МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
Национальный исследовательский ядерный университет «МИФИ»

Институт
интеллектуальных кибернетических систем

Кафедра №22 «Кибернетика»

Направление подготовки 09.03.04 Программная инженерия

Пояснительная записка

к учебно-исследовательской работе студента на тему:

Разработка компьютерной системы для поиска дисперсных повторов в разнообразных геномах

Группа Б21-514

Студент _____ (подпись) Шамаев С. Д. (ФИО)

Руководитель _____ (подпись) Короткова М.А. (ФИО)

Научный консультант _____ (подпись) _____ (ФИО)

Оценка руководителя _____ (0-15 баллов) Оценка консультанта _____ (0-15 баллов)

Итоговая оценка _____ (0-100 баллов) ECTS _____

Комиссия

Председатель _____ (подпись) _____ (ФИО)

_____ (подпись) _____ (ФИО)

_____ (подпись) _____ (ФИО)

_____ (подпись) _____ (ФИО)

Москва 2024
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский ядерный университет «МИФИ»

Институт интеллектуальных кибернетических систем

КАФЕДРА КИБЕРНЕТИКИ

Задание на УИР

Студенту гр. Б21-514 (группа) Шамаеву Сергею Денисовичу (фио)

ТЕМА УИР

Разработка компьютерной системы для поиска дисперсных повторов в разнообразных геномах

ЗАДАНИЕ

№ п/п	Содержание работы	Форма отчетности	Срок исполнения	Отметка о выполнении Дата, подпись
1.	**Аналитическая часть**			
1.1.	Изучение и сравнительный анализ методов поиска дисперсных повторов с целью выбора метода для реализации	Пункт ПЗ	25.02.2024	
1.2.	Изучение биологических особенностей повторяющихся последовательностей	Пункт ПЗ	05.03.2024	
1.3.	Изучение существующих систем поиска повторяющихся последовательностей в геномах	Пункт ПЗ	05.03.2024	
1.4.	Анализ возможностей изменения имеющихся методов поиска применительно к задаче проекта.	Пункт ПЗ	15.03.2024	
1.5.	*Оформление расширенного содержания пояснительной записки (РСПЗ)*	Текст РСПЗ	27.03.2024	
2.	**Теоретическая часть**			
2.1.	Выбор метода для реализации системы поиска дисперсных повторов	Алгоритм поиска повторов	10.03.2024	
2.2.	Разработка методики оценки статистической значимости найденных повторов	Пункт ПЗ	14.03.2024	
2.3.	Разработка структур данных для реализации поиска и представления результатов	Пункт ПЗ	25.03.2024	
3.	**Инженерная часть**			
3.1.	Проектирование системы поиска дисперсных повторов в разнообразных геномах	Пункт ПЗ	15.04.2024	
3.2.	Разработка архитектуры системы поиска	Пункт ПЗ	18.04.2024	
3.3.	Разработать требований к представлению результатов	Пункт ПЗ	20.04.2024	

	поиска			
3.4.	Результаты проектирования оформить с помощью UML			
4.	**Технологическая и практическая часть**			
4.1.	Реализовать систему поиска дисперсных повторов	Исполняемые файлы, исходный текст	10.05.2024	
4.2.	Разработать тестовые примеры для реализованной системы поиска дисперсных повторов. Протестировать систему с использованием разработанных тестов.	Исполняемые файлы, исходные тексты тестов и тестовых примеров	15.05.2024	
4.3.	Реализация должна позволять обнаруживать дисперсные повторы с частотой замен не менее 1 замены на нуклеотид			
4.4.	При реализации использовать технологию распределенных вычислительных систем на базе кластерной архитектуры			
5.	*Оформление пояснительной записки (ПЗ) и иллюстративного материала для доклада*.	Текст ПЗ, презентация	22.05.2024	

ЛИТЕРАТУРА

1.	Crooks GE, Hon G, Chandonia JM, Brenner SE. 2004. WebLogo: a sequence logo generator. *Genome Res***14**: 1188–1190.
2.	2. KorotkovE, SuvorovaY, KostenkoD, KorotkovaM. 2023. Search for Dispersed Repeats in Bacterial Genomes Using an Iterative Procedure. *Int J MolSci***24**: 10964. https://www.mdpi.com/1422-0067/24/13/10964/htm (Accessed September 12, 2023).
3.	3. Liao X, Zhu W, Zhou J, Li H, Xu X, Zhang B, Gao X. 2023. Repetitive DNA sequence detection and its role in the human genome. *CommunBiol***6**. https://pubmed.ncbi.nlm.nih.gov/37726397/ (Accessed January 11, 2024).
4.	4. Storer JM, Hubley R, Rosen J, Smit AFA. 2022. Methodologies for the De novo Discovery of Transposable Element Families. *Genes (Basel)***13**. https://pubmed.ncbi.nlm.nih.gov/35456515/ (Accessed February 2, 2023).
5.	5. Versalovic J, Lupski JR. 1998. Interspersed Repetitive Sequences in Bacterial Genomes. *BactGenomes* 38–48. https://link.springer.com/chapter/10.1007/978-1-4615-6369-3_5 (AccessedJanuary 14, 2024).
6.	
7.	

Дата выдачи задания: Руководитель _____ Короткова М. А.
(ФИО)

« 25 » февраля 2024г. Студент _____ Шамаев С.Д.
(ФИО)

Оглавление

Реферат

Пояснительная записка содержит 35 страниц. Количество источников – 26. Количество рисунков – 7. Количество приложений – 1.

Ключевые слова: множественное выравнивание, динамические программирование, дисперсные повторы.

Целью данной работы является разработка системы поиска дисперсных повторов в последовательностях со скрытой периодичностью.

В первом разделе описывается анализ существующих методов и алгоритмов систем поиска дисперсных повторов.

Второй раздел посвящён теоретической составляющей исследовательской работы, а именно раскрытию выбранного метода реализации поиска и системы оценки статистической значимости найденных повторов.

В третьем разделе описана инженерная составляющая проектируемой системы поиска дисперсных повторов и разработке архитектуры данной системы.

В четвертом разделе приведено описание конкретной реализации системы поиска дисперсных повторов, покрытие его тестами, а также предоставлен анализ результатов этого тестирования.

В заключении приведены итоги исследовательской работы, конечный анализ полученной системы поиска дисперсных повторов и её результаты.

Введение

Применение и создание математических моделей для исследования последовательностей символов является одним из важнейших направлений биоинформатики. Подобная область исследований позволяет находить общие элементы разнообразных геномов, выявлять закономерности и строить предположения о функциональном аспекте тех или иных цепочек ДНК. Развитие новых математических методов дает возможность производить структурную аннотацию ещё не описанных частей генома и выявлять новые закономерности.

Одной из подзадач в области математического анализа последовательностей является поиск скрытой периодичности. Скрытой периодичностью можно считать периодичность, где сходство любых двух периодов не является значимым [1]. Таким образом, выявить данную периодичность парным выравниванием не представляется возможным. Однако скрытую периодичность можно обнаружить с помощью множественного выравнивания, не основанного на парном выравнивании. В предложенном представлении периоды будут записаны в набор последовательностей, собранный для поиска множественного выравнивания. Полученное выравнивание может быть статистически значимым, в то время как парное выравнивание между любыми двумя последовательностями из этого набора – нет. В подобном случаеможно говорить, что скрытая периодичность была обнаружена.

В геноме существуют различные структурные закономерности. Одними из них являются дисперсные повторы. Дисперсные повторы – это повторы символов (нуклеотидов в геноме), которые не идут последовательно друг за другом. Точный поиск мест расположения подобных повторов способен помочь определить эволюционное происхождение геномных последовательностей, а также выявить их свойства. К настоящему моменту времени существует множество методов и алгоритмов поиска дисперсных повторов [2]. Но, несмотря на большое разнообразие существующих методов, задача поиска дисперсных повторов не может считаться решенной. Это связанно с тем, что все существующие алгоритмы не могут найти статистически значимые повторы при слишком большом количестве накопленных мутаций.

Таким образом, задача поиска дисперсных повторов является актуальной, а её решение – востребованной в мировом сообществе.

Раздел 1. Анализ задачи поиска дисперсных повторов

В данном разделе работы представлен анализ различных методов поиска дисперсных повторов, а также разнообразных систем их реализации. Выявлены и изучены существующие закономерности и биологические особенности повторяющихся последовательностей, представлены возможные изменения уже существующих методов поиска для исследования слабо гомологичных последовательностей.

1.1. Изучение и сравнительный анализ методов поиска дисперсных повторов с целью выбора метода для реализации

В настоящее время существует множество возможных методов поиска повторов. Методы, основанные на спектральном подходе, такие как преобразование Фурье [3], Вейвлет-преобразование [4] и другие [5], – не способны эффективно выявлять повторы при наличии вставок или деленций. Это методы единого статистического анализа короткосрочных и долгосрочных корреляций между различными нуклеотидами в цепях геномной ДНК. Подход основан на взаимном изучении спектров структурных факторов Фурье и парных корреляционных функций [5]. Подобные методы не способны эффективно работать с вставками и деленциями – мутациями, связанные с добавлением нового нуклеотида в геном или его выпадением. Поэтому основные подходы, позволяющие решить задачу поиска дисперсных повторов в условиях присутствия всевозможных мутаций, связаны с динамическим программированием.

Большинство из подобных подходов к реализации разбивают задачу на две независимые части [6]. Первая часть – часть А – генерирует или получает начальный набор предполагаемых дисперсных повторов, тогда как вторая часть – часть Б – отвечает за задачу поиска в исходной последовательности тех подпоследовательностей, которые бы удовлетворяли условию сходства с набором, полученным в А. Решение задачи Б тесно связано с применяемыми алгоритмами и системами поиска, описанных в пункте 1.3. В свою очередь, существуют несколько фундаментально различных способов решить задачу А.

Первый подход связан с использованием каких-либо заранее найденных повторов, полученных из баз данных уже исследованных периодов. Данный способ решения ограничен невозможностью выйти за рамки уже найденных повторов, тогда как этот набор нельзя считать полным. Это связанно с тем, что с течением времени и усовершенствованием методов находятся всё новые и новые дисперсные повторы, что доказывает существование ещё не исследованных дисперсных повторов. К тому же данный подход не способен работать в условиях произвольного алфавита, то есть сильно зависит от глубины исследования проблемной области. Это означает, что подобный

метод не способен решить задачу поиска дисперсных повторов в общем случае, и, к примеру, найти периодичность в человеческой речи. При наличии обучающей выборки способны работать подобные алгоритмы, основанные на скрытых Марковских цепях, но в большинстве случаев такие выборки отсутствуют. Метод поиска дисперсных повторов с помощью скрытых Марковских цепей заключается в построении матрицы вероятностных переходов, что возможно при большом объёме анализируемых последовательностей или исходных данных [7]. Подобная методика заключается в построение «профиля»: по частоте встречаемости символа строится матрица вероятностных переходов. Таким образом, данный подход не дает возможности найти новые разновидности повторов, однако успешно применяется при решении задачи поиска заранее известных возможных повторений в новой, ещё не исследованной, последовательности.

Альтернативным способом построения набора предполагаемых повторов (решение задачи А) является его создание каким-либо способом без использования уже известных данных, не входящих в исследуемую последовательность. Данный подход хорош тем, что он не зависит от степени исследования проблемной области. Однако, при накоплении слишком большого числа мутаций, задача поиска исследуемого набора первичных дисперсных повторов не всегда может иметь эффективное решение. Тем не менее, подобный метод поиска de novo использует только данные самой последовательности, тем самым позволяя решать задачи поиска дисперсных повторов в последовательностях с произвольным алфавитом. Данный метод широко применяется в различных системах и алгоритмах (см. пункт 1.3). В зависимости от реализации, этот метод основан на поиске по сходству, подсчету слов или подходах, основанных на сигнатурах [2]. Эти методы основаны на использование определенных метрик, которые оценивают степень схожести объектов. Стоит отметить, что в подобном решении задачи А появляется возможность обнаружить ранее неизвестные дисперсные повторы, тем самым обновляя базу известных повторов и расширяя возможности первого метода решения блока А.

Таким образом, несмотря на многообразие различных методов, каждый из них имеет свои ограничения. Тем не менее, комбинирование нескольких подходов может позволить существенно расширить возможность поиска дисперсных повторов, что является целью данной исследовательской работы.

1.2. Изучение биологических особенностей повторяющихся последовательностей

В настоящее время было обнаружено, что в разнообразных геномах существуют множество повторяющихся цепочек. До определённого момента подобные повторы считали мусорными, однако спустя некоторое время была выявлена их важность.

Несмотря на то, что некоторые их функции до сих пор не ясны, многие из них были распознаны. К примеру, функциями повторных нуклеотидов является регуляция различных молекулярно-генетических процессов и становление источником генетической изменчивости [8].

Всего существует несколько видов повторов – тандемные и диспергированные (дисперсные). Первыми называются повторы, следующие непосредственно друг за другом без пропусков, в то время как для вторых между периодами наблюдаются некоторые другие участки генома. Каждый из этих данных типов имеет свои особенности и характеристики. В данной исследовательской работе основной темой изучения является второй тип повторов, на котором будет заострено внимание.

По функциональному значению и структурным особенностям все существующие дисперсные повторы, которые были обнаружены в геномах эукариот, можно разделить на пять классов: короткие диспергированные повторы (SINE), длинные диспергированные повторы (LINE), LTR-ретро-транспозоны, ДНКтранспозоны и другие [9]. У человека диспергированные повторы составляют около 45% всего генома, причем 21 % из них – длинные диспергированные повторы, 13 % – короткие диспергированные повторы, 8 % – LTR-ретро-транспозоны и 3 % – ДНК-транспозоны [10].

Каждый тип дисперсных повторов обладает своими функциональными свойствами, а также имеет отличие в длине, количестве повторов и занимаемой части генома. К примеру, длинные диспергированные повторы имеют несколько тысяч пар оснований, в то время как короткие диспергированные повторы имеют менее пятисот оснований.

Таким образом, существуют несколько вариантов шаблонных длин и количества оснований в геномах, степень которых можно оценить. Подобное знание позволяет сузить поиск возможных дисперсных повторов в геномах, выполняя нахождение определенного класса дисперсных повторов.

1.3. Изучение существующих систем поиска повторяющихся последовательностей в геномах

Как было показано в пункте 1.1., задачу поиска дисперсных повторов нередко разбивают на две непересекающиеся подзадачи А и Б. В связи с тем, какой из этих блоков реализует система поиска, её можно отнести к тому или иному классу.

Первыми выделяются программы, выполняющие задачу А (генерацию сравнительного набора дисперсных повторов) путём использования уже готовых баз данных. К таким системам относятся программы RepeatMasker [11], Censor[12] и MaskerAid [13]. Эти реализации опираются на базу данных Repbase [14]. Таким образом, данные системы способны хорошо находить известные дисперсные повторы в новых

последовательностях, однако не способны решить задачу поиска новых дисперсных повторов, не входящих в приведенную базу данных.

Другой реализацией подзадачи А является создания сравнительного набора путем анализа исходной последовательности. Используя данный подход, реализованы такие программы как RED [15], Recon [16], PILER [17], RepeatScout [18] и RepeatFinder с REPUter [19]. Подобные программы основаны на применении различных математических методах. Данные программы способны находить новые, ещё не изученные дисперсные повторы, в отличие от предыдущего блока программ.

Также существуют системы, выполняющие решение задачи Б. К таким системам относятся BLAST [20], FASTA [21], MEGA [22] и nHMMER [23]. Каждая из программ имеет свои особенности и принимает различные структуры сравнительных дисперсных повторов. Некоторые способны работать по обобщенной последовательности, другие по конкретному набору не обработанных наборов повторов. Стоит отметить, что существуют программы, работающие по системе множественного выравнивания последовательностей (таких как nHMMER). Данное метод предобработки данных позволяет увеличить эффективность поиска, однако требует дополнительного решения задачи построения множественного выравнивания.

Таким образом, возможно использование комбинирования решений различных подзадач, что способно увеличить эффективность поиска дисперсных повторов. К примеру, опираясь на собственноручно построенные сравнительные наборы дисперсных повторов (далее предложенное решение задачи А), представляется возможным воспользоваться реализацией методов, предложенных системой nHMMER [23]. Подобное решение даёт шанс увеличить число статистически значимых, то есть тех, что не являются случайными, - найденных повторов.

1.4. Анализ возможностей изменения имеющихся методов поиска применительно к задаче проекта

Как было показано выше, для эффективного решения задачи Б необходимо улучшение решения задачи А. В связи с тем, что в данной исследовательской работе уклон будет ставится на поиске новых, еще не изученных дисперсных повторов, решение задачи А представляется путём получения последовательностей de novo. Таким образом, наиболее эффективным решением будет оптимизация и улучшение алгоритмов, опирающиеся на данные самой последовательности, без учетов имеющихся баз данных о дисперсных повторах в геноме.Таким образом, поиск дисперсных повторов может быть улучшен путем использования нового метода решения подзадачи А, основанного на итерационном алгоритме, а также оптимизацией решения подзадачи Б.

1.5. Выводы

1. Была проанализированы уже существующие методы поиска дисперсных повторов
2. Были изучены биологические особенности повторяющихся последовательностей
3. Был произведен анализ существующих алгоритмов и систем поиска
4. Был предложен способ создания, изменения и улучшения существующих методов

1.6. Постановка цели и задачи УИР

Основной целью данной работы является разработка системы поиска дисперсных повторов.

Основные задачи работы:

1. Анализ существующих методов поиска дисперсных повторов
2. Анализ алгоритмов и систем поиска дисперсных повторов
3. Выбор, создание и улучшение метода поиска дисперсных повторов
4. Разработка системы поиска дисперсных повторов
5. Анализ системы и результатов, основанных на полученной системе

Раздел 2. Теоретические основания создания системы поиска дисперсных повторов

В данном разделе осуществляется и приводится выбор методов реализации системы поиска дисперсных повторов, разрабатывается методика оценки статистической значимости полученных дисперсных повторов, а также описывается сам алгоритм поиска дисперсных повторов.

2.1. Выбор метода для реализации системы поиска дисперсных повторов.

Как было показано выше, основной проблемой существующих методов является отсутствие корректного начального набора подпоследовательностей (решение задачи A). В таком случае, ставится задача поиска начального набора de novo. Предлагаемым способом поиска повторов является отказ от анализа закономерностей исходной последовательности в сторону использования случайной генерации набора и итерационной процедуры его улучшения. В данном случае, получение сравнительного набора можно разделить на две подзадачи. Первая — выбор случайных подпоследовательностей и создание из них начального сравнительного набора. Вторая подзадача заключается в нахождении алгоритма улучшения сравнительного набора. Далее

при комбинации решений двух данных подзадач возможно получить алгоритм поиска дисперсных повторов.

Стоит отметить, что выполнение первой подзадачи стоит произвести не один раз, то есть получить несколько наборов начальных сравнительных подпоследовательностей, каждый из которых будет подвергнут оптимизации. Данное решение приведет к увеличению затрат времени на поиск, однако позволит не упустить дисперсные повторы в связи с неудачным выбором начальных подпоследовательностей. Итерационный алгоритм оптимизации может быть основан на методах, сходных с nHMMER, то есть использующий поиск по текущему сравнительному набору повторов (который способен подвергнуться улучшению, благодаря выполнению множественного выравнивания) новых, сходных с исходными, подпоследовательностей.

Таким образом, предложенный метод заключается в многократном создании случайных наборов подпоследовательностей, для каждого их которых будет применен итерационный алгоритм его улучшения. После будет выбран наилучший вариант, соответствующий одному случайно выбранному начальному набору.

2.2. Разработка методики оценки статистической значимости найденных повторов.

Для оценки статистической значимости множественного выравнивания последовательностей используют такие параметры, как Z-score [24], P-value [24] и E-value [25].

Z-score показывает, насколько необычно обнаруженное совпадение. В терминах статистики это расстояние (измеряемое как среднеквадратическое отклонение) данного уровня от среднего значения по набору данных. Чем больше Z-score, тем больше вероятность того, что наблюдаемое выравнивание появилось неслучайно. Это безразмерный статистический показатель, используемый для сравнения значений разной размерности или шкалой измерений.

Формула расчёта считается:

$$z = (x - \mu) / \sigma$$

Формула 2.1 – Расчёт Z-score

x — значение функции, μ — среднее значение, σ — стандартное отклонение. Степень схожести набора x определяется попарной схожестью любых двух подпоследовательностей [24].

Таким образом, для расчета Z-score необходимо сгенерировать случайные наборы последовательностей и определить, насколько исходная последовательность статистически отличается от подобных случайных подпоследовательностей.

P-value вероятность того, что выравнивание не лучше, чем случайное. Так как Z-score имеет схожее с нормальным распределение [24], то для него возможно по выдвинутой гипотезе о степени схожести подобной случайной, определить по уровню квантиля вероятность того, что искомая последовательность случайна.

E-value (expected value) [25]— это ожидаемое количество последовательностей с весом выравнивания, равным или большим веса для анализируемой последовательности, которые, вероятно, будут обнаружены при поиске в базе данных. Например, E-value 0,01 предполагает, что можно ожидать найти совпадение с как минимум таким же баллом, как и в текущем выравнивании, один раз в 100 случайных поисков в базе данных. То есть будет 1 ложноположительное значение.

Чем меньше величина E, тем достовернее выравнивание. При этом следует учитывать, что гомология ниже 50 % при больших значениях E-value, как правило, несущественна.

Значения E-value можно интерпретировать следующим образом:

1. E < 0,02 — вероятно, последовательности являются гомологами;
2. 0,02 < E < 1 — гомология не очевидна;
3. E > 1 — следует ожидать, что это случайное совпадение.

В данной исследовательской работе будет использоваться статистическая оценка E-value, реализация которого предложена в программе nHMMER [23].

2.3. Выводы

1. Были описаны методы оценки статистической значимости дисперсных повторов.
2. Были рассмотрены такие оценки статистической значимости как Z-score, P-value и E-value.
3. Был выбран метод оценки статистической значимости для системы поиска дисперсных повторов.

Раздел 3. Проектирование системы поиска дисперсных повторов.

В данном разделе будут описана конкретная методика проектирования системы, будет представлена архитектуры системы поиска дисперсных повторов, а также выработка системы требований к результату поиска.

3.1. Разработка архитектуры системы поиска.

В данном разделе приведен общий алгоритм системы поиска дисперсных повторов. Далее идёт пояснение конкретных блоков алгоритма. На вход поступает последовательность символов S длиной L (алфавит заранее задан). Далее создается Num2 количество начальных наборов подпоследовательностей Q(S, j). Для последующих

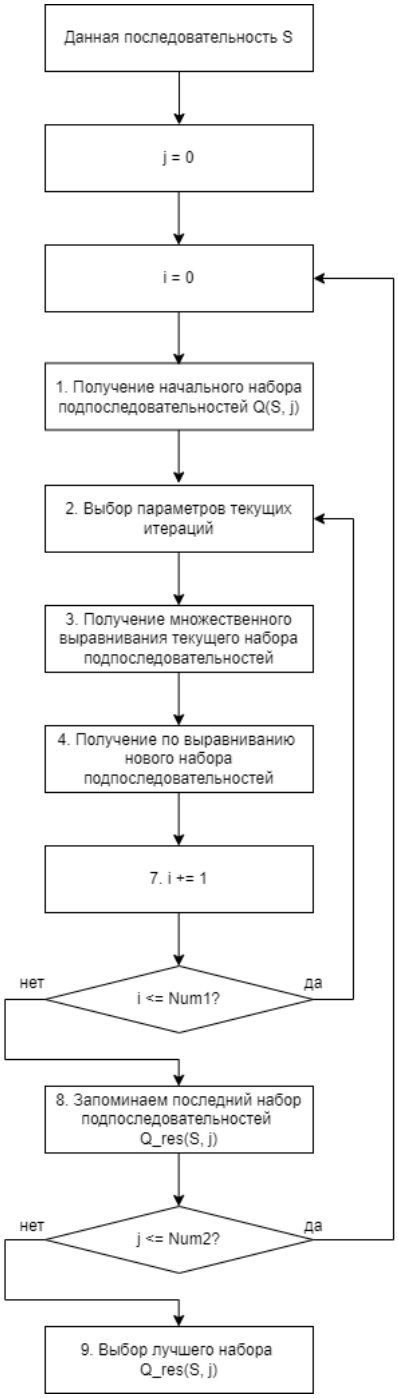

Рис 3.1 – Алгоритм поиска дисперсных повторов.

итераций улучшения выбираются локальные параметры (или берутся одинаковыми для всех), а после выполняется итерационная процедура улучшения набора подпоследовательностей. На вход поступает последовательность символов S длиной L (алфавит заранее задан). Далее создается Num2 количество начальных наборов подпоследовательностей Q(S, j). Для последующих итераций улучшения выбираются локальные параметры (или берутся одинаковыми для всех), а после выполняется итерационная процедура улучшения набора подпоследовательностей.

Алгоритм улучшения состоит из нескольких частей. Первым по созданному ранее набору создается множественное выравнивание. Это необходимо для того, чтобы учесть возможные вставки и деленции. Это может быть реализовано по средствам различных программ, однако в приведенной работе для этого будет использоваться программа MUSCLE [26]. Далее по построенному множественному выравниванию производится поиск наиболее близких этому набору подпоследовательностей исходной последовательности S. Данная процедура реализована с помощью программы nHMMER [23]. Среди полученных наборов выбирается определенное количество лучших (это количество выбирается на этапе выбора текущих параметров), а после записывается в новый набор подпоследовательностей. На этом завершается этап итерационной процедуры улучшения и начинается следующий. Данная процедура заканчивается по истечению определенного количества шагов Num1.

После того, как алгоритм улучшения закончен, происходит получение конечного для данного начального набора j результата поиска – Q_res(S, j). Данный набор является результатом улучшения начального набора Q(S, j). Поэтому, если начальный случайный выбор подпоследовательности не был удачным, результат улучшения также не даст приемлемого результата. В связи с этим, необходимо множество различных начальных подпоследовательностей, чтобы вероятность удачного поиска увеличилась.

3.2. Разработка требований к системе поиска.

Существует несколько существенных ограничений при разработке данной системы поиска дисперсных повторов.

К первому блоку относится проблема малого количества начальных наборов подпоследовательностей. При малом объеме мутаций, то есть когда две любые цепочки подпоследовательностей из корректного блока найденных повторов имеют малые отличия (или не имеют совсем), объем множества начальных наборов может быть сравнительно не велик. Однако при большом количестве мутаций лишь малая часть удачно выбранных подпоследовательностей может иметь положительный результат. В связи с этим целесообразно брать большое количество начальных наборов. Но подобное расширение ведёт к возникновению проблем с памятью и временем работы программы.

Стоит отметить, что каждая из итераций улучшения (таких как множественное выравнивание, получение нового набора, поиск по набору наиболее схожих в исходной последовательности подпоследовательностей) занимает немалое количество времени. В связи с этим, если для каждого набора производятся улучшения, то при сколь-либо крупном числе начальных наборов Num2 работа программы может занимать сутки. Поэтому необходимо оценивать затраченное время и по возможности сократить его.

Для этого может быть предложено несколько путей решений. Одним из таких является регулировка количество итераций улучшения Num1. При этом необходимо учитывать, что при слишком малом количестве итераций могут быть не найдены все дисперсные повторы, которые ищутся и могут быть найденными с текущим начальным набором подпоследовательностей. В связи с этим необходимо выбрать оптимальное количество Num1.

Для уменьшения времени работы программы также важно уметь находить тупиковые ветки и прерывать их. К примеру, в тех случаях, когда итерационная процедура не находит новые или даже уменьшает общее количество найденные повторов, то этот блок с текущим начальным набором подпоследовательностей следует прервать из-за их неэффективности. Данный способ приведет к уменьшению общего числа итераций и, соответственно, сократит время.

Также для существенного сокращения времени стоит использовать распараллеливание обработки начальных наборов при работе программы. Так как все итерации улучшения независимы (относительно начального набора подпоследовательностей), то возможно производить поиск на нескольких начальных наборах одновременно. Тем не менее, после этого необходимо решить задачу поиска наилучшего решения из всех найденных и предоставить его как результат.

Ещё одной проблемой, которая возникает при увеличении числа начальных наборов подпоследовательностей Num2, является проблема с занимаемой памятью. В идеальном случае стоит хранить каждый начальный набор подпоследовательностей и каждую процедуру их улучшения. Однако при таком выборе количество хранимых файлов будет пропорционально произведению Num1 и Num2, что может быть слишком велико. Так для некоторой степени мутации необходимо производить 100000 выборов начальных наборов (и более), то хранить весь объем памяти является нецелесообразным. В связи с этим, важным требованием к работе программы является оптимизация количества памяти и выходных файлов.

Другим блоком возникающих трудностей является проблема с уменьшением длины найденных подпоследовательностей (повторов) при работе итераций улучшения. Это может происходить при слишком большой степени мутаций, когда локальное обрезание длины подпоследовательностей на некотором этапе может увеличивать степень схожести. В связи с этим необходимо разработать систему регулировки длин подпоследовательностей на конечном этапе разработки. Таким образом, важным требованием является установка и регулировка длин подпоследовательностей в окрестности заданного предположения о подобной длине.

Подводя итог, требованиями к разработке системы поиска дисперсных повторов является удовлетворимое соотношение количества начальных наборов, количества итераций улучшения, времени работы, затрачивания памяти и длины найденных подпоследовательностей.

3.3. Выводы.

1. Был разработан алгоритм системы поиска дисперсных повторов.
2. Были выделены основные составляющие алгоритма, для каждого блока была приведена его смысловая нагрузка.
3. Был выдвинуты требования к разрабатываемой системы поиска дисперсных повторов.
4. Были выделены некоторые аспекты возникающих проблем в разработке, были предложены пути их решения.

Раздел 4. Реализация и тестирование системы поиска дисперсных повторов

В данном разделе будет описана программная часть системы поиска дисперсных повторов, описан выбор языка реализации и используемых систем. Также будут предоставлены результаты тестирования.

4.1. Предложение выбора языка и среды разработки для работы со сторонними программами.

Основные вычислительные затраты приходятся на выполнение таких процедур, как множественное выравнивание и поиск по полученному набору. Оба эти действия производятся по средствам готовых программ. В связи с этим, оболочку программ удобнее всего провести на языке Python, в связи с удобным интерфейсом и возможностью работы с командной строкой.

Множественное выравнивание выполняется по средствам программы MUSCLE. Наиболее удобный формат работы с данным средством является его запуск из командной строки Linux. Также параметры этого запуска задается из командной строки. Таким образом, данная программа вызывается через командную строку по средствам языка Python. Возможность запуска командной строки выполняется за счёт библиотеки «subprocess» Python. Данная программа принимает на входной файл формата «FASTA» и выходной файл формата «FASTA» (куда будет записан результат выравнивания). Для преобразования форматов файлов используется библиотека «Bio.SeqIO» и ручное изменение файлов. Пример запуска программы представлен в Приложении 1.

Поиск по набору подпоследовательностей наиболее похожих на них в исходной последовательности S выполняется по средствам программы nHMMER [23]. Наиболее

удобный формат работы, как и с программой MUSCLE, является его запуск из командной строки Linux. Также параметры этого запуска задается из командной строки. Программа вызывается через командную строку по средствам языка Python. Программа принимает на вход названия несколько файлов с определенными флагами и параметры: файл с исходной последовательностью S в формате «Stockholm» - формат файла, используемый в биоинформатике для представления множественных выравниваний последовательностей; файл с набором подпоследовательностей в формате «Stockholm»; файл для общей информации о результате работы программы (флаг «-o»); файл с итоговыми найденными позициями повторов (флаг «---aliscoresout»); значением E-value (флаг «--incE») - значение по умолчанию равно 0,01, что означает, что в среднем на каждые 100 поисковых запросов с различными последовательностями запросов ожидается примерно 1 ложноположительный результат. Также были использованы такие флаги как « --noali» — исключение раздела "Выравнивание" из основного вывода; «--notextw» — отключение ограничения на длину каждой строки в основном выводе; «--singlemx» — для принятия файла с множеством подпоследовательностей; «--dna» — определение, что последовательность имеет символы ДНК. Пример запуска программы представлен в Приложении 1.

Результатом работы программы nHMMER является два выходных файла: файл с общей информацией о работе программы и файл с позициями найденных повторов. Первый файл включает в себя все данные о работе программы, таких как адреса входных файлов, порог E-value, информацию о позициях найденных подпоследовательностей, их E-value, Z-score и их общее описание.

Те подпоследовательности, E-value которых удовлетворяет заданному значению, попадают во второй файл. В нем содержаться номера их позиций (начала и конца в исходной последовательности S). Далее, по этому файлу и заранее определенному числу подпоследовательностей (см. далее), которые берутся для следующих итераций, составляются файлы представления результатов в формате «txt» и файлы для следующей итерации (множественного выравнивания) в формате «FASTA». Программа подобного создания файлов представлена в Приложении 1.

Число выбранных для следующих итераций подпоследовательностей не может быть слишком большим, в связи с тем, что при большом количестве выбранных подпоследовательностей операция построения множественного выравнивания может занимать слишком большой промежуток времени. Однако при слишком маленьком числе выбранных подпоследовательностей для следующего шага операция поиска может быть не удачной. Это связано с тем, что при большой степени мутации отдельные пары

слишком сильно отличаются, и чтобы найди схожие подпоследовательности, необходимо рассматривать степень близости со многими подпоследовательностями одновременно.

Таким образом, число выбираемых (из представленных на определенном шаге работы программы nHMMER) подпоследовательностей должно быть не слишком мало, и не слишком велико. В среднем, наиболее оптимальное подобное число – 40. Для малой степени мутации можно брать меньше (что сокращает время работы программы), для большой – больше (что создает возможность поиска в условиях малой схожести отдельно взятых подпоследовательностей).

Подводя итог, можно сделать вывод, что для работы со сторонними программами (работа которых занимает большую часть общего времени работы программы), наиболее удобно использовать язык Python. Работа с программами осуществляется через командную строку Linux (вызываемую из Python). Для корректной работы этих программ применяется преобразование типов файлов, отсеивание лишней информации и другое (см. Приложение 1).

4.2. Выбор алгоритмической реализации основной программы.

В связи с тем, что количество начальных наборов может быть велико, а итерационные процедуры улучшения независимы для разных начальных наборов, то в разрабатываемой системе поиска дисперсных повторов используется параллельное вычисление. В системе поиска на языке Python параллельность выполняется по средствам библиотеки «multiprocessing». Общее количество начальных наборов разбивается на процессы, число которых равно числу ядер устройства.

Исходя из проблемы, представленной в пункте 3.2., необходимо было решить проблему количества файлов и их независимости. В начальных версиях алгоритма для каждого начального набора создавалась собственная папка, в которой записывались все итерации улучшения, текущих позиций найденных подпоследовательностей и другое. Однако, при большом количестве начальных наборов (тысячи и более) такое количество папок становится излишним. К тому же, в связи с тем, что исходная последовательность S у всех одна (и находится в определенном файле), то к ней обращались из разных потоков одновременно, что создавало задержку времени (так как происходила блокировка всех потоков, кроме текущего). Для решения этой проблемы можно создавать в каждой папке, соответствующей начальному набору, собственную копию файла с исходной последовательностью, однако это ведет к излишнему дублированию информации.

В связи с этим, был предложен иной вариант работы программы. Так как в одном потоке происходит множество последовательных итераций улучшения (для всех начальных набора, принадлежащего данному потоку, создается очередь), то в рамках

одного потока всегда идёт последовательное выполнение действий (см. Приложение 1). Таким образом, было принято решение создать независимые блоки памяти (с выделенными папками) для каждого потока. В связи с этим, общее количество хранимых файлов будет пропорционально числу потоков, то есть, в предложенном исполнении, числу ядер. В связи с этим возникает подзадача поиска наилучшего результата в одном потоке.

Подобная задача выполняется за счёт выбора того результата начального набора (в одном потоке), у которого максимально число найденных последовательностей. Таким образом, для одного участка памяти (для каждого потока своя) будет храниться исходная последовательность, файл с лучшим результатом (набор подпоследовательностей), файл с лучшими позициями подпоследовательностей (соотнесено с лучшим набором подпоследовательностей), постоянно перезаписываемые файлы для итераций последующих начальных наборов, и файлы, связанные с корректировкой длин подпоследовательностей (см. далее). Так исходная последовательность существует для каждого потока своя (а внутри потока итерации для различных наборов выполняется последовательно), то не будет проблемы с постоянным обращением к одному файлу. К тому же, так как число потоков сравнительно не велико, то не происходит слишком большого хранения одной и той же информации огромное число раз. Данная реализация представлена в Приложении 1.

Стоит отметить, что благодаря получаемой информации о числе найденных подпоследовательностей на каждом шаге итераций улучшений, возможно отсекать неудачные итерации текущих начальных наборов. Это происходит при отсутствии увеличения количества найденных подпоследовательностей на шагах итерации. Подобное решение существенно (при большой степени мутаций - на порядки) уменьшает время работы программы.

Разбиение на потоки происходит автоматически по средствам работы с методом «map» библиотеки «multiprocessing.Pool». Работа и обращение к процессам, а также создание, перемещение и копирование папок и файлов реализуется благодаря библиотеке «os» Python. Способ работы указан в Приложении 1.

Таким образом, для уменьшения времени работы, сокращения объёма занимаемой памяти и лучшей структуризации решений был предложен и исполнен вариант реализации распараллеливания работы программы относительно множества начальных наборов подпоследовательностей. Код работы программы предоставлен в Приложении 1.

4.3. Разработка отдельных ключевых узлов программы.

4.3.1. Получение начальных наборов

Есть несколько различных подзадач, об алгоритмической реализации которых стоит поговорить отдельно.

Первая подзадача – это составление начальных наборов. Для этого необходимо выбрать случайно взятые подпоследовательности исходной последовательности S. В связи с этим, необходимо выбрать позиции начал этих подпоследовательностей. Эти подпоследовательности должны обладать следующими свойствами: иметь заданную (вводимую) длину, не пересекаться, не выходить за границы исходной последовательности, быть случайно расположенными. Для этого реализуется несколько функций выбора начальных позиций и получение самих подпоследовательностей (см. Приложение 1).

Однако просто взятыми случайными подпоследовательностями можно не ограничиваться. В связи с особенностями программы nHMMER было произведено исследование, связанное с возможностью дублировать некоторых начальных подпоследовательностей несколько раз. При этом для разной степени мутации исходной последовательности S вывод получается разный.

При малой степени мутации ($x < 1$) выбор большого количества начальных последовательностей без повторений оправдан. Это связанно с тем, что при таком объеме мутаций степень схожести двух реальных повторов сравнительно мала. Поэтому, даже если в начальный набор попала лишь часть одного из реальных повторов, итерационный алгоритм улучшения сможет зацепиться и найти все остальные повторы.

Однако, при большой степени мутации ($x > 1$) предложенный выше подход не дает такой точности. Так как связь отдельно взятых подпоследовательностей мала, то nHMMER, считающий степень близости со всеми последовательностями начального набора одновременно, не сможет найти последующие реальные повторы.

При запуске различных вариантов количества различных начальных подпоследовательностей и количества их дублирования при большой степени мутации были сделаны следующие выводы. Наибольшее количество схожих с начальным набором подпоследовательностей программа nHMMER находит при количестве различных подпоследовательностей равным трем, а количестве повторений – шести. При меньшем или большем количестве повторов наблюдается меньшее количество найденных повторов (при количестве находящихся в окрестности шестерки данные отличия слабо выражены).

На следующих графиках представлены распределение количества найденных итоговых повторов в зависимости от способа выбора начальных наборов. Для первого случая – это 20 непересекающихся различных подпоследовательностей, для второго – это 3 различные подпоследовательности, взятых по 6 раз. Всего было для каждого по 400

начальных наборов, истинных повторов - 400. По оси абсцисс - количество найденных повторов, по оси ординат – количество начальных наборов.

На первом рисунке представлен график при степени мутации x = 0.5.

На первом рисунке представлен график при степени мутации x = 1.05.

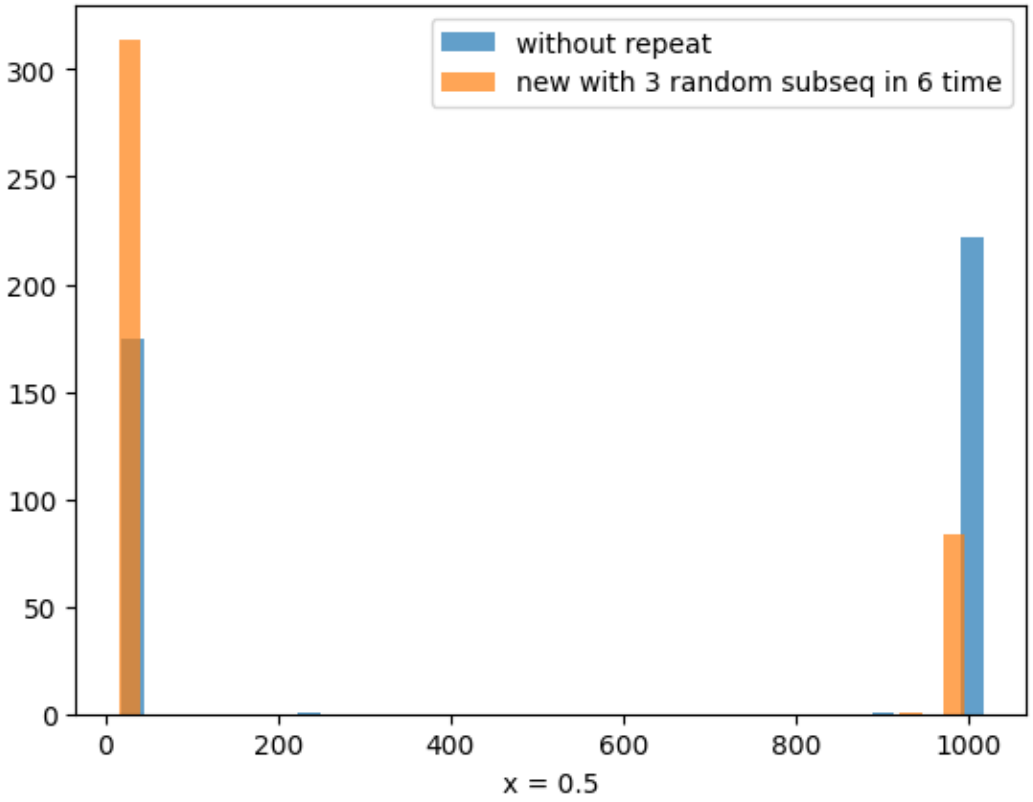

Рис 4.1 – Распределение найденных повторов при x = 0.5

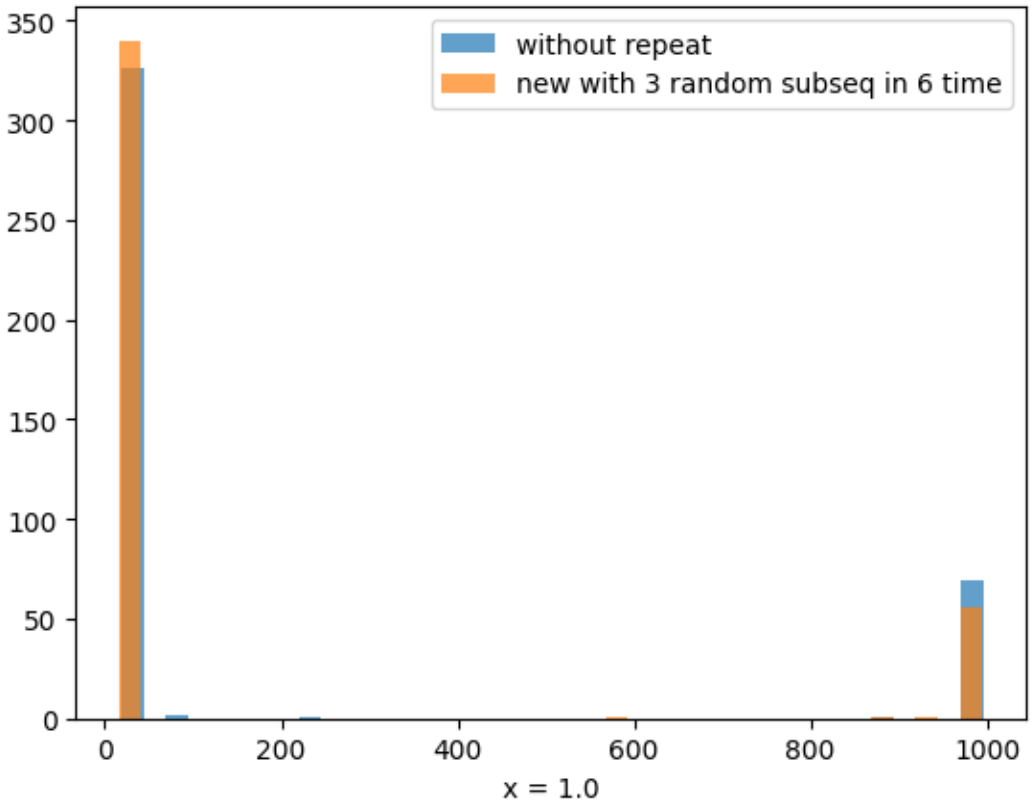

Рис 4.2 – Распределение найденных повторов при x = 1.0

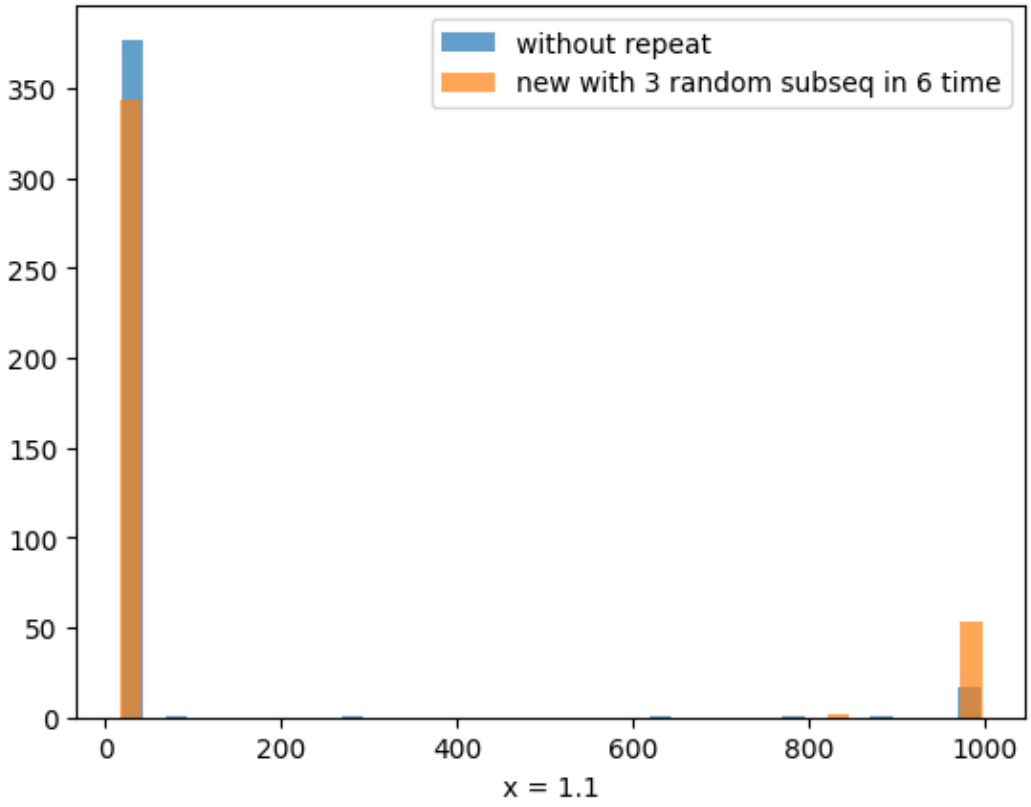

Рис 4.3 – Распределение найденных повторов при x = 1.1

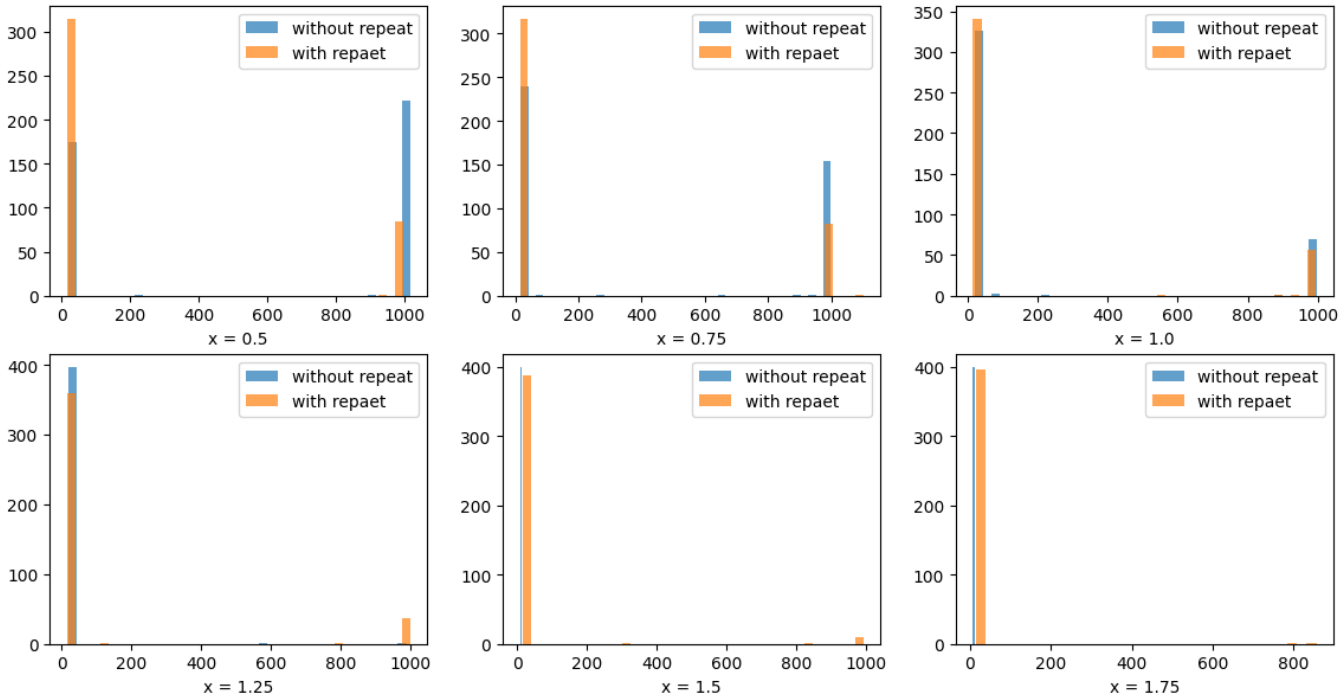

Рис 4.4 – Общее распределение найденных повторов при различных x

По предоставленным графикам можно заметить, что при x < 1.0 большее число найденных повторов имеет первый вариант инициации начального набора.

При x >= 1.1 может быть замечена тенденция к преобладанию другого варианта решения.

На предоставленных графиках отчетливо видно, что при увеличении степени мутации, количество успешно найденных (полностью, то есть 1000 подпоследовательностей) стремительно уменьшается. Для поиска повторов при x >= 1.8 необходимо несколько тысяч начальных наборов.

Таким образом, было предоставлено несколько вариантов решений создания начальных наборов, работающих по-разному в зависимости от степени мутаций. Реализации предоставлены в Приложении 1.

4.3.2. Корректировка длин найденных подпоследовательностей.

При работе с последовательностями, степень мутаций которых велика, возникает следующая проблема. При работе программы nHMMER найденные последовательности обрезают. То есть при последних итерациях улучшения средняя длина подпоследовательностей может достигать 100 при ожидаемых 400. Это связано с тем, что данная программа ищет локальное выравнивание и при небольшом обрезании длины показатель точности может быть локально больше. К тому же, исходя из этого программа может находить меньше подпоследовательностей, чем должна (к примеру, 600 вместо 1000). В связи с этим были разработаны способы решения данной проблемы.

Первым вариантом стало добавление слева и справа от найденных позиций подпоследовательностей какого-то определенного числа. После этого, проводится ещё одна специальная дополнительная процедура улучшения. Однако, в связи с тем, что на последнем шаге итераций улучшения длины в получаемом наборе могут сильно разниться, некоторые подпоследовательности могут быть увеличены слишком сильно, другие наоборот слабо. К тому же, так как сдвиг слева и справа мы задаем вручную, то мы не может понять, сколько и с какой стороны будет оптимально добавлять. В связи с этим были предложены ещё две другие (схожие) процедуры увеличения длин подпоследовательностей.

Во втором варианте реализации количество добавленных слева и справа также задается вручную, однако частично решается проблема различия длин. В этом способе каждое добавление корректируется с учетом начальных и конечных пропусков в полученном множественном выравнивании. Это дает большую кучность длин, однако не решает общую проблему количества добавленных слева и справа.

Третьим вариантом решения увеличения длин стало комбинация второго решения с корреляцией на ожидаемое конечное значение длин повторов. Для начала определяется общее количество символов, которое нужно добавить (ожидаемая средняя длина минус

максимальная длина подпоследовательности в наборе). Далее происходит сдвиг окна слева направо по количеству взятых дополнительно символов слева и справа. Таким образом, запускается алгоритм реализации второго способа несколько раз в зависимости от взятых длин слева и справа, а после берется лучший вариант. Данный способ хорош тем, что в нем количество добавлений слева и справа определяется программой автоматически. Однако третий вариант реализации работает в несколько раз дольше первых двух.

Таким образом, каждых из трех реализаций имеет место на жизнь. В зависимости от усложнения увеличивается время работы программы, то повышается кучность и алгоритмический смысл. График сравнения распределения длин найденных подпоследовательностей приведен ниже. До процедуры увеличения длин было найдено всего 602 подпоследовательности, длины которых варьируется в районе 100 (синий цвет). Первый способ представлен оранжевым цветом, слева и справа прибавляется по 200 символов. Зеленый – второй способ, завязанный на множественном выравнивании. Красный – последний вариант, длины определяется автоматически.

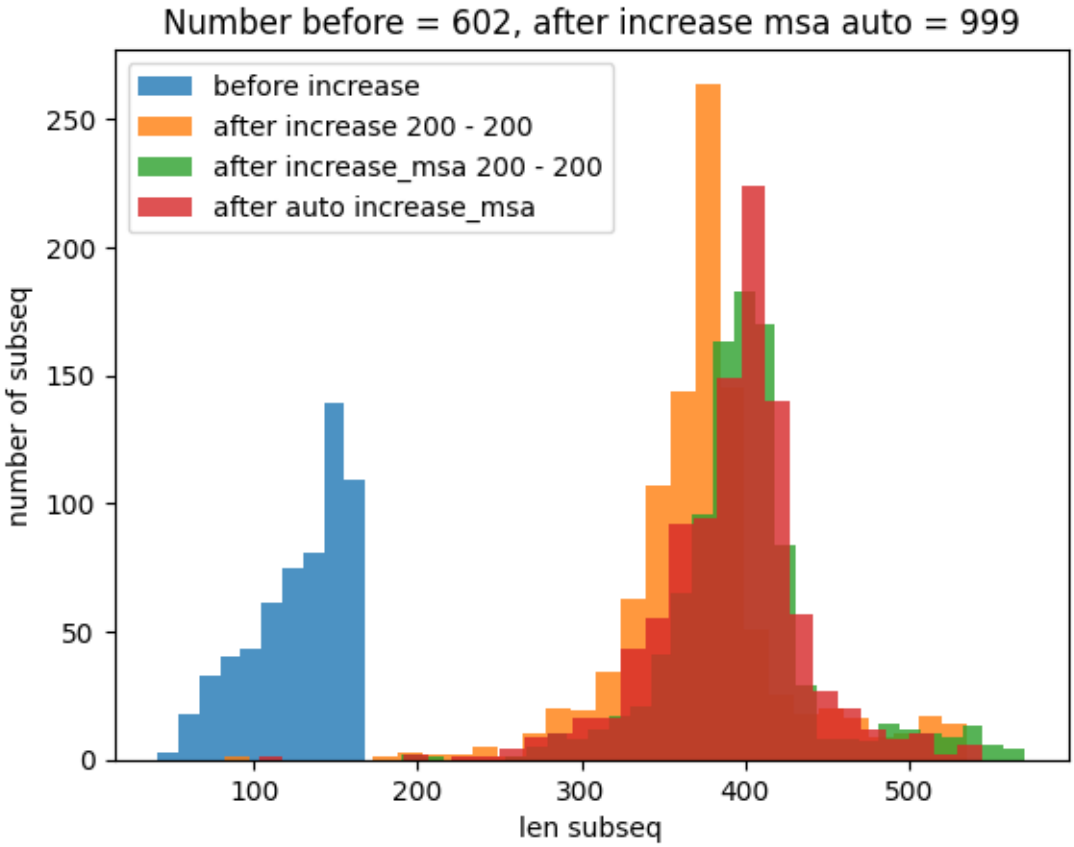

Рис 4.5 – Общее распределение длин найденных повторов при различных способах увеличения.

Таким образом, каждый из предложенных реализаций существенно увеличивает количество найденных повторов и приближает среднюю длину к искомой. В связи с тем,

что третий вариант находит оптимальное решение, то он был реализован в основном алгоритме (можно изменить, используя флаги – см.ниже). Стоит отметить, что данную процедуру необходимо проводить для большой степени мутаций, для малых её можно упустить, сокращая время работы программы. Все предложенные реализации находятся в Приложении 1.

4.4. Разработка внешних входов и выходов программы.

Разработанная система поиска дисперсных повторов представляет собой программу на языке Python. При запуске программы из командной строки существует множество различных вариантов флагов (вариантов работы программы, который реализует алгоритм).

Работа с аргументами командной строки производится за счёт библиотеки «argparse» Python (см. Приложение 1).

Программа может принимать следующие аргументы:

Флаг	Описание	Обязатель-ность	Тип данных	Значение по умолчанию
-f --filename	Имя файла с исходной последовательностью	True	str	-
--ml --mean_len	Средняя ожидаемая длина повтора	False	int	400
--ns --number_subseq	Количество начальных наборов	False	int	100
-N --number_diff_subseq	Количество различных подпоследовательностей в одном начальном наборе	False	int	3
--rp --repeat_subseq	Количество повторов подпоследовательностей в одном начальном наборе	False	int	6
-l --len_subseq	Длина подпоследовательностей в начальном наборе	False	int	410
-E --E_value	E-value при котором последовательность принимается	False	float	0.01
--ni	Количество итераций	False	int	10

--number_iteration	итерационной процедуры улучшения			
--N_search	Количество подпоследовательностей, которые берутся на шаге итерационного улучшения	False	int	40
--increase_len	Нужно ли проводить увеличение длины подпоследовательностей до ожидаемого значения	False	bool	False
--nf --name_folder	Название папки, в которую будут записаны процедуры работы программы	False	str	«SDR» - Search Dispersed Repeats
-o --name_file_out	Название файла для записи результатов (если нет – запись в стандартный поток вывода)	False	str	«-»

Таблица 4.1 – Описание работы флагов

Единственный необходимый атрибут – это название файла, в котором находится исходная последовательность S (алфавит – символы ДНК).

Остальные параметры не являются обязательными, однако по ним можно варьировать способ решения. Это необходимо для подбора наиболее удачных параметров, позволяющих оптимизировать время работы программы и полученного результата.

Выходом работы программы в общем случае является данные о наилучшем найденном запуске, данные о его протекании, времени работы программы и расположении в файловой системе. Данная информация может быть загружена в файл при наличии определённого файла.

Таким образом, разработанная система имеет различные варианты запуска программы. Способ задания данных параметров показан в Приложении 1.

4.5. Разработка тестирования.

Вся работа системы поисков дисперсных повторов на данном этапе реализации была проведена на созданных последовательностях, на которых степень мутаций

определялась при генерации последовательности.

Реализация генераций последовательности S состоит из нескольких этапов. Последовательность генерируется по алфавиту, по заданной общей длине, количества повторов в ней, их длине, и степени мутации x. В связи с тем, что все элементы последовательности S, не входящие в повторы, случайны, то степень мутации на них не играет роли (случайное изменение случайных последовательностей). Таким образом, степень мутации x определяется разницей между двумя любыми повторами.

В связи с этим, была сгенерирована изначальная подпоследовательность – родитель будущих повторов. После для добавления повтора в последовательность с ней происходят мутации.

Первым идёт процедура вставки и деленции. На случайно выбранные позиции будущего повтора добавляют (или удаляют) один символ алфавита. Количество выбранных позиций – 1 на 100 символов. Таким образом, происходит несколько вставок или деленций (их соотношение, как и позиции, случайно).

Далее для будущего повтора происходит процедура замен (степень которых определяется x – среднем число замен на нуклеотид). После происходит последовательная замена символов на случайных позициях на случайных символ алфавита. Число замен – произведение x на длину повтора, деленное пополам. Таким образом, между любыми двумя повторами средняя степень различия на один нуклеотид (символ) – x. Деление пополам производилось в связи тем, что мутация происходила в обоих последовательностях независимо, поэтому общая степень различия двух повторов – сумма различий на каждом.

Таким образом, применив алгоритм мутации для каждого повтора мы получаем набор будущих повторов, которые должна найти разрабатываемая система поиска дисперсных повторов. Эти повторы распределяются на случайные позиции будущей последовательности S, а пространство между ними заполняется случайными символами алфавита. Реализация процесса генераций последовательностей проиллюстрирована в Приложении 1.

При запуске системы поиска дисперсных повторов на специальных последовательностях, которые были сгенерированы случайно (то есть в которых отсутствуют дисперсные повторы), на шаге итерационной процедуры улучшения не происходит увеличения количества найденных подпоследовательностей. Таким образом, система не находит подпоследовательностей в случае отсутствия повторов.

При работе системы на последовательностях, позиции повторов которых записываются, заметны следующие особенности. Каждая из найденных

подпоследовательностей имеет значительные пересечения с реальными повторами. Однако при большой степени мутации длины найденных повторов могут разниться, в связи с чем, полное пересечение найденных и искомых подпоследовательностей достигается не для всех последовательностей. Тем не менее, общее расположение повторов по исходной последовательности находятся корректно.

На последующем графике расположены кривые, показывающие эффективность работы системы поиска дисперсных повторов. По оси абсцисс – степень мутации, по оси ординат – количество найденных повторов.

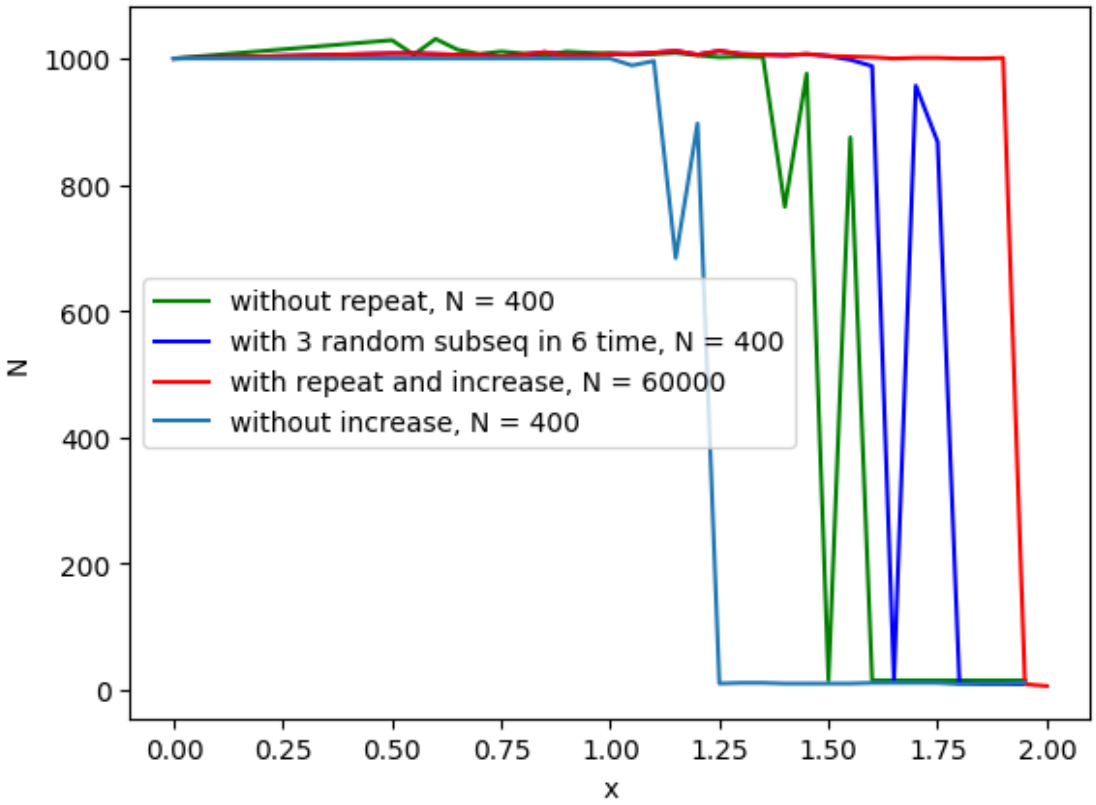

Рис 4.6 – Результат работы алгоритма.

Для первых графиков (все кроме красных) количество начальных наборов бралось по 400 (для малой и средней степени мутаций это достаточно). Первым был реализован график без процедуры увеличения (бледно-синий). Видно, что этот способ реализации работает эффективно для степени мутации меньше 1.1. Последующие по эффективности графики наглядно показывают разницу в использовании повторов в начальном наборе или нет (см. 4.3).

При использовании повторов область эффективности выросла до 1.75. Однако далее для эффективной работы программы требовалось гораздо большее число вариантов начальных повторов, чем 400 (см 4.3). Благодаря реализации остановки неудачных ветвей итераций улучшения, время на неудачные повторы удалось сократить на порядки. Итоговая версия программы при 60000 начальных наборов эффективно находит повторы

до степени мутации 1.9 включительно, что существенно расширило по сравнению с существующими аналогами (см. 1.1).

4.6. Выводы.

1. Были описаны способы взаимодействия со сторонними программами, такими как nHMMER и MUSCLE.
2. Был описан алгоритмический путь работы программы, был описан способ реализации параллельного поиска. Были описаны решения проблем со временем и памятью.
3. Был описан способ проработки отдельных ключевых узлов, таких как создание начальных наборов и решения проблемы сокращения длин найденных подпоследовательностей.
4. Был описан способ работы с системой поиска дисперсных повторов.
5. Был описан способ генерации последовательностей и конечная работе на них.

Заключение

В рамках учебной исследовательской работы была произведена следующая работа:

1. Были проанализированы существующие методы поиска дисперсных повторов, были описаны их ключевые составляющие.
2. Был произведен анализ уже существующих систем поиска дисперсных повторов, были выявлены особенности подобных систем и используемые в них различные подходы.
3. Был предложен собственный метод подхода к решению задачи поиска дисперсных повторов. Были описаны его составляющие, ключевые аспекты и модификации.
4. Была спроектирована и описана система поиска дисперсных повторов. Созданная система имеет возможность поиска при различных параметрах, заданных при запуске программы.
5. Был произведен анализ созданной системы поисков дисперсных повторов.

Подводя итог проделанной работе, можно сделать следующие выводы.

На данном этапе работы конечным результатом является программа, работающая при средней степени мутации 1.9 на один нуклеотид. Согласно [6] такие известные программы как RED, RECON и другие способы эффективно находить повторы при степени мутации 1.0 и меньше (на тех же последовательностях).

Программа, предложенная [6] показывает свою эффективность при степени мутации 1.5 и меньше. Таким образом, разработанная система поиска дисперсных повторов SDR показала существенный прирост области применения алгоритма по сравнению с существующими аналогами.

Стоит отметить, что данная работа имеет множество путей дальнейшего развития. Система поиска дисперсных повторов работает, используя такие сторонние сервисы как nHMMER и MUSCLE. Несмотря на то, что эти программы имеют мировое признание, существуют новые методы, описанные в [6]. Разработанная система ограничена показателями данных программ. В связи с тем, при собственной реализации этих моментов возможно дальнейшее улучшение программы.

Также в связи с спецификой работы приведенных выше программ система поиска дисперсных повторов SDR ограничена алфавитом, с которым эти программы работают. Несмотря на это, в разрабатываемой системе отсутствует использование сторонних, априорных данных о геномах. Таким образом, при собственной реализации приведенных выше программ возможно отказаться от ограничений алфавита и работать с последовательностями произвольного алфавита.

В связи с тем, что созданная система поиска дисперсных повторов передвинула порог допустимых мутаций, чрезвычайно важно проанализировать с её помощью раннее уже отсеквенированные геномы. Вполне вероятно обнаружение ранее не найденных дисперсных повторов, степень мутации которых превышает области эффективной работы ранее существующих программ.

Таким образом, разработанная система поиска дисперсных повторов имеет пространство для улучшения. Однако даже на текущем уровне работы программы она способна открыть некоторые области генома, которые ранее не были найдены. Подобное исследование способно дать существенный толчок к развитию биоинформатики и науки в целом.

Список литературы

1. Durbin, R., S. Eddy, A. Krogh and G. Mitchison (1998): Biological sequence analysis: probabilistic models of proteins and nucleic acids, Cambridge, UK: Cambridge University Press.
2. Storer, J.M.; Hubley, R.; Rosen, J.; Smit, A.F.A. Methodologies for the De novo Discovery of Transposable Element Families. Genes 2022, 13, 709. https://doi.org/10.3390/GENES13040709
3. Tiwari S, Ramachandran S, Bhattacharya A, Bhattacharya S, Ramaswamy R. Prediction of probable genes by Fourier analysis of genomic sequences. ComputApplBiosci. 1997 Jun;13(3):263-70. https://doi.org/10.1093/bioinformatics/13.3.263
4. Meng T, Soliman AT, Shyu ML, Yang Y, Chen SC, Iyengar SS, Yordy JS, Iyengar P. Wavelet analysis in current cancer genome research: a survey. IEEE/ACM Trans ComputBiolBioinform. 2013 Nov-Dec;10(6):1442-59. doi: 10.1109/TCBB.2013.134. PMID: 24407303.
5. Lobzin V V, Chechetkin V R "Order and correlations in genomic DNA sequences. The spectral approach" Phys. Usp. 2000, 43, 55–78
6. Korotkov, E., Suvorova, Y., Kostenko, D., Korotkova, M. Search for dispersed repeats in bacterial genomes using an iterative procedure. International Journal of Molecular Sciences. 2023, 24(13), 10964. https://doi.org/10.3390/ijms241310964
7. Eddy SR. Hidden Markov models. CurrOpinStruct Biol. 1996 Jun;6(3):361-5. https://doi.org/10.1016/S0959-440X(96)80056-X
8. Joly-Lopez Z, Bureau TE. Exaptation of transposable element coding sequences. CurrOpin Genet Dev. 2018 Apr;49:34-42 https://doi.org/10.1016/j.gde.2018.02.011
9. Lander ES, Linton LM, ,Szustakowki J; International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. Nature. 2001 Feb 15;409(6822):860-921.doi: 10.1038/35057062.
10. Smit, A.F.A. The origin of interspersed repeats in the human genome. Curr. Opin. Genet. Dev. 1996, 6, 743–748. https://doi.org/10.1016/S0959-437X(96)80030-X
11. Tempel, S. Using and understanding repeatMasker. Methods Mol. Biol. 2012, 859, 29–51. https://doi.org/10.1007/978-1-61779-603- 6_2.

12. Jurka, J.; Klonowski, P.; Dagman, V.; Pelton, P. CENSOR—A program for identification and elimination of repetitive elements from DNA sequences. Comput. Chem. 1996, 20, 119–121.
13. Bedell, J.A.; Korf, I.; Gish, W. MaskerAid : A performance enhancement to RepeatMasker. Bioinformatics 2000, 16, 1040–1041. https://doi.org/10.1093/bioinformatics/16.11.1040.
14. Bao, W.; Kojima, K.K.; Kohany, O. Repbase Update, a database of repetitive elements in eukaryotic genomes. Mob. DNA 2015, 6, 11. https://doi.org/10.1186/s13100-015-0041-9
15. Girgis, H.Z. Red: An intelligent, rapid, accurate tool for detecting repeats de-novo on the genomic scale. BMC Bioinform. 2015, 16, 227. 11.
16. Bao, Z.; Eddy, S.R. Automated de novo identification of repeat sequence families in sequenced genomes. Genome Res. 2002, 12, 1269–1276. https://doi.org/10.1101/gr.88502.
17. Edgar, R.C.; Myers, E.W. PILER: Identification and classification of genomic repeats. Bioinformatics 2005, 21 (Suppl. S1), i152– i158. https://doi.org/10.1093/BIOINFORMATICS/BTI1003.
18. Price, A.L.; Jones, N.C.; Pevzner, P.A. De novo identification of repeat families in large genomes. Bioinformatics 2005, 21 (Suppl. S1), i351–i358. https://doi.org/10.1093/bioinformatics/bti1018.
19. Volfovsky, N.; Haas, B.J.; Salzberg, S.L. A clustering method for repeat analysis in DNA sequences. Genome Biol. 2001, 2, 0027.1. https://doi.org/10.1186/GB-2001-2-8-RESEARCH0027.
20. Altschul, S.F.; Madden, T.L.; Schäffer, A.A.; Zhang, J.; Zhang, Z.; Miller, W.; Lipman, D.J. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. Nucleic Acids Res. 1997, 25, 3389–3402. https://doi.org/10.1093/nar/25.17.3389.
21. Mount, D.W. Using a FASTA Sequence Database Similarity Search. CSH Protoc. 2007, 2007, pdb.top16.
22. Tamura, K.; Peterson, D.; Peterson, N.; Stecher, G.; Nei, M.; Kumar, S. MEGA5: Molecular evolutionary genetics analysis using maximum likelihood, evolutionary distance, and maximum parsimony methods. Mol. Biol. Evol. 2011, 28, 2731–2739. https://doi.org/10.1093/MOLBEV/MSR121.

23. Wheeler, T.J.; Eddy, S.R. Nhmmer: DNA homology search with profile HMMs. Bioinformatics 2013, 29, 2487–2489. https://doi.org/10.1093/bioinformatics/btt403.
24. Bastien, O.; Aude, J.-C.; Roy, S.; Maréchal, E. Fundamentals of massive automatic pairwise alignments of protein sequences: Theoretical significance of Z-value statistics. Bioinformatics 2004, 20, 534–537.
25. Ignatiadis, N., Wang, R., & Ramdas, A. E-values as unnormalized weights in multiple testing. Biometrika 2024, 111(2), 417-439.
26. Edgar, R.C. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res. 2004, 32, 1792–1797.

Приложение 1.

Использование библиотек Python для запуска команды командной строки Linux и работе с фалами.

```
import subprocess
from Bio import SeqIO
```

Запуск программы Muscle.

```
def muscle_msa(name_in, name_out = 'msa.afa'):
    process = subprocess.run(['muscle', '-in', name_in, '-out', name_out],
capture_output=True, text = True)
    return process
```

Из формата txt в FASTA

```
def txt_into_fasta(name_txt, name_fasta):
    f = open(name_txt, 'r')
    string = '>i1 seq;\n' + f.readline()
    f.close()

    f1 = open(name_fasta, 'w')
    f1.write(string)
    f1.close()
```

Из формата FASTA в Stockholm

```
def fasta_into_stockholm(name_in = 'example.fa', name_out = 'example.sto'):
    records = SeqIO.parse(name_in, 'fasta')
    count = SeqIO.write(records, name_out, 'stockholm')
```

Запуск поиска подпоследовательностей исходной строки, наиболее схожих с набором подпоследовательностей.

Принимает*: name_set_msa* — имя файла с подпоследовательностями, *name_seq* — имя файла с исходной последовательностью, *name_out_inf* — имя файла для информации о работе программы, *name_out_posision* — имя файла для информации о найденных позициях, *E*_val — значение E-value, по умолчанию равно 0,01, что означает, что в

среднем на каждые 100 поисковых запросов с различными последовательностями запросов ожидается примерно 1 ложноположительный результат.

Возвращает: *name_out_posision* — имя файла, в который записана информация о найденных позициях

```
def nhmmer_on_stock_msa(name_set_msa = 'msa_sto.sto', name_seq = 'seq.fa',
name_out_inf = 'result_hmm_info.fa', name_out_posision = 'posision.fa', E_val =
0.001):
    command = 'nhmmer -o ' + name_out_inf + ' --aliscoresout ' + name_out_posision +
' --noali --notextw --singlemx --dna --incE ' + str(E_val) + ' ' + name_set_msa + ' '
+ name_seq
    process = subprocess.run(command, shell = True, capture_output = True, text =
True)
    return name_out_posision
```

Получение набора подпоследовательностей из выходного файла работы nHMMER.

Принимает: *name_seq* — имя файла с исходной последовательностью, *name_out_posision* — имя файла для информации о найденных позициях, *N* — количество подпоследовательностей, которые пойдут в новый набор, *left_step* — величина добавленного смещения слева, *right_step* — величина добавленного смещения справа, *name_position_subseq step* — названия файла для записи полученного набора

```
def creating_new_set_from_out_nhmmer(name_seq, name_out_posision, N = 20, left_step =
0, right_step = 0, name_position_subseq = 'position_subseq.fa'):
    #получение исходной последовательности из предоставленного файла
    s = string_from_file(name_seq)

    #чтение файла с найденными позициями подпоследовательностей
    f = open(name_out_posision, 'r')
    all_string = f.readlines()
    f.close()
    len_set = len(all_string)

    new_set = []

    #создание файла с итоговыми подпоследовательностями
    f = open(name_position_subseq, 'w')
    f.write('')
    f.close()
    for i in range(len_set):
        #получение номера начального и конечного символа подпоследовательности
```

```
        string = all_string[i]
        index_sep = string.rfind(':') - 1
        prev_space = string.rfind(' ', 0, index_sep - 1)
        try:
            end = int(string[prev_space + 1:index_sep])
        except:
            print("string:", string)
            print("index_sep:", index_sep)
            print("prev_space:", prev_space)
            print("name_file:",name_out_posision)
            print("i:", i)
            x1 = 1 / 0
        prev_space2 = string.rfind(' ', 0, prev_space - 1)
        begin = int(string[prev_space2 + 1:prev_space])

        #изменение номера начального и конечного символа с учётом входящего смещения
        if i > (N - 1):
            break
        if (begin - left_step) <= 0:
            begin = 0
        else:
            begin = begin - left_step

        if (end + right_step) >= (len(s) - 1):
            end = len(s) - 1
        else:
            end = end + right_step

        #добавление найденной подпоследовательности в получаемый набор new_set
        if end - begin > 0:
            f = open(name_position_subseq, 'a')
            f.write('>' + str(begin) + ' ' + str(end) + ':\n' + str(s[begin:end]) +
'\n')
            f.close()
            new_set.append(s[begin:end])
    return new_set
```

Получение файла с набором подпоследовательностей в формате FASTA из выходного файла работы nHMMER.

Принимает: *name_seq* — имя файла с исходной последовательностью, *name_out_posision* — имя файла для информации о найденных позициях В FASTA, *N* — количество подпоследовательностей, которые пойдут в новый набор, *left_step* — величина добавленного смещения слева, *right_step* — величина добавленного смещения справа.

```
def creating_fasta_new_nhmmer_set(name_seq, name_out_posision, N = 20, name_fasta_set
= 'example.fa', left_step = 0, right_step = 0):
    #получение набора подпоследовательностей
```

```
    set_of_subseq = creating_new_set_from_out_nhmmer(name_seq, name_out_posision, N,
left_step = left_step, right_step = right_step)

    #запись в файл набора подпоследовательностей
    string_set_of_subset = ''
    for i in range(len(set_of_subseq)):
        string_set_of_subset += '>i' + str(i) + ' iteration;\n' +
str(set_of_subseq[i]) + '\n'
    file = open(name_fasta_set,'w')
    file.write(string_set_of_subset)
    file.close()
```

Получение результирующего файла с набором подпоследовательностей в формате txt из выходного файла работы nHMMER с заданным диапазоном длин и возвращающих их количество.

Принимает: *name_seq* — имя файла с исходной последовательностью, *name_out_posision* — имя файла для информации о найденных позициях В FASTA, *N* — максимальное количество подпоследовательностей, которые пойдут в новый набор, *left_step* — величина добавленного смещения слева, *right_step* — величина добавленного смещения справа, *min_len_subseq* — минимальная удовлетворяющая величина, *max_len_subseq* — максимальная удовлетворяющая величина.

Возвращает: *number_subseq* — число удовлетворяющих подпоследовательностей.

```
def creating_txt_res_nhmmer_set(name_seq, name_out_posision, min_len_subseq = 300,
max_len_subseq = 500, N = 10 ** 6, name_res_txt = 'example.fa', left_step = 0,
right_step = 0):
    #получение набора подпоследовательностей
    set_of_subseq = creating_new_set_from_out_nhmmer(name_seq, name_out_posision, N,
left_step = left_step, right_step = right_step)

    #запись в файл набора подпоследовательностей
    string_set_of_subset = ''
    number_subseq = 0
    for i in range(len(set_of_subseq)):
        if (len(set_of_subseq[i]) >= min_len_subseq) and (len(set_of_subseq[i]) <=
max_len_subseq):
            string_set_of_subset +=  str(i) + ': ' + str(len(str(set_of_subseq[i])))
+ '\n' + str(set_of_subseq[i]) + '\n'
            number_subseq += 1
    file = open(name_res_txt,'w')
    file.write(string_set_of_subset)
    file.close()
    return number_subseq
```

Реализация работы с аргументами командной строки.

Параметр dest определяет название переменной, куда будет положено значение, required определяет необходимость наличия данного флага, help – информация о параметрах, type – тип переменной, default – значение по умолчанию, при отсутствии флага.

```
parser = argparse.ArgumentParser(description="Ping script")

    parser.add_argument('-f', '--filename', dest='s_name', required=True, help='Name
of file with sequence')
    parser.add_argument('--ml', '--mean_len', dest='mean_len', default=400, type=int,
required=False, help='Mean len of repeat sequence')
    parser.add_argument('--ns', '--number_subseq', dest='number_subseq', default=100,
type=int, required=False, help='Number of different search')
    parser.add_argument('-N', '--number_diff_subseq', dest='N', default=3, type=int,
required=False, help='Number of different initial subsequences')
    parser.add_argument('--rp', '--repeat_subseq', dest='repeat_subseq', default=6,
type=int, required=False, help='Number of repeat initial subsequences')
    parser.add_argument('-l', '--len_subseq', dest='len_subseq', default=410,
type=int, required=False,  help='Len initial subsequence')
    parser.add_argument('-E', '--E_value', dest='E', default=0.01, type=float,
required=False,  help='E-value')
    parser.add_argument('--ni', '--number_iteration', dest='number_iteration',
default=10, type=int, required=False,  help='Number of iteration in one proccess')
    parser.add_argument('--N_search', dest='N_search', default=40, type=int,
required=False,  help='Number of selected sequences')
    parser.add_argument('--increase_len', dest='increase_len', default=True,
type=bool, required=False,  help='Increase len, bool')
    parser.add_argument('--nf', '--name_folder', dest='name_folder_root',
default='SDR', type=str, required=False,  help='Name folder for files')
    parser.add_argument('-o', '--name_file_out', dest='name_file_out', default='-',
type=str, required=False,  help='Name file for result')

    args = parser.parse_args()
```

Запуск основного алгоритма поиска при его распараллеливании и аналогичной процедуры увеличения длин.

```
    #запуск таймера времни
    time_start = time.time()

    #создание общей папки
    name_folder_root = args.name_folder_root

    if not os.path.isdir(name_folder_root):
        os.mkdir(name_folder_root)
```

```
    #создание общей папки, завязанной на время запуска
    name_folder = os.path.join(name_folder_root, str(int(time_start)))

    if not os.path.isdir(name_folder):
        os.mkdir(name_folder)

    #функция, необходимая для запуска основной программы по средствам Pool.map,
принимающей функцию с 1 аргументом
    #запуск основной программы с переданными начальными аргументами
    def iteration_run(x):
        return mult_iteration(x, N = args.N, repeat_set = args.repeat_subseq,
len_subseq = args.len_subseq,
                              E_val = args.E, number_set_of_subseq =
args.number_subseq,
                              s_name_txt_orig = args.s_name,
                              name_folder_root = name_folder, N1 = args.N_search,
number_of_iteration = args.number_iteration)

    #функция, необходимая для запуска функции увеличения длин по средствам Pool.map,
принимающей функцию с 1 аргументом
    def iteration_run_increase(x):
        return increasing_best(x, mean_predict_len_subseq = args.mean_len, working =
args.increase_len)

    #создание процессов по количеству доступных ядер и запуск основной программы
    with Pool(processes=cpu_count()) as pool:
        values = [j + 1 for j in range(args.number_subseq)]
        #results_folder - названия папок, созданных для отдельных процессов
        results_folder = pool.map(iteration_run, values)

    #отсчет времени работы основной программ поиска
    time_finish_search = time.time()

    #создание массива с названиями папок без повторений
    results_folder_set = set(results_folder)
    results_folder_diff = []
    for res_folder in results_folder_set:
        results_folder_diff.append(res_folder)

    #запуск по процессам функций увеличения длин подпоследовательностей
    with Pool(processes=cpu_count()) as pool2:
        results_N = pool2.map(iteration_run_increase, results_folder_diff)

    #получение максимального количества найденных подпоследовательностей и
определение процесса, в котором он найден
    max_N = max(results_N)
    max_N_folder = results_folder_diff[results_N.index(max_N)]

    #получение времени работы алгоритма поиска и алгоритма увеличения длин
    time_finish = time.time()
    time_res_seach = time_finish_search - time_start
    time_res_increase = time_finish - time_finish_search

    #print('N:', *results_N)
```

```
    #запись в файл или в основной поток вывода результата о времени, количества
найденных подпоследовательностей и названия папки, где лежит ответ
    s = 'N = {}, max_N_folder = {}, time_search = {}, time_increase =
{}'.format(max_N, max_N_folder, round(time_res_seach / 60, 2),
 round(time_res_increase / 60, 2))
    if args.name_file_out == '-':
        print(s)
    else:
        f = open(args.name_file_out)
        f.write(s)
        f.close()
```

Реализация основного алгоритма поиска повторов

```
def mult_iteration(j, N = 3, repeat_set = 6, len_subseq = 410, E_val = 1,
number_set_of_subseq = 8,
                   s_name_txt_orig = 'iter4/example_new.txt', name_folder_root =
'SDR1/1/', N1 = 40, number_of_iteration = 10):
    #j - номер начального набора
    #N - количество различных подпоследовательностей в начальном наборе
    #repeat_set - количество повторов подпоследовательностей в начальном наборе
    #len_subseq - длина подпоследовательностей в начальном наборе
    #E-val - E-value
    #number_set_of_subseq - количество начальных наборов
    #s_name_txt_orig - название файла с исходной последовательностью
    #name_folder_root - название папки, где сохранять результат
    #N1 - количество подпоследовательностей, которые нужно брать на шаге улучшения
    #number_of_iteration - количество итераций улучшения

    #создание папок под процесс
    name_process = os.getpid()

    name_folder = os.path.join(name_folder_root, str(name_process))

    if not os.path.isdir(name_folder):
        os.mkdir(name_folder)

    #создание исходной подпоследовательности в папке
    s_local_txt = 's.txt'
    s_name_txt = os.path.join(name_folder, s_local_txt)
    if not os.path.exists(s_name_txt):
        shutil.copy(s_name_txt_orig, s_name_txt)

    #создание исходной подпоследовательности в папке в формате FASTA
    s_local_fa = 's.fa'
    s_name_fasta = os.path.join(name_folder, s_local_fa)
    if not os.path.exists(s_name_fasta):
        sq.txt_into_fasta(s_name_txt, s_name_fasta)

    #создание файла для записи лучшего по процессу результата
    best_res_txt_local = 'best_result.txt'
```

```
    best_res_txt = os.path.join(name_folder, best_res_txt_local)
    if not os.path.exists(best_res_txt):
        f = open(best_res_txt, 'w')
        f.write('')
        f.close()

    #создание файла для записи позиций лучшего по процессу результата
    best_nhmm_posision_local = 'best_posision.fa'
    best_nhmm_posision = os.path.join(name_folder, best_nhmm_posision_local)
    if not os.path.exists(best_nhmm_posision):
        f = open(best_nhmm_posision, 'w')
        f.write('')
        f.close()

    #создание файла для записи текущего набора подпоследовательностей
    name_set_subseq = os.path.join(name_folder, 'set_subseq.fa')
    #создание файла для записи выравненного текущего набора подпоследовательностей
    name_set_subseq_msa = os.path.join(name_folder, 'set_subseq_msa.fa')
    #создание файла для записи текущего набора подпоследовательностей в формате
Stockholm
    name_set_subseq_msa_stockh = os.path.join(name_folder, 'set_subseq_msa.sto')
    #создание файла для записи текущей информации о работе nHMMER
    name_result_nhmmer_info = os.path.join(name_folder, 'result_nhmmer_info.fa')
    #создание файла для записи текущей информации о позициях, найденных nHMMER
    name_result_nhmmer_posision = os.path.join(name_folder,
'result_nhmmer_posision.fa')
    #создание файла для записи текущих найденных подпоследовательностей в формате txt
    name_set_res = os.path.join(name_folder, 'resulst.txt')

    #создание начального набора в формате FASTA
    sq.creating_fasta_set(s_name_txt, N, len_subseq, repeat_set, name_set_subseq)

    #запись количества лучших по потоку и текущих найденных наборов
    number_of_subseq_i = 0
    number_of_subseq_i_best = 0

    #запуск итерационной процедуры улучшения
    for i in range(number_of_iteration):
        #получение множественного выравнивания набора
        sq.muscle_msa(name_set_subseq, name_set_subseq_msa)
        #перевод его в формат Stockholm
        sq.fasta_into_stockholm(name_set_subseq_msa, name_set_subseq_msa_stockh)

        #запуск поиска наиболее схожих с текущем набором подпоследовательностей
исходной строки
        sq.nhmmer_on_stock_msa(name_set_subseq_msa_stockh, s_name_fasta,
name_result_nhmmer_info, name_result_nhmmer_posision, 20)
        #создание нового набора в формате FASTA
        sq.creating_fasta_new_nhmmer_set(s_name_txt, name_result_nhmmer_posision, N1,
name_set_subseq)
        #получение количества найденных подпоследовательностей
        number_of_subseq_i = sq.creating_txt_res_nhmmer_set(s_name_txt,
name_result_nhmmer_posision, min_len_subseq = 0, max_len_subseq = 1000, N =  10 ** 6,
name_res_txt = name_set_res)
```

```
        #критерий прекращения итерации улучшения
        if number_of_subseq_i <= N:
            break

    #получение лучшего числа найденных по потоку подпоследовательностей
    number_of_subseq_i_best = sq.creating_txt_res_nhmmer_set(s_name_txt,
best_nhmm_posision, min_len_subseq = 0, max_len_subseq = 1000, N =  10 ** 6,
name_res_txt = best_res_txt)

    #измениние лучшего по потоку значения в случае большем найденном количестве
подпоследовательностей
    if number_of_subseq_i > number_of_subseq_i_best:
        #изменение лучшего набора
        f = open(name_set_res, 'r')
        l = f.read()
        f.close()

        f = open(best_res_txt, 'w')
        f.write(l)
        f.close()

        #изменение лучших позиций
        f = open(name_result_nhmmer_posision, 'r')
        l = f.read()
        f.close()

        f = open(best_nhmm_posision, 'w')
        f.write(l)
        f.close()

    #возвращение названия папки по потоку
    return name_folder
```

Некоторые используемые функции в функции поиска:

- Создание начального набора

Запись в файл

```
def creating_fasta_set(name_S, N, len_subsequence, repeat_set = 1, name_fasta_set =
'example.fa'):
    #получение исходной последовательности
    S = string_from_file(name_S)

    #получение начального набора
    set_of_subseq = string_split(S, N, len_subsequence) * repeat_set

    #запись в файл в формате FASTA
    string_set_of_subset = ''
    for i in range(len(set_of_subseq)):
```

```
        string_set_of_subset +=  '>i' + str(i) + ' iteration;\n'
+str(set_of_subseq[i]) + '\n'
    file = open(name_fasta_set,'w')
    file.write(string_set_of_subset)
    file.close()
```

Создание массива подпоследовательностей

```
def string_split(S, N, len_subsequence):
    #получение номеров начал подпоследовательностей
    beginnings = number_split(len(S), N, len_subsequence)

    #создание массива подпоследовательностей
    set_of_subseq = [S[i:i + len_subsequence] for i in beginnings]
    return set_of_subseq
```

Создание начальных положений подпоследовательностей

```
def number_split(len_s, N, len_subsequence, dist = 0):
    #dist - минимальное расстояние между подпоследовательностями

    #создание массива для записи позиций начал
    beginnings = [0] * N

    for i in range(N):
        #генерация случайного числа таким образом, чтобы подпоследовательность влезла
в последовательность
        begin_i = random.randint(0, len_s - len_subsequence - 1)
        j = 0
        while j < i:
            #добавляем подпоследовательнсть в случае, когда она не пересекается ни с
одной из предыдущих
            if abs(beginnings[j] - begin_i) < (len_subsequence + dist):
                j = 0
                begin_i = random.randint(0, len_s - len_subsequence - 1)
            else:
                j += 1
        beginnings[i] = begin_i
    #возвращаем массив начальных позиций
    return sorted(beginnings)
```

- Создание нового набора по полученному из nHMMER файла с позициями в формате FASTA

Запись в файл

```
def creating_fasta_new_nhmmer_set(name_seq, name_out_posision, N = 20, name_fasta_set
= 'example.fa', left_step = 0, right_step = 0):
    #функция создание нового набора подпоследовательностей в формате FASTA
    #создается по файлу, созданному nHMMER
    #left_step и right_step - возможные расширения подпоследовательностей

    #создание массива подпоследовательностей
    set_of_subseq = creating_new_set_from_out_nhmmer(name_seq, name_out_posision, N,
left_step = left_step, right_step = right_step)

    #запись в файл
    string_set_of_subset = ''
    for i in range(len(set_of_subseq)):
        string_set_of_subset += '>i' + str(i) + ' iteration;\n' +
str(set_of_subseq[i]) + '\n'
    file = open(name_fasta_set,'w')
    file.write(string_set_of_subset)
    file.close()
```

Получение массива подпоследовательностей см. выше.

Операции по увеличению длин подпоследовательностей

```
def increasing_best(name_folder, mean_predict_len_subseq = 400, working = True):
    #применяет операцию увеличения длин до определенного среднего для лучшего по
потоку значения

    #получение файла с исходной последовательностью
    s_local_txt = 's.txt'
    s_name_txt = os.path.join(name_folder, s_local_txt)
    if not os.path.exists(s_name_txt):
        f = open(s_name_txt, 'w')
        f.write('')
        f.close()

    #получение файла с лучшими позициями по потоку
    best_nhmm_posision_local = 'best_posision.fa'
    best_nhmm_posision = os.path.join(name_folder, best_nhmm_posision_local)
    if not os.path.exists(best_nhmm_posision):
        f = open(best_nhmm_posision, 'w')
        f.write('')
        f.close()

    #получение файла с лучшим результатом по потоку
    best_res_txt_local = 'best_result.txt'
    best_res_txt = os.path.join(name_folder, best_res_txt_local)
    if not os.path.exists(best_res_txt):
```

```
        f = open(best_res_txt, 'w')
        f.write('')
        f.close()

    #создание файля для новых лучший позиций
    best_nhmm_posision_local_new = 'best_posision_new.fa'
    best_nhmm_posision_new = os.path.join(name_folder, best_nhmm_posision_local_new)
    if not os.path.exists(best_nhmm_posision_new):
        f = open(best_nhmm_posision_new, 'w')
        f.write('')
        f.close()

    #получение лучшего количества найденных повторов по потоку
    number_of_subseq =  sq.creating_txt_res_nhmmer_set(s_name_txt,
best_nhmm_posision, min_len_subseq = 0,
                                                       max_len_subseq = 1000, N =  10
** 6, name_res_txt = os.path.join(name_folder, 'best_without_increase_result.txt'))
    #запуск итерации улучшения
    if number_of_subseq >= 6 and working:
        number_of_subseq = sq.increase(s_name_txt, best_nhmm_posision,
                name_result_nhmmer_posision_new=best_nhmm_posision_new,
                name_folder=name_folder,
                best_name_res_txt = os.path.join(name_folder,
'best_with_increase_result.txt'),
                mean_predict=mean_predict_len_subseq, N=70, diff_variants=5)

    #возвращает новое улучшенное количество значений
    return number_of_subseq
```

Процедура улучшения

```
def increase(s_name_txt, name_result_nhmmer_posision,
             name_result_nhmmer_posision_new = 'increase2/name_result_pos.fa',
             name_folder = 'increase/',
             best_name_res_txt = 'best_result_posision_increase.fa',
             mean_predict = 400, N = 150, diff_variants = 5):

    #производит процедуру увеличения длин по файлу с позициями
    #mean_predict - средняя ожидаемая длина
    #N - количество подпоследовательностей для шага итерации
    #diff_variants - количество вариантов движений окна с позициями

    #создание папки для вспомогательных файлов
    name_folder = os.path.join(name_folder, 'increase')
    if not os.path.isdir(name_folder):
        os.mkdir(name_folder)

    #получение файла с подпоследовательностями
    creating_txt_res_nhmmer_set(s_name_txt, name_result_nhmmer_posision,
min_len_subseq = 0, max_len_subseq = 1000, N =  10 ** 6, name_res_txt =
os.path.join(name_folder, 'first_subseq.txt'))
    f = open(os.path.join(name_folder, 'first_subseq.txt'), 'r')
    set_subseq = f.read()
```

```
    f.close()

    #получение максимальной и минимальной длины подпоследовательности в наборе
    min_x = 10000
    max_x = 0
    for i in range(1, min(len(set_subseq), N), 2):
        if min_x > len(set_subseq[i]):
            min_x = len(set_subseq[i])
        if max_x < len(set_subseq[i]):
            max_x = len(set_subseq[i])

    #получение общего количества символов для увеличения
    diff = mean_predict - max_x

    #создание массива для результирующих длин в зависимости от положения окна
    res_number_all = []
    max_res_number = 0
    max_res_number_index = 0


    for i, left_diff in enumerate([j for j in range(0, diff + 1, diff //
diff_variants)]):
        #запуск увеличения по методу 2 для разных позиций окна
        name_folder_increase_i = os.path.join(name_folder, 'increase_' + str(i))
        if not os.path.isdir(name_folder_increase_i):
            os.mkdir(name_folder_increase_i)
        name_result_nhmmer_posision_new_i = os.path.join(name_folder, str(i) +
'_result_pos.fa')
        res_number = increase_len_subseq_msa(s_name_txt,
 name_result_nhmmer_posision,
                                             name_result_nhmmer_posision_new =
name_result_nhmmer_posision_new_i,
                                             left_step = left_diff, right_step = diff
- left_diff,
                                             name_folder = name_folder_increase_i,
                                             min_len_subseq = 20, max_len_subseq =
1000, N = N)
        res_number_all.append(res_number)
        #поиск лучшего положения окна
        if res_number >= max_res_number:
            max_res_number_index = i
            max_res_number = res_number
    #print(res_number_all)

    f = open(os.path.join(name_folder, str(max_res_number_index) + '_result_pos.fa'),
'r')
    res = f.read()
    f.close()

    #запись новых лучший позиций
    f = open(name_result_nhmmer_posision_new, 'w')
    f.write(res)
    f.close()

    #запись нового лучшего набора подпоследовательностей
```

```
    number_of_subseq = creating_txt_res_nhmmer_set(s_name_txt,
name_result_nhmmer_posision_new, min_len_subseq = 0, max_len_subseq = 1000, N =  10
** 6, name_res_txt = best_name_res_txt)

    #возвращает лучшее количество найденных повторов
    return number_of_subseq
```

Процедура улучшения по методу 2

```
def increase_len_subseq_msa(s_name_txt, name_result_nhmmer_posision,
                            name_result_nhmmer_posision_new =
'increase2/name_result_pos.fa',
                            name_folder = 'increase/',
                            left_step = 0, right_step = 0,
                             min_len_subseq = 20, max_len_subseq = 1000, N = 150):
    #создание локальных файлов
    name_set_subseq0 = os.path.join(name_folder, 'name_old_set_fasta.fa')
    name_set_subseq0_msa = os.path.join(name_folder, 'name_old_set_fasta_msa.fa')
    name_set_subseq0_msa_stockh = os.path.join(name_folder,
'name_old_set_fasta_msa.sto')
    s_name_fasta = os.path.join(name_folder,'s_name.fa')
    name_result_nhmmer_info = os.path.join(name_folder, 'name_result_nhmmer_info.fa')
    name_position_subseq = os.path.join(name_folder, 'pos_sub2.fa')
    name_position_subseq_msa = os.path.join(name_folder, 'pos_sub2_msa.fa')

    txt_into_fasta(s_name_txt, s_name_fasta)

    set_subseq = creating_new_set_from_out_nhmmer(s_name_txt,
name_result_nhmmer_posision, N = N, left_step = 0, right_step = 0,
name_position_subseq = name_position_subseq)

    #создание выравнивания по позициям
    muscle_msa(name_position_subseq, name_position_subseq_msa)
    f = open(name_position_subseq_msa, 'r')
    msa_pos = f.read()
    f.close()

    s = string_from_file(s_name_txt)

    #создание нового массива подпоследовательностей с заданным расширением
    set_of_subseq = []
    i = 0
    while i != -1:
        g = msa_pos.find(':', i + 1)
        begin, end = map(int, msa_pos[i + 1:g].split())

        i = msa_pos.find('>', i + 1)
```

```
        seq = msa_pos[g + 2:i - 1]
        seq = seq.replace('\n', '')

        j_left = 0
        k = seq[0]
        while k == '-':
            j_left += 1
            k = seq[j_left]
        begin -= j_left

        j_right = -1
        k = seq[-1]
        while k == '-':
            j_right -= 1
            k = seq[j_right]
        end -= j_right + 1

        #добавление смещения слева и справа
        if (begin - left_step) <= 0:
            begin = 0
        else:
            begin = begin - left_step

        if (end + right_step) >= (len(s) - 1):
            end = len(s) - 1
        else:
            end = end + right_step
        set_of_subseq.append(s[begin:end])

    #запись в новый набормв формате FASTA
    string_set_of_subset = ''
    for i in range(len(set_of_subseq)):
        string_set_of_subset +=  '>i' + str(i) + ' iteration;\n' +
str(set_of_subseq[i]) + '\n'

    file = open(name_set_subseq0,'w')
    file.write(string_set_of_subset)
    file.close()

    #совершение одной итерации улучшения по новому набору
    muscle_msa(name_set_subseq0, name_set_subseq0_msa)
    fasta_into_stockholm(name_set_subseq0_msa, name_set_subseq0_msa_stockh)
    nhmmer_on_stock_msa(name_set_subseq0_msa_stockh, s_name_fasta,
name_result_nhmmer_info, name_result_nhmmer_posision_new, N)

    name_res_txt = os.path.join(name_folder, 'second_subseq.txt')
```

```
    #возращает количество найденных последовательностей
    return creating_txt_res_nhmmer_set(s_name_txt, name_result_nhmmer_posision_new,
min_len_subseq = 0, max_len_subseq = 1000, N =  10 ** 6, name_res_txt = name_res_txt)
```