algorytmu).

Kilka ważnych uwag praktycznych:

1. Szukanie wierzchołków do **zbioru Y** (punkt 2 algorytmu, czyli docelowych z problematycznego **wierzchołka X**) nie można będzie zapewne robić w nieskończoność, tj. do samego końca poszukiwania ścieżki w grafie. Na początku takich wierzchołków będzie bardzo dużo, ale im dłuższa nasza ścieżka, tym trudniej będzie je znaleźć. Trzeba tutaj jakoś to odpowiednie oprogramować, np. zrobić algorytm, który, jeżeli nie znajdzie już żadnego potencjalnego wierzchołka Y, to musi spróbować dokończyć budowę pełnej ścieżki w grafie innym sposobem.
    a. Jeżeli do końca zostało jeszcze z kilkadziesiąt nukleotydów, to warto by tu pomyśleć o jakimś inteligentniejszym szukaniu.
    b. Jeżeli jednak do końca pełnej rekonstrukcji pozostało np. kilkanaście nukleotydów, to można zaimplementować to tak, że niech nasz algorytm idzie gdziekolwiek, byle do przodu, nawet powtarzając już odwiedzone wierzchołki, byle szybko dokończył rekonstrukcję DNA, a nie marnował minuty na kombinowanie jak tu dotrzeć do końca.
2. Cały algorytm „chodzi” po grafie, tj. interesują go tylko wierzchołki i łuki pomiędzy nimi. A mówiąc precyzyjnie: na etapie szukania ścieżki **algorytmu nie powinny obchodzić literki A, C, G , T**, tylko same

wagi łuków. Warunek stopu jest więc taki, że jeżeli na przykład **n = 100** (docelowa długość DNA do znalezienia), **k = 7**, to znaczy, że znamy pierwsze siedem nukleotydów a musimy zrekonstruować w sumie **100**. Jeżeli od wierzchołka startowego poszliśmy łukami o wagach kolejno: 1, 1, 1, 1, 2, 1, 1, 2, 3, 1, 1, to wtedy algorytm wie, że zrekonstruował dokładnie 7+1+1+1+1+2+1+1+2+3+1+1 = 22 nukleotydy. Do końca zostało 78. **NIE OBCHODZI GO NA TYM ETAPIE Z JAKICH LITER SKŁADA SIĘ DNA BĘDĄCE ODPOWIEDNIKIEM AKTUALNIE ZREKONSTRUOWANEJ ŚCIEŻKI.** Czyli nie używamy na tym etapie żadnych substring() czy innych czasochłonnych funkcji, tylko chodzimy od wierzchołka do wierzchołka łukami o odpowiednich wagach. Dopiero gdy mamy już całą ścieżkę, tj. z sumy wszystkich użytych na ścieżce łuków, plus początkowe k liter, doszliśmy do wartości n, to wtedy algorytm wie, że skończył swoje zadanie. I wtedy dopiero zmienia całą znalezioną ścieżkę na kod A, C, G, T, nie wcześniej!

3. Kolejna sprawa, na bazie pytań które zadawano mi w poprzednich latach. Załóżmy, że z jakiegoś powodu nasz algorytm „zakleszczy" się w pewnym wierzchołku. Np. nie ma z niego łuków wyjściowych. Gdyby pojawił się taki, to może i byłby to prawdziwy wierzchołek/oligonukleotyd końcowy DNA, ale po pierwsze tego nie wiemy (może być takich wiele w grafie), a po drugie, zapewne nasz algorytm wszedł do tego wierzchołka zanim zrekonstruował DNA o długości n. Co dalej robić? Zacznijmy od tego, czego nie robić: funkcja w projekcie pozwalająca się wycofać po własnych śladach zazwyczaj jest fatalnym pomysłem, bo algorytm często może z niej korzystać i przez to nie kończy w ogóle pracy nad jedną ścieżką, bo co chwila gdzieś się wycofuje. No to co zrobić w takim wypadku?
    a. Proponuję stworzyć „wirtualny łuk" z tego wierzchołka do jakiegoś innego. Załóżmy, że w naszym grafie są łuki o wagach 1, 2 i 3, ale żadnego nie ma w tym wierzchołku, którego problem rozważamy. Wtedy możemy spróbować na potrzeby tego wierzchołka stworzyć jakiś łuk o wyższej wadze, 4 albo jeszcze większej. Po prostu np. dla wagi 4, pobieramy ostatnie 4 nukleotydy tego wierzchołka i sprawdzamy, czy gdzieś w grafie jest (najlepiej nieodwiedzony jeszcze) wierzchołek, który zaczyna się czterema nukleotydami takimi, jakimi ten w którym się zatrzymaliśmy się kończy. Jak taki znajdziemy, to „uciekamy" z tego zakleszczonego wierzchołka takim tymczasowym łukiem o wasze 4 i kontynuujemy podróż po grafie. Jak nie będzie o wadze 4, to może o wadze 5. W końcu jakiś znajdziemy, i nastąpi „ucieczka do przodu" zamiast bardzo problematycznej zazwyczaj procedury zawracania po własnych śladach na ścieżce w nadziei, że wcześniej znajdziemy jakieś rozgałęzienie na łukach i pójdziemy inną drogą.

Potencjalne lektury wstępne:

**Strona prof. Marty Kasprzak:**

http://www.cs.put.poznan.pl/mkasprzak/bio/bio.html

A dokładniej wykład Sekwencjonowanie cz. 2 ( http://www.cs.put.poznan.pl/mkasprzak/bio/Bioinfo-wA-slajdy.pdf ) oraz materiały uzupełniające do niego: ( http://www.cs.put.poznan.pl/mkasprzak/bio/Bioinfo-wB-uzupelnienie.pdf )

**Dr inż. Kamil Kwarciak, na stronie:**

https://docplayer.pl/28865443-Modele-grafowe-i-algorytmy-dla-klasycznego-problemu-sekwencjonowania-dna-przez-hybrydyzacje-oraz-dla-jego-odmiany-z-informacja-o-powtorzeniach.html

Jest dostępny doktorat Kamila Kwarciaka, oczywiście nie trzeba czytać całego (nawet tego nie sugeruję) ale szczególnej uwadze polecam:

Rozdział 6-6.1 (bez 6.2 – izotermiczne SBH)
Rozdział 7 – sekwencjonowanie klasyczne
Rozdział 8 – algorytm dla sekwencjonowania SBH z informacją o powtórzeniach

**Dodatkowe materiały:**

http://bix.ucsd.edu/bioalgorithms/presentations/Ch08_GraphsDNAseq.pdf

https://www.scribd.com/document/356206930/Ch08-GraphsDNAseq