Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем

Розробка серверного біокалькулятора на мові Java

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник курсової роботи
Старший викладач Борозенний С.О.

_____
(підпис)
“____” _____ 2022 р

Роботу виконав студент БП ІПЗ-4
Рожко А. С.
“____” _____ 2022 р

Київ 2022

Тема: **Розробка серверного біокалькулятора на мові Java.**

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Проходження курсу Biology Meets Programming: Bioinformatics for Beginners на платформах Coursera та Stepik	19.07.2021	https://www.coursera.org/learn/bioinformatics/home/welcome
2.	Проходження курсу Genome Sequencing (Bioinformatics II) на платформах Coursera та Stepik	23.08.2021	https://www.coursera.org/learn/genome-sequencing/home/welcome
3.	Переклад алгоритмів з мови Python 3 на Java 8	27.12.2021	
4.	Написання текстової частини роботи	16.01.2022	
5.	Створення серверного застосунку для надання доступу до алгоритмів	23.01.2022	
6.	Створення Unit-тестів для алгоритмів, що використовують випадкові числа і Postman тестів для решти	30.01.2022	
7.	Створення презентації доповіді	30.01.2022	
8.	Захист курсової роботи		

Студент _____

Курівник _____

“_____”_____

Зміст

Вступ

Ще з початку навчання на Факультеті інформатики, з перших лабораторних робіт я почав задавати собі питання: «Навіщо ці завдання?», - я розумів, що для початку завдання на кшталт «підрахувати суму ряду з точністю до …», чи зробити гру про літак, або знайти рекурентне співвідношення – корисні вправи, але мені хотілось чогось більш прикладного, більшого, ніж математика заради математики, чи програмування заради програмування. Хотілось поєднання програмуванні з іншою предметною областю, бажано природничою. На першому ж курсі це бажання вилилося у використання різницевих рівнянь для симуляції зміни кількості хижих і нехижих риб за моделлю хижак-жертва при створенні гри “Stern fisherman”, на літній практиці. На другому курсі мене вгамовували Основи комп’ютерних алгоритмів та Інформаційний пошук. На третьому я подався у вивчення Spring Boot та пов’язаних технологій мотивуючись їх актуальністю та корисністю. Коли наприкінці третього курсу мій потік отримав звістку про те, що замість курсової роботи і випускних екзаменів у нас буде дипломна робота, я подумав: «Це знак, час вчити біологію і цікавитися тим, що таке біоінформатика». На той момент я зрозумів, що саме біоінформатика має стати темою моєї наступної наукової роботи, мета якої – ознайомитися з обчислювальними методами роботи з біологічними сутностями.

Розуміючи, що жодного викладача з Факультету природничих наук я не знаю, а навіть якби знав, скоріше за все наукового керівника з іншого факультету мені взяти було б заборонено, я усвідомив, що мій шлях мав починатися на платформах на кшталт Coursera. Там я знайшов і пройшов два вступні курси: «Biology Meets Programming: Bioinformatics for Beginners» та «Genome Sequencing (Bioinformatics II)», хоч більшу частину часу я проводив на платформі Stepik, куди вели посилання з Coursera. Там розміщувалися як теоретичні відомості та опис алгоритмів, які лягли в основу моєї наукової роботи, так і практичні завдання, які автоматизовано перевірялися і стали запорукою правильності моєї імплементації зазначених алгоритмів.

Від початку масштаб дипломної роботи планувався таким:

- Імплементувати запропоновані алгоритми
- Створити сервіс доступу до них, який би передбачав розподіл алгоритмів на класи за складністю, доступ до яких мав би надаватися відповідно до підписки зареєстрованого користувача
- Графічний інтерфейс для зручної роботи користувачів
- Додатковий функціонал, як от: збереження попередніх обчислень, фонові обчислення для великих вхідних даних, нотифікація на пошту тощо

На жаль, в процесі написання практичної частини виявилося, що дипломна робота не прийде на заміну курсовій та випускним екзаменам, тому обсяг даної роботи довелося скоротити, залишивши перший пункт, мінімальну версію другого і додавши запити-тести в Postman замість останніх двох.

Таким чином акцент курсової роботи змістився з розробки продукту на вивчення алгоритмів опрацювання ДНК та сутностей, що з неї складаються. Відтак, актуальність даної роботи стає очевидна у час, коли вакцини на основі мРНК є найпоширенішими і найбільш визнаними у боротьбі з коронавірусною хворобою. Крім того, коли як не на четвертому курсі студент може припинити хвилюватися про брак знань і просто займатися тим, що йому подобається.

Розділ 1. Опис алгоритмів пошуку регуляторних мотивів

До складу ДНК входять фактори транскрипції, або регуляторні мотиви, знаходження яких також має цінність. “Regulatory motifs are short nucleotide sequences typically upstream of genes that are used to control the expression of genes, dictating under which conditions a gene will be turned on or off. Direct identification of regulatory elements is more challenging than that of genes. Such elements are typically short (6-15 bp), **tolerate some degree of sequence variation** and follow few known rules. To date, the majority have been found by experimentation, such as systematic mutation of individual promoter regions; the process is laborious and unsuited for genome-scale analysis.

Computational analysis of single genomes has been successfully used to identify regulatory elements associated with known sets of related genes.” [1] bp – “A base pair is two chemical bases bonded to one another forming a «rung of the DNA ladder.»” [2]

Оскільки фактори транскрипції допускають мінливість у своєму складі, точного способу їх знайти немає. Розгляньмо приклад уривків ДНК, які позначають регуляторні мотиви.

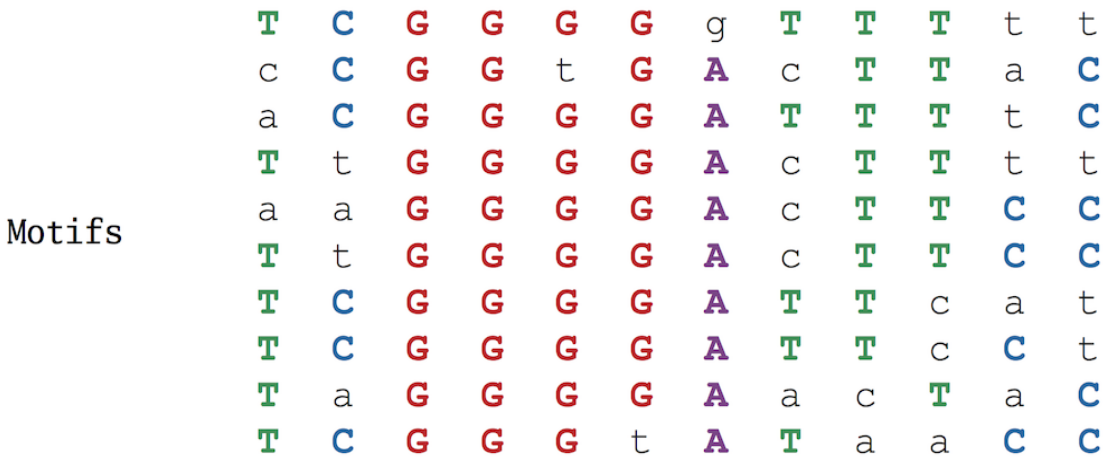

Рисунок 1 - мотив захований в уривках ДНК [3]

Для того, щоб боротися із очевидно шумними даними використовуватимемо профілювання уривків ДНК. Кожному нуклеотиду присвоюватимемо ймовірність його правильності як відношення суми його входжень у конкретному стовпчику до кількості рядків. Тобто для 1 колонки результат буде такий: A – 2/10, C – 1/10, G – 0/10, T – 7/10. У подальших обчисленнях такі профілі будуть використовуватись для знаходження найімовірніше правильних мотивів, для цього буде обчислюватися добуток ймовірностей їх нуклеотидів. Очевидно, що одного точно неправильного нуклеотиду буде достатньо, щоб увесь k-мер сприймався як повністю неправильний, щоб цього уникнути додаватимемо один (1) при профілюванні. Таким чином наведений вище

приклад виглядатиме так: A – 3/14, C – 2/14, G – 1/14, T – 8/14. Завдяки цій зміні майже правильний регуляторний мотив матиме набагато вищий бал, ніж повністю неправильний.

```
01    private static Map<Character, List<Double>> profileWithPseudocounts(List<String>
motifs) {
02        Map<Character, List<Double>> countMat = new HashMap<>();
03        int k = motifs.get(0).length();
04        for (char symbol : "ACGT".toCharArray()) {
05            countMat.put(symbol, new ArrayList<>());
06            for (int i = 0; i < k; i++)
07                countMat.get(symbol).add(1.0);
08        }
09        for (String motif : motifs)
10            for (int j = 0; j < k; j++) {
11                char symbol = motif.charAt(j);
12                countMat.get(symbol).set(j, countMat.get(symbol).get(j) + 1);
13            }
14
15        for (char nucKey : countMat.keySet())
16            for (int i = 0; i < motifs.get(0).length(); i++)
17                countMat.get(nucKey).set(i, countMat.get(nucKey).get(i) / (motifs.size()
+ 4));
18        return countMat;
19    }
```

Для знаходження найпопулярніших нуклеотидів і побудову найімовірнішого правильного регуляторного мотиву також знадобиться метод consensus, його імплементація винесена у додаток А.

Для порівняння проміжних результатів роботи алгоритмів, що будуть представлені нижче, знадобиться метод score, який хоч і виконує важливу роль, має тривіальну імплементацію, яку наведено в додатку А.

1.1 Жадібний алгоритм пошуку мотивів (greedyMotifSearch)

Розгляньмо перший, жадібний алгоритм пошуку регуляторних мотивів імовірної довжини k.

```
01    public static List<String> greedyMotifSearch(String[] dna, int k) {
02        List<String> bestMotifs = new ArrayList<>();
03        for (String strand : dna)
04            bestMotifs.add(strand.substring(0, k));
05        int n = dna[0].length();
06        for (int i = 0; i < n - k + 1; i++) {
07            List<String> motifs = new ArrayList<>();
08            motifs.add(dna[0].substring(i, i + k));
09            for (int j = 1; j < dna.length; j++) {
10                Map<Character, List<Double>> profile =
profileWithPseudocounts(motifs.subList(0, j));
11                motifs.add(profileMostProbableKmer(dna[j], k, profile));
12            }
13            if (score(motifs) < score(bestMotifs)) bestMotifs = motifs;
14        }
15        return bestMotifs;
16    }
```

Це ітеративний алгоритм, що поступово покращує свої проміжні результати, все ж із чогось треба почати, тому починаємо з матриці k-мерів утворених з перших k нуклеотидів переданих уривків. Цикл, що починається на шостому (6) рядку

шукає кращі варіанти bestMotifs. У тілі цього циклу спочатку перебираються усі k-мери першого рядка переданої матриці, записуються в тимчасову змінну motifs. Починаючи з дев'ятого (9) рядка з решти рядків обираються кращі мотиви у методі findMostProbableKmer, куди передається заздалегідь обчислений профіль з попередньо підібраних мотивів. Якщо отримані в результаті проміжних обчислень мотиви мають менше розбіжностей, вони записуються у bestMotifs. Після опрацювання усіх k-мерів першого рядка у даній змінній знаходяться кращі обчислені мотиви з усіх розглянутих варіантів, вони і повертаються.

Імплементація findMostProbableKmer проста: використовуючи переданий профіль цей метод обчислює імовірності правильності кожного k-мера, що міститься в переданому рядку і повертає той, що має найбільшу ймовірність. Код тривіальний, наведений у додатку А.

1.2 Алгоритм випадкового пошуку мотивів (randomizedMotifSearch)

Тепер розгляньмо алгоритм пошуку регуляторних мотивів, що використовує випадковості.

```
01    public static List<String> randomizedMotifSearch(String[] dna, int k) {
02        List<String> bestMotifs = randomMotifs(dna, k);
03        while (true) {
04            Map<Character, List<Double>> profile = profileWithPseudocounts(bestMotifs);
05            List<String> m = findMotifs(profile, dna);
06            if (score(m) > score(bestMotifs))
07                bestMotifs = m;
08            else return bestMotifs;
09        }
10    }
```

Починаючи свою роботу зі знаходження випадкових мотивів цей алгоритм надалі їх використовує для знаходження кращих варіантів. Пошук триває доти, доки проміжні результати покращуються.

```
01    private static List<String> randomMotifs(String[] dna, int k) {
02        List<String> motifs = new ArrayList<>();
03        for (String segment : dna) {
04            int start = random.nextInt(dna[0].length() - k + 1);
05            motifs.add(segment.substring(start, start + k));
06        }
07        int score = dna.length * k;
08        while (true) {
09            Map<Character, List<Double>> profile = profileWithPseudocounts(motifs);
10            motifs = findMotifs(profile, dna);
11            int newScore = score(motifs);
12            if (newScore < score) score = newScore;
13            else break;
14        }
15        return motifs;
16    }
```

```
1    private static List<String> findMotifs(Map<Character, List<Double>> profile, String[]
dna) {
2        List<String> motifs = new ArrayList<>();
3        for (String strand : dna)
4            motifs.add(profileMostProbableKmer(strand, profile.get('A').size(), profile));
```

```
5         return motifs;
6     }
```

Як видно у методі randomMotifs, випадковий вибір мотивів відбувається лише раз, але вони далі використовуються для профілювання і вибору подальших, кращих з обчислювальної точки зору варіантів. Все ж випадковість, яка дала початок такому пошуку лишається і впливає на напрям розвитку кінцевого результату. Цей напрям може бути невдалим, але через дешевизну алгоритму randomizedMotifSearch його можна виконувати багато разів обираючи кращі результати.

1.3 Поміркований алгоритм випадкового пошуку мотивів (gibbsSampler)

Останній запропонований алгоритм можна вважати логічним розвитком randomizedMotifSearch, який замість того щоб змінювати усі k-мери на кожному кроці випадковим чином обрає який можна було б змінити протягом n ітерацій:

```
01    public static List<String> gibbsSampler(String[] dna, int k, int n) {
02        List<String> bestMotifs = randomMotifs(dna, k);
03        for (int i = 0; i < n; i++) {
04            int searchStrIdx = random.nextInt(dna.length);
05            List<String> investedMotifs = new ArrayList<>(Arrays.asList(dna));
06            investedMotifs.remove(searchStrIdx);
07            Map<Character, List<Double>> profile =
profileWithPseudocounts(investedMotifs);
08            String kmer = profileGeneratedStringDna(dna[searchStrIdx], profile, k);
09            List<String> updMotifs = new ArrayList<>(bestMotifs);
10            updMotifs.set(searchStrIdx, kmer);
11            if (score(updMotifs) < score(bestMotifs)) bestMotifs = updMotifs;
12        }
13        return bestMotifs;
14    }
```

На п’ятому рядку змінна investedMotifs ініціюється як копія вхідних уривків ДНК, потім за випадково обраним індексом прибирається рядок з отриманої матриці-копії. Оскільки мотив у цьому рядку хочемо замінити, брати його до уваги при профілюванні не будемо. Отриманий з profileGeneratedStringDna k-мер перетирає старий мотив у копії updMotifs. На одинадцятому рядку видно, що лише у випадку успішної заміни оновлений список bestMotifs буде оновлений.

Розгляньмо метод підбору k-меру:

```
1    private static String profileGeneratedStringDna(String text, Map<Character,
List<Double>> profile, int k) {
2        Map<String, Double> probabilities = new HashMap<>();
3        for (int i = 0; i < text.length() - k + 1; i++) {
4            String dnaStr = text.substring(i, i + k);
5            probabilities.put(dnaStr, pr(dnaStr, profile));
6        }
7        return weightedDie(normalize(probabilities));
8    }
```

Спочатку у циклі для кожного варіанту обчислюється імовірність його правильності. Потім результати нормалізуються, щоб компенсувати відсутність рядка, який намагаємось змінити (масштабуємо решту коефіцієнтів так, щоб вони підсумовувались до одиниці):

```
1    private static Map<String, Double> normalize(Map<String, Double> probabilities) {
2        double coef = 1.0 / probabilities.values().stream().reduce(0.0, Double::sum);
3        Map<String, Double> normProfile = new HashMap<>(probabilities.size());
4        for (Map.Entry<String, Double> entry: probabilities.entrySet())
5            normProfile.put(entry.getKey(), entry.getValue() * coef);
6        return normProfile;
7    }
```

З отриманих k-мерів повернемо у якості результату випадковий, але тим варіантам, які мають більший шанс бути ближчими до правильного даватимемо більший шанс. Для цього спочатку оберемо випадкове число від нуля до одного, а потім обчислюватимемо кумулятивну суму імовірностей ново отриманих мотивів:

```
1    public static String weightedDie(Map<String, Double> prob) {
2        double selectedProb = random.nextDouble();
3        double accum = 0;
4        for (String kmer: prob.keySet()) {
5            accum += prob.get(kmer);
6            if (selectedProb <= accum) return kmer;
7        }
8        return prob.keySet().toArray(String[]::new)[prob.size() - 1];
9    }
```

Таким чином відрізок числової прямої від нуля до одиниці можна уявно поділити на послідовність шматочків, кожен з яких відповідає якомусь k-меру, при чому кандидати з більшою імовірністю мають більші шматочки, а отже і вищий шанс бути обраними.

Розділ 2. Опис алгоритмів зчитування

Перш ніж опрацьовувати з геномом, його треба прочитати: «Традиційний метод секвенування геномів описується наступним чином. Дослідники беруть невеликий зразок тканини або крові, що містить мільйони клітин з ідентичною ДНК, використовують біохімічні методи, щоб розбити ДНК на фрагменти, а потім секвенувати ці фрагменти для отримання зчитування (див. малюнок нижче).

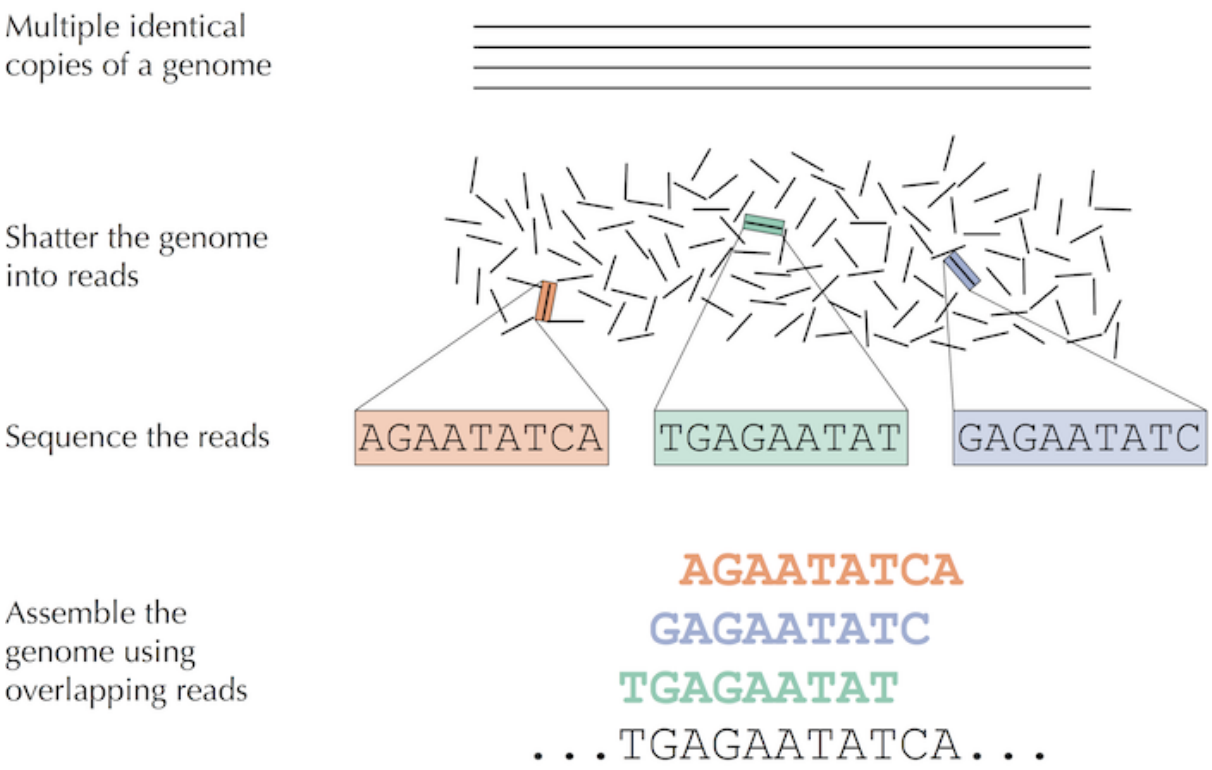

Рисунок 2 - секвенування [4]

Складність полягає в тому, що дослідники не знають звідки в геномі взялися ці зчитування, і тому вони повинні використовувати зчитування, що перекриваються, щоб реконструювати геном» [6].

З обчислювальної точки зору, цю задачу можна було б представити як таку: нехай на вхід подається колекція k-мерів, треба впорядкувати їх так, щоб вони утворювали початковий геном.

Наприклад, з вхідних даних {AAC, ACT, AAG, AGA, GAA, TAA} отримати TAAGAACT. Найпростіший алгоритм перебору, очевидно, буде вкрай неефективний, адже часто зустрічатимуться такі випадки як:

TAA + AAC + ACT + Ø

Ця гілка відтворення виявилась хибною, адже використано не всі тримери, треба пробувати розташувати їх інакше.

2.1 Граф Де Брейна (deBruign)

Перейдімо до ненаївного алгоритму. Перш за все, потрібно дати раду вхідній колекції k-мерів, організувати її у зручну для подальшої роботи структуру. Зручною формою їх представлення є граф, ребра якого є k-мерами, а вершини – префіксами та суфіксами відповідно. Далі вершини з однаковими значеннями можна «склеїти», тим самим зменшивши кількість вершин у графі, лишаючи при цьому усі ребра. Побудувати такий граф Де Брейна можна використовуючи такий алгоритм:

```
01  public static Map<String, List<String>> deBruign(List<String> patterns) {
02    Map<String, List<String>> prefMap = new HashMap<>();
03    for (String pattern : patterns) {
04       String prefix = getPrefix(pattern);
05       prefMap.putIfAbsent(prefix, new ArrayList<>());
06       prefMap.get(prefix).add(pattern);
07    }
08    Map<String, List<String>> adjList = new HashMap<>();
09    for (Map.Entry<String, List<String>> prefPat : prefMap.entrySet()) {
10       adjList.put(prefPat.getKey(), new ArrayList<>(prefPat.getValue().size()));
11       for (String pattern : prefPat.getValue()) {
12          adjList.get(prefPat.getKey()).add(getSuffix(pattern));
13       }
14    }
15    return adjList;
16 }
```

Просто поєднуючи префікс-вершини з суфікс-вершинами отримуємо шуканий граф.

2.2 Реконструкція геному (patternComposition)

Маючи можливість легко будувати граф Де Брейна можемо застосувати такий алгоритм реконструкції геному:

```
01  public static String patternComposition(List<String> patterns) {
02    Map<String, List<String>> dbGraph = deBruign(patterns);
03    Pair<Map<Integer, String>, Map<Integer, Deque<Integer>>> transformed =
transformGraph(dbGraph);
04    Map<Integer, String> dec = transformed.getFirst();
05    Map<Integer, Deque<Integer>> intGraph = transformed.getSecond();
06    List<Integer> path = eulerianPath(intGraph);
07    List<String> patternPath = path.stream()
08          .map(dec::get)
09          .collect(Collectors.toList());
10    StringBuilder resDna = new StringBuilder();
11    patternPath.forEach(p -> resDna.append(p.charAt(0)));
12    resDna.append(patternPath.get(patternPath.size() - 1).substring(1));
13    return resDna.toString();
14 }
```

Побудований граф Де Брейна кодується у числове представлення для легшої подальшої роботи, далі у ньому шукається Ейлеровий шлях, тобто послідовність усіх з’єднаних ребер графу, а в нашому випадку –послідовність усіх переданих k-мерів, які вибудовують шуканий ланцюг. Після декодування Ейлерового шляху з кожного ребра береться по одному нуклеотиду, з останнього береться хвіст, отриманий рядок і є представленням шуканого геному.

Код трансформації графу та пошуку Ейлерового шляху не мають великої цінності у контексті даної роботи, тому їх винесено у додаток Б.

2.3 Пошук контигів (findContigs)

Оскільки зчитати усю ДНК одним махом, строго кажучи, неможливо, науковці використовують контиги. “A contig - from the word "contiguous" - is a series of overlapping DNA sequences used to make a physical map that reconstructs the original DNA sequence of a chromosome or a region of a chromosome” [6]. Очевидно, що для пошуку таких послідовностей алгоритм потрібен, розгляньмо його варіант, який знаходить контиги за переданими k-мерами.

```
01    public static List<String> findContigs(String[] dna) {
02        Map<String, List<String>> graph = deBruign(dna);
03        Pair<Map<Integer, String>, Map<Integer, Deque<Integer>>> transformed =
transformGraph(graph);
04        Map<Integer, String> decoder = transformed.getFirst();
05        Map<Integer, Deque<Integer>> numGraph = transformed.getSecond();
06        Set<Integer> seeds = branchingNodes(numGraph);
07        List<List<Integer>> numContigs = new ArrayList<>();
08        for (Integer seed : seeds) {
09            Deque<Integer> seedAdj = numGraph.get(seed);
10            while (!seedAdj.isEmpty()) {
11                List<Integer> contig = new ArrayList<>(Arrays.asList(seed));
12                Integer v = seedAdj.pop();
13                while (!seeds.contains(v) && numGraph.containsKey(v) &&
numGraph.get(v).size() > 0) {
14                    contig.add(v);
15                    v = numGraph.get(v).pop();
16                }
17                contig.add(v);
18                numContigs.add(contig);
19            }
20        }
21        List<String> contigs = new ArrayList<>();
22        for (List<Integer> numContig : numContigs) {
23            List<String> splitContig =
numContig.stream().map(decoder::get).collect(Collectors.toList());
24            List<String> contigBatch = new
ArrayList<>(Arrays.asList(getPrefix(splitContig.get(0))));
25            contigBatch.addAll(splitContig.stream().map(x ->
String.valueOf(x.charAt(x.length() - 1))).collect(Collectors.toList()));
26            contigs.add(String.join("", contigBatch));
27        }
28        return contigs;
29    }
```

Для початку, звернімо увагу на приклад графу Де Брейна, відправної точки пояснення запропонованого алгоритму.

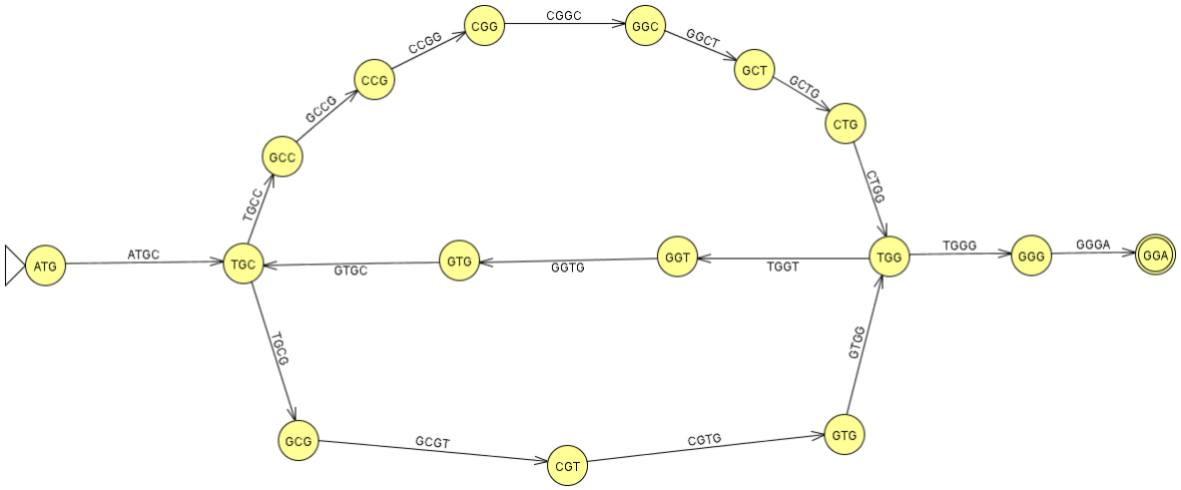

Рисунок 3 - граф Де Брейна

Як видно з ілюстрації вище, існує кілька варіантів пройтися по графу, а отже і зібрати початкову послідовність ДНК. Однак, його можна розбити на суцільні частини:

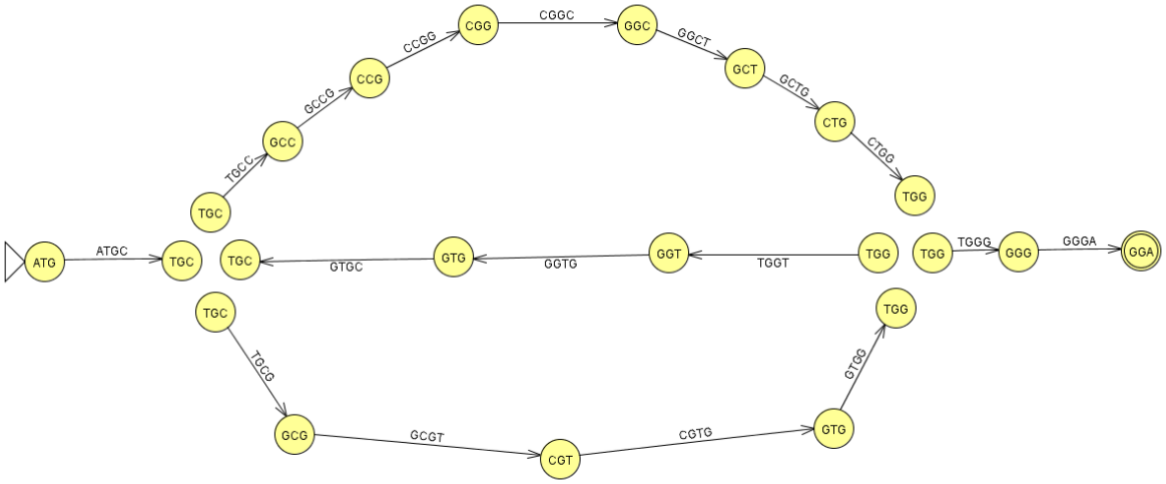

Рисунок 4 - роз'єднаний граф Де Брейна

Легко бачити, що виділені послідовності утворюють нерозгалужені шляхи, які притаманні довільному обходу графу, - контиги. Відповідно, задача пошуку контигів у колекції k-мерів зводиться до пошуку нерозгалужених шляхів побудованого з них графу Де Брейна.

Отож у першому (1) рядку імплементованого алгоритму будується граф Де Брейна. Далі він трансформується у числове представлення для легшого опрацювання. У шостому (6) рядку викликається метод branchingNodes:

```
01    private static Set<Integer> branchingNodes(Map<Integer, Deque<Integer>> numGraph) {
02        Map<Integer, List<Integer>> adjIn = new HashMap<>();
03        for (Map.Entry<Integer, Deque<Integer>> adjEntry : numGraph.entrySet())
04            for (Integer adjV : adjEntry.getValue()) {
```

```
05                if (!adjIn.containsKey(adjV))
06                    adjIn.put(adjV, new ArrayList<>());
07                adjIn.get(adjV).add(adjEntry.getKey());
08            }
09        Set<Integer> res = new HashSet<>();
10        for (Integer v : numGraph.keySet())
11            if (numGraph.get(v).size() > 1) res.add(v);
12            else if (numGraph.get(v).size() == 1 && !adjIn.containsKey(v) ||
adjIn.get(v).size() > 1) res.add(v);
13        return res;
14    }
```

Завдання цього методу – знайти у переданому графі можливі початки нерозгалужених шляхів. Іншими словами, знайти ті вершини, які не можуть бути внутрішньою частиною такого шляху, тобто не мають вхідний і вихідний степені рівні одному. Вершини, що порушують цю умову можуть бути на початку і кінці шляху, але не в середині.

Після знаходження можливих початків нерозгалужених шляхів та ініціалізації списку шуканих контигів на восьмому рядку методу findContigs починається їх перевірка. На дев’ятому (9) рядку виймаються усі суміжні до даної вершини, на прикладі рисунка 4 даною вершиною може бути TGC. Оскільки нас цікавить саме шлях, початок, що не має продовження не має цінності, з десятого (10) рядка використовуються лише вершини, що мають неопрацьовані вихідні ребра. Змінні seed, яка позначає початок контигу ставиться у початок списку, що позначає шукану послідовність. Повертаючись до прикладу з вершиною TGC бачимо, що для двох контигів, щоб пройтися обома шляхами на дванадцятому рядку дістаємо наступну суміжну вершину з набору неопрацьованих. Для того, щоб вершина була проміжною у нерозгалуженому шляху потрібно, щоб виконувались такі умови:

- Проміжна вершина не можу бути початком іншого контигу, очевидно.
- Вона повинна мати непорожній список суміжних вершин. Тобто в неї повинна бути одна суміжна вершина, інакше вона була б потенційним початком контигу.

Після їх перевірки в тринадцятому (13) рядку у тілі циклу контиг нарощує довжину. Після знаходження вершини, що не задовольняє умови проміжної вона додається у хвіст контигу. У вісімнадцятому (18) рядку повністю сформований нерозгалужений шлях додається до колекції усіх знайдених контигів. Далі результати опрацювання чисельного графу декодуються назад у символьні і k-мери, що утворюють контиги конкатинуються та додаються у результуючу колекцію.

2.4 Реконструкція пептиду з використанням пар зчитувань (pairedPatternReconstruction)

Робота зі зчитуваннями – це, звичайно, добре, але є дещо краще – робота з парами зчитувань. Річ у тім, що з довшими k-мерами граф виходить менш заплутаним. Що більша довжина вхідних даних, тим легше відтворити оригінальний ланцюг. На жаль, генерація довгих зчитувань складна з практичної точки зору, тому біологи вигадали виконувати пари зчитувань довжиною k, із пропусками довжиною d. Взявши цю ідею за основу, адаптуємо попередній алгоритм до роботи із парами зчитувань:

```
01    public static String pairedPatternReconstruction(int d, List<Pair<String, String>>
vertices) {
02        int p = vertices.get(0).getFirst().length() - 1;
03        Map<Pair<String, String>, List<Pair<String, String>>> originalGraph =
pairedDeBruign(vertices);
04        Pair<Map<Integer, Pair<String, String>>, Map<Integer, Deque<Integer>>>
transformation = transformGraph2(originalGraph);
05        Map<Integer, Pair<String, String>> remapper = transformation.getFirst();
06        Map<Integer, Deque<Integer>> mappedGraph = transformation.getSecond();
07        List<List<Integer>> paths = allEulerianPaths(mappedGraph, new
ArrayDeque<>(List.of(peekStart(mappedGraph))))
08                .stream().distinct().collect(Collectors.toList());
09        paths.forEach(Collections::reverse);
10        List<List<Pair<String, String>>> decodedPaths = new ArrayList<>(paths.size());
11        for (var encPath : paths)
12
decodedPaths.add(encPath.stream().map(remapper::get).collect(Collectors.toList()));
13        List<Pair<String, String>> validPath = null;
14        for (List<Pair<String, String>> path : decodedPaths) {
15            boolean pathFailed = false;
16            for (int i = 0; i < path.size(); i++) {
17                String fstRead = path.get(i).getFirst();
18                String sndRead = path.get(i).getSecond();
19                if (i == 0) {
20                    for (int j = 0; j < p - 1; j++) {
21                        pathFailed = !checkColumn(fstRead, sndRead, i, j, p, d, path);
22                        if (pathFailed) break;
23                    }
24                    if (pathFailed) break;
25                }
26                pathFailed = !checkColumn(fstRead, sndRead, i, p - 1, p, d, path);
27                if (pathFailed) break;
28            }
29            if (pathFailed) continue;
30            validPath = path;
31        }
32        final StringBuilder genomePrefix = new StringBuilder();
33        if (validPath == null) return null;
34        validPath.forEach(node -> genomePrefix.append(node.getFirst().charAt(0)));
35        genomePrefix.append(validPath.get(validPath.size() - 1).getFirst().substring(1));
36        for (int i = d + 1; i > 0; i--)
37            genomePrefix.append(validPath.get(validPath.size() - 1 -
i).getSecond().charAt(0));
38        genomePrefix.append(validPath.get(validPath.size() - 1).getSecond());
39        return genomePrefix.toString();
40    }
```

З основних відмінностей від попереднього варіанту відмітимо, що на сьомому (07) рядку шукаються усі Ейлерові шляхи, при цьому для вибору початку шляхи використовується інший алгоритм описаний в методі peekStart. У чотирнадцятому (14) рядку починається перевірка отриманих шляхів, на цьому

етапі неправильні відкидаються, решта починаючи з тридцять другого (32) рядка використовується для формування відповіді.

Аналогічно до попереднього разу, код методів transformGraph2 та allEulerianPaths винесено у додаток В. Код pairedDeBruign подібний до описаного вище варіанту, тому він теж занесений у додаток.

Розгляньмо код алгоритму вибору вершини для пошуку шляху Ейлера.

```
01    private static Integer peekStart(Map<Integer, Deque<Integer>> mappedGraph) {
02        Optional<Integer> maybeMax = Stream.concat(mappedGraph.keySet().stream(),
mappedGraph.values().stream().flatMap(Collection::stream)).max(Integer::compareTo);
03        if (maybeMax.isEmpty()) throw new NoSuchElementException("Could not find maximum
vertex label.");
04        int[] dDegrees = new int[maybeMax.get() + 1];
05        for (Integer v : mappedGraph.keySet()) {
06            Deque<Integer> adj = mappedGraph.get(v);
07            dDegrees[v] += adj.size();
08            for (Integer vx : adj)
09                dDegrees[vx]--;
10        }
11        Integer starter = null;
12        for (int i = 0; i < dDegrees.length; i++)
13            if (dDegrees[i] == 1) starter = i;
14        if (starter == null) throw new NoSuchElementException("Could not find a peek.");
15        return starter;
16    }
```

Спочатку обирається найбільше числове значення у закодованому графі, яке використовується для створення масиву підрахунку степенів вершин (різниця між кількістю вхідних та вихідних ребер). Ребро яке має на одне вихідне ребро більше ніж кількість вхідних обирається початком для пошуку усіх шляхів Ейлера.
Розгляньмо алгоритм перевірки отриманих шляхів. Для цього корисно спершу побачити результат роботи методу allEulerianPaths та його тлумачення в контексті реконструкції початкової послідовності нуклеотидів. Нижче наведено приклад правильного шляху з k=4 і d=2.

1	G	T	G	-	G	T	G							
2		T	G	G	-	T	G	A						
3			G	G	T	-	G	A	G					
4				G	T	C	-	A	G	A				
5					T	C	G	-	G	A	T			
6						C	G	T	-	A	T	G		
7							G	T	G	-	T	G	T	
8								T	G	A	-	G	T	T
9									G	A	G	-	T	G	G
10										A	G	A	-	T	G	A

Легко бачити, що символи рухаються у лівий бік. Це показує, що кожна наступна пара містить один правіший символ, як і решту попередніх, крім найлівішого, який не вмістився у розмір пари. Більш формально, символ, що стоїть в i-тому рядку, на j-тій позиції повинен бути присутній в (i+1)-тому рядку на (j-1)-ій позиції (якщо він не виходить за межі зчитування). Таким чином можна стовпці однакових символів, наявність яких є свідченням прийнятності обчисленого шляху.

Особливої уваги заслуговує випадок переходу символу перед розривом. Він зникає з поля зору на d+1 крок, адже алгоритм побудови графу Де Брейна «обрізає» з обох зчитувань один нуклеотид і рухається на d позицій праворуч.

Тепер розгляньмо код, що реалізує алгоритм перевірки таких стовпців.

```
01    private static boolean checkColumn(String fstRead, String sndRead, int i, int j, int k, int d,
List<Pair<String, String>> path) {
02        for (int p = 1; p < k; p++) {
03            if (j >= p && i + p < path.size() && fstRead.charAt(j) != path.get(i +
p).getFirst().charAt(j - p))
04                return false;
05            int absPos = k + d + j - p;
06            Boolean pair2Proj = null;
07            if (absPos < k)
08                pair2Proj = true;
09            else if (absPos >= k + d)
10                pair2Proj = false;
11            if (pair2Proj != null && i + p < path.size()
12                    && sndRead.charAt(j) != (pair2Proj ? path.get(i + p).getFirst().charAt(absPos)
13                    : path.get(i + p).getSecond().charAt(absPos - k - d)))
14                return false;
15        }
16        return true;
17    }
```

Оскільки перший символ першого зчитування кожної пари «випаде» у наступній парі починаємо перевірку з індексу «1». На 3 рядку виконується перевірка вказаного нуклеотиду в першому зчитуванні, якщо така можлива:

1. індекс j нуклеотиду, що перевіряється, більший або дорівнює p, тобто ітерація перевірки ще не обігнала цільову позицію
2. під даним рядком є ще p наступних
3. у першому зчитуванні символ з індексом j відрізняється від такого в (i+p)-тому рядку, (j-p)-тому стовпчику

Якщо вимоги 1 та 2 виконуються, це означає, що перевірка 3 може бути виконана, якщо вона валиться, то далі нема сенсу нічого перевіряти.

У 5 рядку обчислюється абсолютний індекс заданого нуклеотиду в повній результуючій послідовності. Він може припасти (спроектуватися) на:

- Перше зчитування

- Друге зчитування
- Розрив довжиною d

Якщо він не припадає на розрив та існують наступні p рядків, слід перевірити, що даний нуклеотид рівний такому на відповідній позиції зчитування, на яке падає проекція.

Якщо перевірка ніде не «звалилась», вона вважається пройденою.

Розділ 3. Робота з амінокислотами

Як відомо, протеїни складаються з амінокислот, які у свою чергу створюються на базі РНК. Для цього РНК розбивається у 3-мери, які називаються кодонами, далі кожна така частина транслюються у амінокислоти відповідно до генетичного коду, виняток становлять лише стоп-кодони, які припиняють створення протеїну.

3.1 Трансляція РНК в протеїн (translateProtein)

Нижче наведено тривіальний алгоритм, який вирішує задачу трансляції РНК в протеїн. Звернімо увагу на константу GENETIC_CODE, у реалізації алгоритму використовується набір із 64 пар «кодон – амінокислота». Застосовується колекція найбільш часто зустріваних відповідностей [7], але вона не є вичерпною.

```
1   public static String translateProtein(String rnaPattern) {
2     StringBuilder amonoAcid = new StringBuilder();
3     for (int I = 0; I < rnaPattern.length(); I += 3) {
4        char acid = Constants.GENETIC_CODE.get(rnaPattern.substring(I, I + 3));
5        if (acid == '*') return amonoAcid.toString();
6        amonoAcid.append(acid);
7     }
8     return amonoAcid.toString();
9   }
```

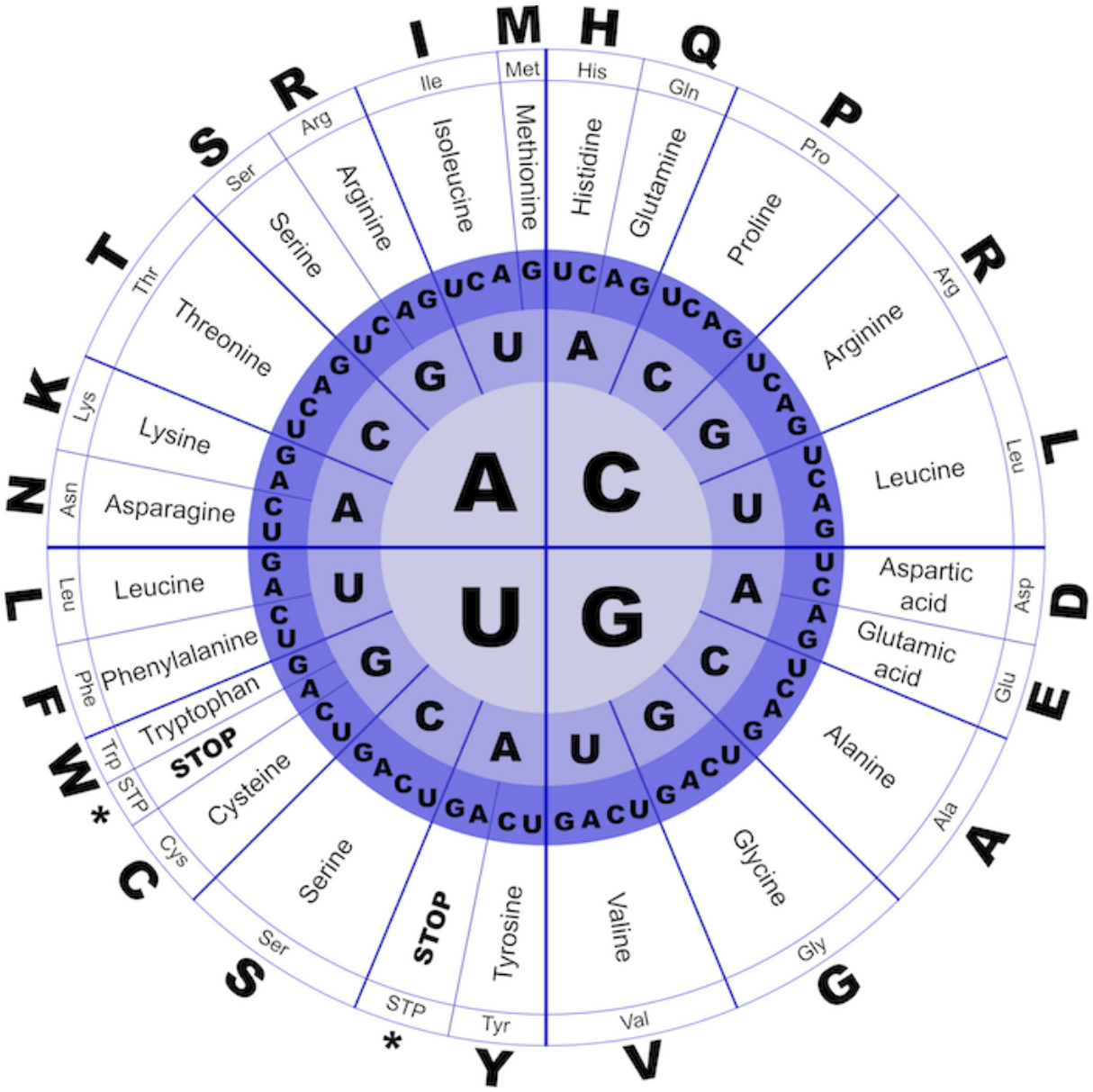

Рисунок 5 - кодування амінокислот [8]

3.2 Пошук кодувань пептиду (peptideEncoding)

З огляду на таблицю стає очевидним, що декілька послідовностей нуклеотидів можуть кодувати ту саму амінокислоту, а отже і пептид чи навіть протеїн. Через це виникає природнє питання: «Маючи певний уривок ДНК, як дізнатися які підпослідовності нуклеотидів кодують певний вибраний пептид». Враховуючи те, що існує 3 способи розбити уривок ДНК на набір кодонів (адже їх довжина = 3) та беручи до уваги той факт, що спіраль ДНК складається з двох протилежно напрямлений частин складаємо алгоритм:

```
01  private static final Map<Character, Character> RC_MAP = new HashMap<>() {{
02    put('A', 'U');
03    put('T', 'A');
04    put('C', 'G');
05    put('G', 'C');
```

```
06    }};
07
08 public static List<String> peptideEncoding(String dna, String peptide) {
09    String rna = dna.replaceAll("T", "U");
10    List<Pair<Integer, Integer>> dnaResPos = findPeptidePos(rna, peptide);
11    StringBuilder sb = new StringBuilder(dna).reverse();
12    for (int I = 0; I < sb.length(); i++) sb.setCharAt(I, RC_MAP.get(sb.charAt(i)));
13    String reverseComplementRna = sb.toString();
14    List<Pair<Integer, Integer>> rnaResPos = findPeptidePos(reverseComplementRna,
peptide);
15    List<String> res = new ArrayList<>();
16    dnaResPos.forEach(posPair -> res.add(dna.substring(posPair.getFirst(),
posPair.getSecond())));
17    rnaResPos.forEach(posPair -> res.add(dna.substring(dna.length() -
posPair.getSecond(),
18          dna.length() - posPair.getFirst()).replace("U", "T")));
19    return res;
20 }
21
22 private static List<Pair<Integer, Integer>> findPeptidePos(String rna, String peptide) {
23    List<Pair<Integer, Integer>> res = new ArrayList<>();
24    int matchIndex = 0;
25    for (int I = 0; I < rna.length() - 3 * peptide.length(); i++) {
26       if (Constants.GENETIC_CODE.get(rna.substring(I, I + 3)) ==
peptide.charAt(matchIndex)) {
27          matchIndex = 1;
28          int j = I + 3;
29          while (true) {
30             if (matchIndex == peptide.length() - 1) {
31                res.add(Pair.of(j - (peptide.length() - 1) * 3, j + 3));
32                break;
33             }
34             if (Constants.GENETIC_CODE.get(rna.substring(j, j + 3)) ==
peptide.charAt(matchIndex)) {
35                matchIndex++;
36                j += 3;
37             } else break;
38          }
39          matchIndex = 0;
40       }
41    }
42    return res;
43 }
```

Відповідно до алгоритму спочатку вхідна ДНК транскрибується в РНК, будується її доповнення, далі в методі findPeptidePos тривіальним чином позиції знаходяться усіх підпослідовностей, які транслюються у шукані пептиди (масив амінокислот).

Розділ 4. Опис алгоритмів для відтворення пептидів, що відповідають заданому спектру

Загально відомий факт, що генетична інформація зберігається у вигляді ДНК. Центральна догма молекулярної біології стверджує: «DNA makes RNA, and RNA makes protein», однак не для всіх протеїнів вона справджується. Нерибосомні пептиди, наприклад, не залежать від мРНА і створюється на нерибосомних пептидсинтезах (NRPS) [9]. Для виведення таких пептидів використовується спектрометрія мас. Mass spectrometry is an analytical tool useful for measuring the mass-to-charge ratio (m/z) of one or more molecules present in a sample. These measurements can often be used to calculate the exact molecular weight of the sample components as well [10]. Надалі для спрощення припускатимемо, що результатом роботи спектрометра мас є масив цілочисельних мас частин пептиду: від першої амінокислоти до сумарної маси усіх амінокислот у пептиді, відсортованих у природному порядку.

Теоретичним спектром вважатимемо ідеальний масив мас, який достовірно відображає усі амінокислоти та їх послідовності у пептиді.

Експериментальним спектром вважатимемо результат роботи спектрометра мас, який містить хибні, відсутні у теоретичному спектрі, маси і не має частини мас наявних в ідеальному спектрі.

4.1 Опис алгоритму пошуку пептидів за заданим теоретичним спектром (perfectSpectrumPeptides)

Мета наведеного нижче алгоритму – за переданим теоретичним спектром відтворити усі пептиди, що можуть його утворити.

```
01  public static Collection<int[]> perfectSpectrumPeptides(final int[] spectrum) {
02    Map<Integer, Integer> spectrumSet = new HashMap<>();
03    for (int I = 1; I < spectrum.length; i++) {
04       spectrumSet.compute(spectrum[i], (acid, num) -> num == null ? 1 : num + 1);
05    }
06    Set<Integer> basicAcids = spectrumSet.keySet();
07
08    basicAcids.retainAll(Constants.AMINO_ACID_MASS.values());
09
10    BiMap<Integer, Integer> basicOrder = HashBiMap.create(basicAcids.size());
11    for (Integer acidMass : basicAcids)
12       basicOrder.put(basicOrder.size(), acidMass);
13
14    // Queue is a collection of consistent peptide abstractions
15    // fst pair = a peptide denoted by an array representing masses of its amino acids
16    // snd pair = num of occurrences of each basic consistent amino acid
17    Queue<Pair<int[], int[]>> consistentPeptides = new ArrayDeque<>();
18    List<int[]> peptides = new ArrayList<>();
19    basicAcids.forEach(acid -> {
20       int[] freqCache = new int[basicAcids.size()];
21       freqCache[basicOrder.inverse().get(acid)] = 1;
22       consistentPeptides.add(Pair.of(new int[]{acid}, freqCache));
23    });
```

```
24    while (!consistentPeptides.isEmpty()) {
25       Pair<int[], int[]> currPeptide = consistentPeptides.poll();
26       for (Integer acidMass : basicAcids) {
27          int[] acidsCache = currPeptide.getSecond();
28          int alreadyPresent = acidsCache[basicOrder.inverse().get(acidMass)];
29          if (spectrumSet.get(acidMass) == alreadyPresent) continue;
30
31          int[] nextPeptide = Arrays.copyOf(currPeptide.getFirst(),
currPeptide.getFirst().length + 1);
32          nextPeptide[nextPeptide.length - 1] = acidMass;
33
34          if (spectrumSet.get(acidMass) == alreadyPresent + 1) {
35             int peptideMass = 0;
36             for (int I = 0; I < acidsCache.length; i++) {
37                peptideMass += acidsCache[i] * basicOrder.get(i);
38             }
39             peptideMass += acidMass;
40             if (peptideMass == spectrum[spectrum.length - 1]
41                   && Arrays.equals(spectrum, theoreticalPeptideSpectrum(nextPeptide,
true))) {
42                peptides.add(nextPeptide);
43                continue;
44             }
45          }
46          int[] cacheUpd = Arrays.copyOf(acidsCache, acidsCache.length);
47          cacheUpd[basicOrder.inverse().get(acidMass)]++;
48          consistentPeptides.add(Pair.of(nextPeptide, cacheUpd));
49       }
50    }
51    return peptides;
52 }
```

У другому (2) рядку визначається асоціативний масив spectrumSet, ключем якого є маса підпептиду (частина послідовності амінокислот, що містяться у пептиді), значенням – кількість входжень цієї маси у спектр. Кількість амінокислот, з яких складається пептид імовірно менша за їх загальну кількість, тому у шостому (6) рядку із переданого спектру прибираються усі маси, які не відповідають жодній єдиній амінокислоті, тобто такі, які точно складаються з мас кількох амінокислот. В результаті, множина basicAcids містить маси амінокислот, з яких і складається пептид, що відповідає спектру. У десятому рядку визначається двосторонній асоціативний масив, ключем якого є порядковий номер, а значенням – маса і-тої амінокислоти, яка входить до складу пептиду. На перший погляд може здатися надлишковим використання такої структури даних, адже доступ за послідовним цілочисельним індексом, може забезпечити і звичайний масив, але у масиві не можна константний час за значенням отримати індекс. У сімнадцятому (17) рядку визначається черга консистентних підпептидів, consistentPeptides. Пептид вважається консистентним із заданим спектром, якщо спектр створений з мас його амінокислот є підмножиною заданого спектру. У рядках дев’ятьнадцять – двадцять три (19-23) встановлюється тривіальний консистентний пептид та кеш, що йому відповідає. У рядках двадцять чотири – п’ятдесят (24-50) виконується алгоритм, який можна коротко описати так:

- Дістати з черги наступний консистентний пептид.

- Для заданого пептиду пробігтися по усіх доцільних амінокислотах.
- Подивитися у кеші, скільки таких амінокислот вже наявно у пептиді і якщо додавання і-тої амінокислоти до пептиду зробить його неконсистентним, перейти до наступної амінокислоти.
- Створити новий пептид на базі даного, додавши до нього і-ту амінокислоту.
- Якщо сумарна маса новоутвореного пептиду дорівнює масі переданого, занести такий пептид до колекції результатів, інакше оновити кеш та додати його у чергу консистентних пептидів.
- Коли консистентні пептиди у черзі закінчились, повернути колекцію результуючих пептидів.

Частиною наведеної вище реалізації є виклик методу theoreticalPeptideSpectrum, який будує теоретичний спектр переданого пептиду залежно від того, чи є він циклічним. Тут спектр представлений мультимножиною, яка є частковим випадком асоціативного масиву і є аналогічною до spectrumSet. Очевидно, що значення спектру генеруються із переданого пептиду ітеративно, а прапор isCircular визначає, чи продовжиться нарощування маси з лівого боку масиву, що абстрагує пептид, коли індекс дійде до правого його краю.

```
01  public static int[] theoreticalPeptideSpectrum(final int[] peptide, boolean isCircular)
{
02    SortedMultiset<Integer> spectrum = TreeMultiset.create();
03    spectrum.add(0);
04    int accum = 0;
05    for (int I = 0; I < peptide.length; i++) {
06       accum = 0;
07       for (int j = I; j < peptide.length + (isCircular ? I - 1 : 0); j++) {
08          accum += peptide[j - (isCircular && j >= peptide.length ? peptide.length : 0)];
09          spectrum.add(accum);
10       }
11    }
12    if (peptide.length > 1)
13       spectrum.add(accum + peptide[peptide.length - 2]);
14    return Ints.toArray(spectrum);
15 }
```

Реалізація perfectSpectrumPeptides наведена вище використовує кешування (ці чистини коду виділено) задля пришвидшення роботи програми, однак очевидно, що зберігання результатів попередніх обчислень вимагає додаткової пам’яті. Якщо поглянути на дане рішення скептично, то у гіршому випадку, якщо кешування неефективне, швидкодія може не змінитися, а може навіть і погіршитися (якщо кеш фактично не допомагає), адже операції читання/запису та створення кеш-об’єктів також вимагають часу. Аби пересвідчитися у доцільності застосованого кешування було проведено тест. Програмний код

спрощеного алгоритму та тесту (benchmark) наведено у додатку Д, розділах 1 та 2 відповідно. Як можна бачити з даних у розділі 3, для наведених там-таки вхідних даних алгоритм з кешуванням виявився на порядок швидшим, отже використання кешу є виправданим.

4.2 Опис алгоритму пошуку пептидів за заданим експериментальним спектром (leaderboardCyclopeptideSequencing)

Світ не ідеальний, тому не досконала і спектрометрія мас. Оскільки експериментальний спектр містить помилки, потрібен і алгоритм здатний правильно працювати і з неідеальними вхідними даними. Очевидно, попередніх підхід фільтрації неконсистентних пептидів тут не спрацює, тому використовується механізм зважування: зберігаються перші n класів пептидів, чиї теоретичні найбільше подібні до переданого експериментального. Під класом пептидів вважатимемо множину пептидів, що мають однаковий бал подібності. Бали подібності поступово нараховуються у процесі виконання алгоритму. Масив acidMasses представляє набір мас, що відповідають амінокислотам, з яких може складатися шуканий пептид.

```
01  public static List<int[]> leaderboardCyclopeptideSequencing(final int[] spectrum, final
int[] acidMasses, final int n) {
02     final Multiset<Integer> spectrumIdx = HashMultiset.create();
03     for (int mass : spectrum) spectrumIdx.add(mass);
04
05     ArrayDeque<PeptideCandidate> candidates = new ArrayDeque<>();
06     candidates.add(new PeptideCandidate());
07     List<Leader> leaders = new ArrayList<>();
08     TreeMap<Integer, Integer> scoreLen = new TreeMap<>();
09     scoreLen.put(0, Integer.MAX_VALUE);
10
11     while (!candidates.isEmpty()) {
12        PeptideCandidate candidate = candidates.poll();
13        for (Integer acidMass : acidMasses) {
14           if (candidate.mass + acidMass > spectrum[spectrum.length - 1]) continue;
15           int scoreInc = 0;
16           int[] outgrowth = candidate.deltaSpectrum(acidMass);
17           for (int spMass : outgrowth) {
18              if (spectrumIdx.count(spMass) - candidate.localSpectrum.count(spMass) > 0)
scoreInc++;
19           }
20           int offeredScore = candidate.score + scoreInc;
21           if (offeredScore < scoreLen.firstKey() && scoreLen.size() == n ||
22                 scoreLen.containsKey(offeredScore) && scoreLen.get(offeredScore) <
candidate.acids.length + 1)
23              continue;
24           PeptideCandidate nextCandidate = candidate.ascend(acidMass, outgrowth,
scoreInc);
25
26           if (nextCandidate.mass == spectrum[spectrum.length - 1]) {
27              Multiset<Integer> cyclicPart = nextCandidate.cyclicSpectrumAddition();
28              int cyclicScore = nextCandidate.score;
29              for (Multiset.Entry<Integer> ptSum : cyclicPart.entrySet()) {
30                 int needed = Math.max(spectrumIdx.count(ptSum.getElement()) -
nextCandidate.localSpectrum.count(ptSum.getElement()), 0);
31                 cyclicScore += ptSum.getCount() - (Math.max(ptSum.getCount() - needed,
0));
```

```
32              }
33              if (leaders.isEmpty() || cyclicScore > leaders.get(0).score) {
34                 leaders.clear();
35                 leaders.add(new Leader(nextCandidate.acids, cyclicScore));
36              } else if (cyclicScore == leaders.get(0).score) {
37                 if (nextCandidate.acids.length > leaders.get(0).acids.length) continue;
38                 if (nextCandidate.acids.length < leaders.get(0).acids.length) {
39                    leaders.clear();
40                 }
41                 leaders.add(new Leader(nextCandidate.acids, cyclicScore));
42              }
43           } else if (nextCandidate.mass < spectrum[spectrum.length - 1]) {
44              if (!scoreLen.containsKey(offeredScore)) {
45                 scoreLen.put(offeredScore, nextCandidate.acids.length);
46                 if (scoreLen.size() > n) {
47                    final int outweighed = scoreLen.firstKey();
48                    candidates.removeIf(c -> c.score == outweighed);
49                    scoreLen.remove(outweighed);
50                 }
51              } else scoreLen.put(offeredScore, nextCandidate.acids.length);
52              candidates.add(nextCandidate);
53           }
54        }
55     }
56     return leaders.stream().map(x -> x.acids).collect(Collectors.toList());
57  }
```

Спочатку, подібно до попереднього алгоритму, з масиву спектру створюється відповідна мультимножина. Далі визначається черга candidates, яка містить щонайбільше n+1 класів пептидів із найвищими балами. У шостому (6) рядку черга наповнюється тривіальним пептидом, що стане базою для наступних кандидатів. Список leaders – колекція результатів із найдорожчими пептидуми. Далі визначається асоціативний масив порогів, що ставить у відповідність бал подібності та мінімальну довжину пептиду, який зміг отримати такий бал. Така структура потрібна у наступному випадку: нехай є підпептид PAS, який має бал 3 і вже прийнятий в чергу кандидатів. Пептиди PASh, PAShq тощо побудовані на його основі також мають бал 3, але їх збільшена довжина не робить вклад у бал, який вони отримують. Таких омнливий кандидатів може утворитися грандіозно багато, адже їх кількість, не обмежена видаленням класів із малим балом, має швидкість зростання експоненційну відносно кількості амінокислот в acidMasses. scoreLen базується на червоно-чорних деревах, такий вибір зумовлений властивістю даної структури зберігати ключі у відсортованому порядку, що використана в алгоритмі. Наповнюється дана структура даних тривіальною парою, що відповідає ініціалізуючому пептиду довжини нуль. Можна стверджувати, що будь-який пептид кращий за тривіальний, але можлива ситуація, коли пептид із високим балом має помилку у першій амінокислоті. Він має пройти далі. Хоч цей випадок взято до уваги нижче, інтуїтивно зрозуміло, що мінімальну довжину для пептиду з балом нуль гарно ініціалізувати найбільшим цілим числом, зокрема і для того, щоб наочно підкреслити винятковість даного значення.

На одинадцятому (11) рядку починається головна частина алгоритму.

Спочатку з черги candidates дістається наступний кандидат для розгляду. Далі цикл проходиться по кожній допустимій амінокислоті, якщо додавання і-тої амінокислоти до даного кандидата призведе до появи пептиду, чия маса більша, за останню масу спектру, то такий новоутворений пептид далі не розглядається, адже він майже точно хибний. Звісно існує ймовірність, що останнє значення експериментального спектру не відповідає такому в теоретичному, але ж і експериментів можна провести кілька. Інтуїтивно зрозуміло, що в більшості випадків довільне значення в експериментальному спектрі буде правильним, отже і останнє теж.

У п’ятнадцятому (15) рядку визначається змінна scoreInc, яка зберігає кількість балів, які набуде даний кандидат у разі приєднання до нього даної амінокислоти. Масив outgrowth містить усі нові підпептиди, які стануть частиною теоретичного спектру новоутвореного пептиду. У рядках сімнадцять (17) – дев’ятнадцять (19) для кожного такого підпептиду порівнюється кількість його входжень в експериментальному спектрі та батьківському пептиді-кандидаті. Якщо «вакантне місце» для цього пептиду є, то лічильник балів, збільшується. Важливо звернути увагу, що при порівнянні використовується саме спектр батька, який не оновлюється у разі «прийняття» підпептиду, адже цей самий спектр потрібен буде і для всіх наступних нащадків. Може виникнути питання, що буде, якщо у масиві outgrowth виявляться дублікати. Якщо, наприклад, в експериментальному спектрі буде міститися 3 підпептиду масою 329, у кандидата вже буде 2 таких, а в масиві outgrowth ще 3, тоді вищезгаданий цикл тричі подивиться, що є ще одне вільне місце і додасть по одному балу 3 рази. Не може такого статись, однак, через те, що значення в масиві outgrowth згенеровані рекурсивно – кожне неступне більше за попереднє і залежить від послідовності амінокислот батька.

У двадцятому (20) рядку обчислюється сумарний бал пропонованого пептиду, далі відбувається перевірка, яка блокує (не пропускає далі) новоутворення, якщо його бал нижчий, за мінімально допустимий, або якщо його довжина перевищує таку в іншого пептиду з таким самим балом. Однак ці обмеження не діють, поки кандидатів не набереться n класів, таким чином на початкових етапах алгоритм охоче набирає усіх, а потім починає реально фільтрувати. Якщо перевірку пройдено, на базі даного кандидата створюється новий.

У двадцять шостому (26) рядку перевіряється маса новоутворення. Якщо вона відповідає останньому значенню експериментального спектру, тобто сумі мас усіх амінокислот твірного пептиду, то такий пептид розглядається як потенційний лідер. Для нього будується циклічна частина спектру, яка не створюється для звичайних, проміжних кандидатів. Ця частина потім

використовується для підрахунку повного балу подібності. Якщо фінальний бал даного потенційного лідера більший за бал пептидів, які наразі вважаються лідерами, то всі ті лідери видаляються, а новий, кращий пептид стає першим елементом у колекції лідерів. Якщо ж лідерів і так не було, то природньо він просто додається. У випадку, якщо бал даного новоутворення рівний балу теперішніх лідерів, він просто долучається до їх списку. В іншому випадку, якщо бал менший, такий недолідер відкидається.

Звернімо особливу увагу на рядок тридцять (30). Версія, яка пройшла перший набір тестів не обгортала зазначену різницю кількості однакових елементів експериментального і теоретичного згенерованого спектрів функцією максимуму з нулем у якості другого аргументу. У першій версії цього рядка коду не було взято до уваги той факт, що коли спектр згенерованого кандидату має більшу кількість якогось елементу, ніж експериментальний спектр, значенням змінної needed буде негативним, хоч насправді у такому разі дана часткова сума взагалі не потрібна/не корисна для збільшення балу кандидату. Відомо, що ця помилка призводила до втрати правильних лідерів, а її вплив став помітним лише при тестуванні алгоритму «згорткового пошуку», описаного нижче.

Важливо звернути увагу на сорок третій (43) рядок: до candidates додаються лише кандидати, в яких маса не перевищує масу шуканого пептиду. Така оптимізація дозволяє уникнути розгляду кандидатів, чиї нащадки будуть невідворотно відкинуті на початку наступних ітерацій. Це дозволяє зекономити acidMasses.length операцій і в ході експериментів виявилось критичним для швидкодії програми.

Якщо маса нового кандидата не збігається з такою в шуканого пептиду, nextCandidate додається до черги candidates. Разом із тим, якщо новий кандидат дає початок наступному класу, тобто має вищий бал, ніж усі кандидати до цього, scoreLen оновлюється його значенням і якщо фаза інціалізації вже закінчилась, тобто n класів вже набрано, клас пептидів із найнижчим балом видаляється з черги. Якщо ж із таким балом клас вже існує, то оновлюється значення максимальної допустимої довжини для нових пептидів, яке гарантовано буде не більше, ніж попереднє.

У п’ятдесят шостому (56) рядку відбувається екстракція власне масивів амінокислот із колекції лідерів, це значення, а саме, набір пептидів, сума мас амінокислот яких дорівнює масі шуканого пептиду, і повертається.

Для зручності було створено клас незмінних об’єктів PeptideCandidate, який інкапсулює такі поля:

- acids – масив амінокислот, що творять даний підпептид
- mass – маса підпептиду, використовується як кешована змінна задля уникнення сумування усіх елементів масиву acids
- localSpectrum – теоретичний спектр побудований на базі даного пептиду, кешована змінна
- score – бал подібності спектру даного пептиду до експериментального
- lastDiagonal – краще пояснити буде через приклад. Розгляньмо уявний пептид ABCDE, побудуймо для нього спектр:

A	B	C	D	E
AB	BC	CD	DE	[EA]
ABC	BCD	CDE	[DEA]	[EAB]
ABCD	BCDE	[CDEA]	[DEAB]	[EABC]
ABCDE				

Таблиця 1 – візуалізація алгоритму побудови спектру

Легко бачити, що таблиця «наростає» діагоналями. Кожна нова літера додається починає свою колонку і навкоси додається до останніх слів у попередніх рядках. Ідеальним прикладом є літера D, однак в таблиці наявні і слова в квадратних дужках, вони відповідають циклічним пептидум. Алгоритм побудови тривіальний, тому зауважимо лише, що усі 5 літер зустрічаються лише 1 раз, адже, як відомо, додавання є комутативним, і n однакових передбачуваних сум цінної інформації у спектр не додадуть. lastDiagonal очевидно – масив сум утворений при додаванні останньої амінокислоти до підпептиду.

```
01    public static class PeptideCandidate {
02        public final int[] acids;
03        public final int mass;
04        public final Multiset<Integer> localSpectrum;
05        public final int score;
06        private final int[] lastDiagonal;
07
08        public PeptideCandidate(int[] acids, int mass, Multiset<Integer> localSpectrum,
int score, int[] lastDiagonal) {
09            this.acids = acids;
10            this.localSpectrum = localSpectrum;
11            this.mass = mass;
12            this.score = score;
13            this.lastDiagonal = lastDiagonal;
14        }
15
16        public PeptideCandidate() {
17            acids = new int[0];
18            mass = 0;
19            localSpectrum = HashMultiset.create();
20            score = 0;
21            lastDiagonal = new int[0];
22        }
```

```
23
24          public int[] deltaSpectrum(int newAcid) {
25              int[] delta = new int[lastDiagonal.length + 1];
26              for (int I = 0; I < lastDiagonal.length; i++) {
27                  delta[i] = lastDiagonal[i] + newAcid;
28              }
29              delta[lastDiagonal.length] = newAcid;
30              return delta;
31          }
32
33          public PeptideCandidate ascend(int newAcid, int[] recentSubPeptides, int
scoreRise) {
34              int[] nextAcids = Arrays.copyOf(acids, acids.length + 1);
35              nextAcids[acids.length] = newAcid;
36              Multiset<Integer> nextLocalSpectrum = HashMultiset.create(localSpectrum);
37              for (int subPeptide : recentSubPeptides) nextLocalSpectrum.add(subPeptide);
38              return new PeptideCandidate(nextAcids, mass + newAcid, nextLocalSpectrum,
score + scoreRise, recentSubPeptides);
39          }
40
41          public Multiset<Integer> cyclicSpectrumAddition() {
42              Multiset<Integer> cyclic = HashMultiset.create();
43              int columnFirst = 0;
44              for (int I = acids.length - 1; I > 1; i--) {
45                  int columnAccum = columnFirst += acids[i];
46                  for (int j = 0; j < I - 1; j++) {
47                      cyclic.add(columnAccum += acids[j]);
48                  }
49              }
50              return cyclic;
51          }
52
53      }
```

Крім полів даний клас має 2 тривіальні конструктори, метод deltaSpectrum, який будує вищеописану діагональ, метод ascend, який приймаючи масу нової амінокислоти, згенеровану раніше діагональ спектру та кількість, на яку зростає бал нащадка даного пептиду, власне, будує нащадка. Цей та описаний вище метод використовуються найчастіше і парно, особливо deltaSpectrum, результати якого потрібні для фільтрування неоптимальних кандидатів. Відповідно в них максимально використовується кеш. Оптимальними, як описано вище, вважаються ті кандидати, що мають найменшу довжину у своєму класі і чий клас має бал достатній для проходу порогу.

Метод cyclicSpectrumAddition будує «циклічну» частину спектру, ту, що позначена квадратними дужками у прикладі вище. Вона використовується для опрацювання кандидатів, які претендують на звання лідера.

Також використовується клас Leader – полегшена версія кандидату, яка не зберігає поля потрібні для створення наступних кандидатів, адже лідер – вінець еволюції кандидата і тому нічого не породжує.

```
1       public static class Leader {
2           public final int[] acids;
3           public final int score;
```

```
4
5       public Leader(int[] acids, int score) {
6           this.acids = acids;
7           this.score = score;
8       }
9   }
```

4.3 Опис алгоритму пошуку пептидів за заданим експериментальним спектром з використанням згортки спектра (convolutionCyclopeptideSequencing)

Наведені вище реалізації тестувалися виключно на непротеїногенних амінокислотах, яких всього 20. При зростанні кількості розглядуваних амінокислот, по-перше, збільшується час виконання алгоритму, по-друге, знижується його точність, адже більша конкуренція з боку «зайвих» амінокислот, тобто таких, що не входять до складу шуканого пептиду, збільшує ймовірність відкидання правильних кандидатів та лідерів. Відтак, існує потреба у зменшенні кількості розглядуваних амінокислот. Якщо спробувати узагальнити алгоритм до використання з усіма амінокислотами, то їх кількість грандіозно велика, тому за основу слід брати експериментальний спектр. Він містить помилки, які слід брати до уваги, шукаючи усі доцільні амінокислоти, адже якщо бодай одна амінокислота наявна у шуканому пептиді буде відсутня в обчисленнях, їх результат буде істотно спотворений. Для виявлення потенційно доцільних амінокислот використовується згортка експериментального спектра, тобто набір усіх часткових різниць значень спектра. Сформована мультимножина фільтрується, відповідно до переданих параметрів, адже не кожне числове значення може відповідати масі амінокислоти. Отримана колекція і є результатом функції spectrumConvolution.

```
01  public static Multiset<Integer> spectrumConvolution(final int[] spectrum, final int
minAcidMass, final int maxAcidMass) {
02    Multiset<Integer> convolution = HashMultiset.create();
03    for (int I = 0; I < spectrum.length - 1; i++) {
04       for (int j = I + 1; j < spectrum.length; j++) {
05          int diff = spectrum[j] - spectrum[i];
06          if (diff < minAcidMass) continue;
07          if (diff > maxAcidMass) break;
08          convolution.add(diff);
09       }
10    }
11    return convolution;
12 }
```

Описана вище функція застосовується у convolutionCyclopeptideSequencing, яка насправді обгортає leaderboardCyclopeptideSequencing, передаючи на вхід масив з n мас, які мали найвищу частоту у згортці експериментального спектру. Хоч наведений нижче код є очевидним, звернемо увагу на порядок виклику

конвеєрних функцій sorted та distinct. Завдяки відсортованій природі вхідних даних, distinct виконується швидше.

```
01    public static List<int[]> convolutionCyclopeptideSequencing(final int[] spectrum,
final int n, final int m, final int minAcidMass, final int maxAcidMass) {
02        Multiset<Integer> convolution = spectrumConvolution(spectrum, minAcidMass,
maxAcidMass);
03        Set<Multiset.Entry<Integer>> massFreq = convolution.entrySet();
04        int minViableCount = massFreq.stream()
05                .map(Multiset.Entry::getCount)
06                .sorted(Comparator.reverseOrder())
07                .distinct()
08                .skip(m - 1)
09                .findFirst()
10                .orElse(1);
11        int[] acidsPool = massFreq.stream()
12                .filter(x -> x.getCount() >= minViableCount)
13                .mapToInt(Multiset.Entry::getElement)
14                .toArray();
15        return leaderboardCyclopeptideSequencing(spectrum, acidsPool, n);
16    }
```

Висновки

У результаті виконання даної курсової роботи було здійснено знайомство з численними обчислювальними методами роботи з біологічними сутностями, зокрема:

- Пошук регуляторних мотивів
- Алгоритми зчитування
- Робота з амінокислотами
- Відтворення пептидів за заданим спектром

Розглянули такі типи алгоритмів як жадібні алгоритми, алгоритми базовані на випадковостях тощо. На додачу до цього було проведено практику написання запитів-тестів, які дозволяють як демонструвати працездатність запропонованого функціоналу, так і являють собою доказ його правильності, адже вхідні та очікувані вихідні дані для їх проведення взято з курсів «Biology Meets Programming: Bioinformatics for Beginners» та «Genome Sequencing (Bioinformatics II)». Оскільки алгоритми, що використовують випадковості вимагають особливого підходи при перевірці, для них було створено Unit тести, відповідно практику в їх написанні також було набуто.

Внаслідок неочікуваної зміни ресурсу виділеного на виконання даної роботи вже під час її створення довелося змінити пріоритети та перепланувати очікувані результати. Це стало практикою для гнучкого ставлення до проекту, перевірило вміння швидко орієнтуватися «посеред переправи» і трохи «змінити коней». Через це довелося скасувати графічний інтерфейс застосунку та значною мірою спростити серверну його частину, сконцентрувавшись на алгоритмах. На щастя, темою минулорічної роботи була саме розробка клієнт-серверних Web-застосувань з використанням Spring Boot framework та ReactJS, тому втрата додаткової практики з використання цих технологій може вважатися мінімальною для вмінь, знань та навичок виконавця.

Оскільки вступний курс максимально наполегливо радив використовувати Python 3 було також отримано практику з написання коду на цій мові. Усвідомлення того, чому ця мова використовується за замовчанням у наукових цілях замість Java прийшло тоді, коли написані на Python алгоритми довелося перекладати, зокрема показовою була робота з асоціативними масивами, яка в «пітона» реалізована через оператор квадратних дужок, а в «жабі» - як звичайний метод, через крапку. Було відчуто різницю між легкою, апріорі узагальненою динамічною типізацією мови Python 3 та прискіпливою, але інформативною статичною типізацією Java 8, особливо помітною вона була при перекладі коду, вік якого подекуди сягав кількох місяців.

Процес імплементації алгоритмів не можна назвати проблемним. Основні складності, що виникали були пов’язані із необхідністю перекласти код після тривалої перерви. Під час проходження курсів план був такий, щоб дійти на стільки далеко, на скільки вистачить часу, а потім з отриманих алгоритмів відібрати найцікавіші, тому описувати усі імплементації паралельно з написанням коду було не доцільно і навіть неможливо, адже код писався на Python 3, а кінцевий результат мав бути на Java 8. Перекладати легше, ніж вчити, тому було б нерозумно витратити цінний літній час на те, що можна робити і в більш напружений навчальний період. Також проблемою стало тестування алгоритмів з використанням випадковостей, адже правильні результати навіть для тих самих вхідних даних завжди різні. Звичайно, можна налаштувати генератор випадкових чисел, щоб він повертав визначені значення щоразу, тоді можна легко перевіряти правильність роботи відповідних алгоритмів. Однак, в такому разі буде частково втрачено сенс самого підходу, адже як зазначалося вище, сила рандомізованих алгоритмів – у їх дешевизні, що дозволяє запускати їх багато разів отримуючи різні результати, які можна порівнювати між собою та шукати кращі. Для того, щоб сісти одразу на два стільці було вирішено скористатися бібліотекою Mockito і надаючи доступ до випадкових чисел через окрему залежність замінити її під час тестування на підробку, яка повертатиме очікувані значення.

Додатки

Додаток А. Допоміжні функції для знаходження факторів транскрипції.

Розділ 1. Пошук рядка-консенсусу.

```
01    private static String consensus(List<String> motifs) {
02        Map<Character, List<Integer>> count = countNucsPositions(motifs);
03        StringBuilder consensus = new StringBuilder();
04        for (int i = 0; i < motifs.get(0).length(); i++) {
05            int m = 0;
06            char frequentSymbol = 0;
07            for (char symbol : "ACTG".toCharArray()) {
08                if (count.get(symbol).get(i) > m) {
09                    m = count.get(symbol).get(i);
10                    frequentSymbol = symbol;
11                }
12            }
13            consensus.append(frequentSymbol);
14        }
15        return consensus.toString();
16    }
```

Розділ 2. Обчислення кількості входжень кожного нуклеотиду ДНК на позиціях кожної колонки переданої матриці уривків.

```
01    private static Map<Character, List<Integer>> countNucsPositions(List<String> motifs)
{
02        int k = motifs.get(0).length();
03        Map<Character, List<Integer>> count = new HashMap<>();
04        for (char symbol : "ACGT".toCharArray()) {
05            count.put(symbol, new ArrayList<>());
06            for (int i = 0; i < k; i++)
07                count.get(symbol).add(0);
08        }
09        for (String motif : motifs)
10            for (int j = 0; j < k; j++) {
11                char symbol = motif.charAt(j);
12                count.get(symbol).set(j, count.get(symbol).get(j) + 1);
13            }
14
15        return count;
16    }
```

Розділ 3. Підрахунок розбіжностей в матриці факторів транскрипції.

```
1    private static int score(List<String> motifs) {
2        String cons = consensus(motifs);
3        int score = 0;
4        for (String motif : motifs)
5            for (int j = 0; j < motifs.get(0).length(); j++)
6                if (motif.charAt(j) != cons.charAt(j))
7                    ++score;
8        return score;
9    }
```

Розділ 4. Пошук k-мера, що має найбільшу імовірність бути правильним.

```
01    private static String profileMostProbableKmer(String text, int k, Map<Character,
List<Double>> profile) {
02        double maxProb = 0;
03        Integer resStartIdx = null;
04        for (int i = 0; i < text.length() - k + 1; i++) {
05            double nextProb = pr(text.substring(i, i + k), profile);
```

```
            if (nextProb > maxProb) {
                maxProb = nextProb;
                resStartIdx = i;
            }
        }
        return resStartIdx != null ? text.substring(resStartIdx, resStartIdx + k) :
text.substring(0, k);
    }
```

Розділ 5. Обчислення імовірності k-мера бути правильним.

```
    private static double pr(String text, Map<Character, List<Double>> profile) {
        double init = 1;
        for (int i = 0; i < text.length(); i++)
            init *= profile.get(text.charAt(i)).get(i);
        return init;
    }
```

Додаток Б. Допоміжні функції для реконструкції геному

Розділ 1. Код трансформації текстового графу у числовий.

```
01    private static Pair<Map<Integer, String>, Map<Integer, Deque<Integer>>>
transformGraph(Map<String, List<String>> graph) {
02        int indexAccum = 0;
03        Map<String, Integer> mapper = new HashMap<>();
04        Map<Integer, String> remapper = new HashMap<>();
05        Map<Integer, Deque<Integer>> outGraph = new HashMap<>();
06        for (Map.Entry<String, List<String>> node : graph.entrySet()) {
07            if (!mapper.containsKey(node.getKey())) {
08                mapper.put(node.getKey(), ++indexAccum);
09                remapper.put(indexAccum, node.getKey());
10            }
11            outGraph.put(mapper.get(node.getKey()), new ArrayDeque<>());
12            for (String adjV : node.getValue()) {
13                if (!mapper.containsKey(adjV)) {
14                    mapper.put(adjV, ++indexAccum);
15                    remapper.put(indexAccum, adjV);
16                }
17                outGraph.get(mapper.get(node.getKey())).add(mapper.get(adjV));
18            }
19        }
20        return Pair.of(remapper, outGraph);
21    }
```

Розділ 2. Імплементація алгоритму пошуку Ейлерового шляху.

```
01  private static List<Integer> eulerianPath(Map<Integer, Deque<Integer>> intGraph) {
02        Optional<Integer> maybeMax = Stream.concat(intGraph.keySet().stream(),
intGraph.values().stream().flatMap(Collection::stream)).max(Integer::compareTo);
03        if (maybeMax.isEmpty()) return Collections.emptyList();
04        int[] dDegrees = new int[maybeMax.get() + 1];
05        for (Map.Entry<Integer, Deque<Integer>> node : intGraph.entrySet()) {
06            dDegrees[node.getKey()] += node.getValue().size();
07            for (Integer v : node.getValue())
08                dDegrees[v]--;
09        }
10        int starter = -1;
11        for (int I = 0; I < dDegrees.length; i++) {
12            if (dDegrees[i] == 1) starter = I;
13        }
14        if (starter == -1) return Collections.emptyList();
15        Deque<Map.Entry<Integer, Deque<Integer>>> stack = new ArrayDeque<>();
16        stack.add(Map.entry(starter, intGraph.get(starter)));
17        List<Integer> cycle = new ArrayList<>();
18        while (!stack.isEmpty()) {
19            Map.Entry<Integer, Deque<Integer>> node = stack.removeLast();
20            Integer v = node.getKey();
21            Deque<Integer> adj = node.getValue();
22            while (!adj.isEmpty()) {
23                stack.add(Map.entry(v, adj));
24                v = adj.removeFirst();
25                adj = intGraph.getOrDefault(v, new ArrayDeque<>());
26            }
27            cycle.add(v);
28        }
29        return Lists.reverse(cycle);
30    }
```

Додаток В. Адаптовані допоміжні функції для роботи з парами зчитувань.

Розділ 1. Побудова графу Де Брейна за парами зчитувань.

```
01    private static Map<Pair<String, String>, List<Pair<String, String>>>
pairedDeBruign(List<Pair<String, String>> patterns) {
02        Map<Pair<String, String>, List<Pair<String, String>>> prefMap = new HashMap<>();
03        for (Pair<String, String> readPair : patterns) {
04            Pair<String, String> nextPrefix = Pair.of(getPrefix(readPair.getFirst()),
getPrefix(readPair.getSecond()));
05            prefMap.putIfAbsent(nextPrefix, new ArrayList<>());
06            prefMap.get(nextPrefix).add(readPair);
07        }
08        Map<Pair<String, String>, List<Pair<String, String>>> adjList = new HashMap<>();
09        for (Pair<String, String> prefPair : prefMap.keySet()) {
10            adjList.put(prefPair, new ArrayList<>());
11            for (var patternPair : prefMap.get(prefPair))
12                adjList.get(prefPair).add(Pair.of(getSuffix(patternPair.getFirst()),
getSuffix(patternPair.getSecond())));
13        }
14        return adjList;
15    }
```

Розділ 2. Код трансформації текстового парованого графу у числовий.

```
01    private static <T> Pair<Map<Integer, T>, Map<Integer, Deque<Integer>>>
transformGraph2(Map<T, List<T>> graph) {
02        int indexAccum = 0;
03        Map<T, Integer> mapper = new HashMap<>();
04        Map<Integer, T> remapper = new HashMap<>();
05        Map<Integer, Deque<Integer>> outGraph = new HashMap<>();
06        for (Map.Entry<T, List<T>> node : graph.entrySet()) {
07            if (!mapper.containsKey(node.getKey())) {
08                mapper.put(node.getKey(), ++indexAccum);
09                remapper.put(indexAccum, node.getKey());
10            }
11            outGraph.put(mapper.get(node.getKey()), new ArrayDeque<>());
12            for (T adjV : node.getValue()) {
13                if (!mapper.containsKey(adjV)) {
14                    mapper.put(adjV, ++indexAccum);
15                    remapper.put(indexAccum, adjV);
16                }
17                outGraph.get(mapper.get(node.getKey())).add(mapper.get(adjV));
18            }
19        }
20        return Pair.of(remapper, outGraph);
21    }
```

Розділ 3. Алгоритм знаходження усіх Ейлерових шляхів.

```
01    private static List<List<Integer>> allEulerianPaths(Map<Integer, Deque<Integer>> graph,
Deque<Integer> stack) {
02        List<Integer> singlePath = new ArrayList<>();
03        List<List<Integer>> paths = new ArrayList<>();
04        while (!stack.isEmpty()) {
05            Integer v = stack.removeLast();
06            Deque<Integer> adj = graph.getOrDefault(v, new ArrayDeque<>());
07            while (!adj.isEmpty()) {
08                stack.add(v);
09                if (adj.size() == 1) {
10                    v = adj.removeFirst();
11                    adj = graph.getOrDefault(v, new ArrayDeque<>());
12                } else {
13                    for (Integer adjV : adj) {
14                        @SuppressWarnings("unchecked")
15                        Map<Integer, Deque<Integer>> localGraph =
Objects.requireNonNull(graph.getClass()
16
```

```
.cast(SerializationUtils.deserialize(SerializationUtils.serialize(graph))));
17                          localGraph.get(v).remove(adjV);
18                          @SuppressWarnings("unchecked")
19                          Deque<Integer> localStack = Objects.requireNonNull(stack.getClass()
20
.cast(SerializationUtils.deserialize(SerializationUtils.serialize(stack))));
21                          localStack.add(adjV);
22                          for (List<Integer> pathTail : allEulerianPaths(localGraph, localStack))
23                              paths.add(ListUtils.union(singlePath, pathTail));
24                          return paths;
25                      }
26                  }
27              }
28              singlePath.add(v);
29          }
30          return List.of(singlePath);
31      }
```

Додаток Д. Тест продуктивності реалізацій алгоритму пошуку пептидів за заданим теоретичним спектром з кешуванням та без.

Розділ 1. Програмний код алгоритму без кешування.

```
01  public static Collection<int[]> perfectSpectrumPeptides2(final int[] spectrum) {
02    Map<Integer, Integer> spectrumSet = new HashMap<>();
03    for (int I = 1; I < spectrum.length; i++) {
04       spectrumSet.compute(spectrum[i], (acid, num) -> num == null ? 1 : num + 1);
05    }
06    Set<Integer> basicAcids = spectrumSet.keySet()
07          .stream().filter(x -> x <=
Constants.AMINO_ACID_MASS.get('W')).collect(Collectors.toSet());
08
09    basicAcids.retainAll(Constants.AMINO_ACID_MASS.values());
10
11    BiMap<Integer, Integer> basicOrder = HashBiMap.create(basicAcids.size());
12    for (Integer acidMass : basicAcids)
13       basicOrder.put(basicOrder.size(), acidMass);
14
15    Queue<int[]> consistentPeptides = new ArrayDeque<>();
16    List<int[]> peptides = new ArrayList<>();
17    basicAcids.forEach(acid -> consistentPeptides.add(new int[]{acid}));
18    while (consistentPeptides.size() != 0) {
19       int[] currPeptide = consistentPeptides.poll();
20       for (Integer acidMass : basicAcids) {
21          int[] nextPeptide = Arrays.copyOf(currPeptide, currPeptide.length + 1);
22          nextPeptide[nextPeptide.length - 1] = acidMass;
23
24          int peptideMass = 0;
25          for (int I = 0; I < currPeptide.length; peptideMass += currPeptide[i++]) ;
26          peptideMass += acidMass;
27
28          if (peptideMass > spectrum[spectrum.length - 1]) continue;
29
30          if (peptideMass == spectrum[spectrum.length - 1]
31                && Arrays.equals(spectrum, theoreticalPeptideSpectrum(nextPeptide, true))) {
32             peptides.add(nextPeptide);
33             continue;
34          }
35          consistentPeptides.add(nextPeptide);
36       }
37    }
38    return peptides;
39 }
```

Розділ 2. Програмний код тесту.

```
BufferedReader br = new BufferedReader(new FileReader("input.txt"));
int[] spectrum = Ints.toArray(Stream.of(br.readLine().split("\\s"))
        .map(Integer::parseInt).collect(Collectors.toList()));

StopWatch sw = new StopWatch();
StopWatch sw2 = new StopWatch();

sw.start();
Collection<int[]> peptides = amino_acids.TheoreticalPeptides.perfectSpectrumPeptides(spectrum);
sw.stop();
sw2.start();
Collection<int[]> peptides2 = amino_acids.TheoreticalPeptides.perfectSpectrumPeptides2(spectrum);
sw2.stop();

assert peptides.equals(peptides2);

System.out.println("fst: " + sw.getTotalTimeMillis());
```

```
System.out.println("snd: " + sw2.getTotalTimeMillis());

for(int[] peptide: peptides) {
    System.out.println(String.join("-",
            Arrays.stream(peptide).mapToObj(String::valueOf).toArray(String[]::new)));
}
```

Розділ 3. Тестові дані та результати виконання тесту.

Вхідні дані:

0 97 97 99 99 113 131 163 186 186 196 198 210 228 262 283 295 299 317 327 349 361 396 396 414 426 430 446 448 458 513 527 527 545 547 559 582 589 612 624 626 644 644 658 713 723 725 741 745 757 775 775 810 822 844 854 872 876 888 909 943 961 973 975 985 985 1008 1040 1058 1072 1072 1074 1074 1171

Вихідні дані:

fst: 887

snd: 6796

97-113-186-131-97-99-99-163-186

97-131-186-113-97-186-163-99-99

97-99-99-163-186-97-113-186-131

97-186-163-99-99-97-131-186-113

113-97-186-163-99-99-97-131-186

113-186-131-97-99-99-163-186-97

131-97-99-99-163-186-97-113-186

131-186-113-97-186-163-99-99-97

163-99-99-97-131-186-113-97-186

163-186-97-113-186-131-97-99-99

99-97-131-186-113-97-186-163-99

99-163-186-97-113-186-131-97-99

99-99-97-131-186-113-97-186-163

99-99-163-186-97-113-186-131-97

186-97-113-186-131-97-99-99-163

186-113-97-186-163-99-99-97-131

186-131-97-99-99-163-186-97-113

186-163-99-99-97-131-186-113-97

Список використаних джерел

1. MIT - Massachusetts Institute of Technology. URL: http://web.mit.edu/manoli/www/thesis/Chapter3.html (дата звернення: 16.01.2022).

2. Base pair. Genome.gov. URL: https://www.genome.gov/genetics-glossary/Base-Pair (дата звернення: 16.01.2022).

3. BioinformaticsAlgorithms.github.io | Homepage for Bioinformatics Algorithms Materials. URL: https://bioinformaticsalgorithms.com/images/Motifs/nfkb.png (дата звернення: 16.01.2022).

4. BioinformaticsAlgorithms.github.io | Homepage for Bioinformatics Algorithms Materials. URL: https://bioinformaticsalgorithms.com/images/Assembly/sequencing_overview.png (дата звернення: 16.01.2022).

5. Exploding Newspapers. Stepik: online education. URL: https://stepik.org/lesson/196/step/6?unit=8239 (date of access: 16.01.2022).

6. Contig. Genome.gov. URL: https://www.genome.gov/genetics-glossary/Contig (date of access: 16.01.2022).

7. Catalog – Stepik. URL: https://stepik.org/media/attachments/lessons/96/RNA_codon_table_1.txt (дата звернення: 16.01.2022).

8. BioinformaticsAlgorithms.github.io | Homepage for Bioinformatics Algorithms Materials. URL: https://bioinformaticsalgorithms.com/images/Antibiotics/genetic_code.png (дата звернення: 16.01.2022).

9. Nonribosomal peptides synthetases and their applications in industry - Sustainable Chemical Processes. SpringerOpen. URL: https://sustainablechemicalprocesses.springeropen.com/articles/10.1186/s40508-016-0057-6 (дата звернення: 16.01.2022).

10. What is Mass Spectrometry?. Broad Institute. URL: https://www.broadinstitute.org/technology-areas/what-mass-spectrometry (date of access: 16.01.2022).