МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра дискретной математики и алгоритмики

МУРАВЕЙКО Даниил Олегович

АЛГОРИТМЫ НАХОЖДЕНИЯ ЛИНЕЙНЫХ НЕЙТРАЛИЗУЮЩИХ МОТИВОВ В МОЛЕКУЛЯРНЫХ ДАННЫХ

Магистерская диссертация
специальность 1-31 80 09 «Прикладная математика и информатика»

Руководитель
Мушко Вилена Владимировна
кандидат физико-математических наук

Допущена к защите

«____» _____ 2021 г.

Заведующий кафедрой дискретной математики и алгоритмики

_____ В.М. Котов

доктор физико-математических наук, профессор

Минск, 2021

ОГЛАВЛЕНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ

1. **E** — список ребер графа.
2. **GPU** — graphics processing unit — отдельное устройство персонального компьютера или игровой приставки, выполняющее графический рендеринг.
3. **HDFS** — Hadoop Distributed File System — распределенная файловая система Hadoop.
4. **N(v) —** список вершин-соседей вершины v в графе.
5. **V —** список вершин графа.
6. **ДНК** — Дезоксирибонуклеиновая кислота — макромолекула (одна из трёх основных, две другие — РНК и белки), обеспечивающая хранение, передачу из поколения в поколение и реализацию генетической программы развития и функционирования живых организмов.
7. **Мьютекс** — mutual exclusion — примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода.
8. **РНК** — Рибонуклеиновая кислота — одна из трёх основных макромолекул (две другие — ДНК и белки), которые содержатся в клетках всех живых организмов и играют важную роль в кодировании, прочтении, регуляции и выражении генов.
9. **Тред** — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.
10. **Фреймворк** — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.
11. **Δ(G) —** максимальная степень вершины в G.

ВВЕДЕНИЕ

Многие задачи могут быть сформулированы в виде графов, где граф состоит из набора вершин и набора ребер, в которых вершины обозначают рассматриваемые объекты, а ребра обозначают некоторые отношения между объектами. Клика — это подграф, в котором все пары вершин смежны. Таким образом, максимальная клика означает максимальную совокупность объектов, которые связаны между собой по некоторому заданному критерию. Так называемая проблема максимальной клики — одна из 21 проблем, NP-полнота которой была доказано Р. Карпом [1]. Поэтому существует убеждение, что задача о максимальной клике не решается за полиномиальное время. Тем не менее, по этой проблеме была проделана большая экспериментальная и теоретическая работа. Она привлекает большое внимание, особенно в последнее время, поскольку она нашла множество практических приложений в биоинформатике [2 — 5], разработке лекарств [6], распознавании образов [7], обработке изображений [8], кластеринге [9], сборе данных [10], разработке квантовых схем [11], разработке последовательностей ДНК и РНК для биомолекулярных вычислений [12], теории кодирования [13], проектировании беспроводных сетей [14] и многих других.

Молекулы можно описать как графы с вершинами, представляющими атомы или группы атомов, и ребрами, представляющими связи. Обнаружение сходства между молекулами, может быть выражено как проблема поиска максимальной клики в графах, полученных из сравниваемых молекул. В частности, алгоритм поиска максимальной клики позволяет обнаруживать структурные сходства белков, которые полезны при классификации белков или прогнозировании функций белков [15] и выполнять поиск структурных сходств в небольших соединениях в больших базах данных химических соединений, таких как ZINC [15], что является ключом к разработке новых лекарств.

В результате воздействия антигенных белков иммунная система пациента вырабатывает антитела, которые могут быть использованы в качестве биомаркеров для выявления рака или вирусной инфекции. Антиген (англ. antigen от antibody generator — «производитель антител») — любое вещество, которое организм рассматривает как чужеродное или потенциально опасное и против которого организм обычно начинает вырабатывать собственные антитела (иммунный ответ). Молекулярные взаимодействия между эпитопом антигена (эпитоп (англ. epitope), или антигенная детерминанта — часть макромолекулы антигена, которая распознаётся иммунной системой) и антителом часто определяются короткими линейными мотивами аминокислотной последовательности антигена. Такие взаимодействия могут быть обнаружены экспериментально с использованием библиотек случайных фаговых дисплеев и секвенирования нового поколения. В этом случае создается

библиотека всех возможных пептидов фиксированной длины, и пептиды, распознаваемые антителами, содержащимися в сыворотке крови человека, отбираются, амплифицируются в бактериях и секвенируются. Вычислительная проблема состоит в обнаружении истинных мотивов связывания, соответствующих эпитопам, связанным с диагностированным заболеванием.

Инструменты, разработанные для решения данной задачи, используют разнообразные алгоритмические методы, включая кластеризацию [15], нейронные сети [16] и оптимизацию исследуемых последовательностей [17]. Однако эти методы имеют ряд недостатков: многие из них не масштабируются для больших наборов данных, требуют предварительных знаний о количестве истинных последовательностей и имеют низкую точность [15]. Данные недостатки показывают, что современные инструменты требуют улучшения, что и является целью магистерской диссертации.

Для проверки предложенного алгоритма планируется использовать данные предоставленные и использованные авторами алгоритмов, доступных на данный момент [2 — 5]. Далее будут использоваться данные из Los Alamos National Labs HIV-1, NCBI Short Read Archive, European Nucleotide Archive, The Cancer Genome Atlas и других публичных репозиториев.

В первой главе рассмотрена проблема построения графа на основе молекулярных данных, введены необходимые понятия и теоремы, обозначена цель магистерской диссертации, кратко рассмотрен алгоритм поиска и расширения клик и введена специальная структура данных, необходимая для хранения графа и произведения операций над ним.

Во второй главе рассматривается алгоритм поиска максимальных клик, описаны существующие алгоритмы, а также их улучшения.

В третьей главе выявляются требования к задаче, генерация исходных тестовых данных, имплементация разработанного во второй главе алгоритма и тестирование получившегося решения.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Магистерская диссертация, 57 с., 18 рис., 24 источника, 2 приложения.

Ключевые слова: ГРАФ, К-УНИФОРМНЫЙ ГИПЕРГРАФ, БИОИНФОРМАТИКА, КЛИКА, РАСКРАСКА ГРАФА, ЗАДАЧА О КЛИКЕ, ПОИСК МАКСИМАЛЬНЫХ КЛИК, ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ, РАСШИРЕНИЯ КЛИКИ.

Объект исследования — проектирование и реализация алгоритма нахождения линейных нейтрализующих мотивов в молекулярных данных, что является решением задачи поиска списка максимальных клик в графе, для нахождения биомаркеров для выявления рака или вирусной инфекции.

Цель работы — изучить и проанализировать существующие подходы к решению задачи поиска максимальной клики и спроектировать и реализовать алгоритм поиска списка максимальных клик.

В ходе работы был произведен анализ существующих подходов к решению задач поиска максимальной клики, обоснование применения максимальных клик для нахождения биомаркеров для выявления рака или вирусной инфекции, был спроектирован, реализован последовательный и параллельный алгоритм поиска списка максимальных клик.

Результатом работы является приложение, реализованное на языке Java и Scala с использованием Spark и фреймворка Fractal, которое позволяет по списку ребер найти список максимальных клик за приемлемое время.

Область применения — медицина, программное обеспечение диагностирования рака или вирусной инфекции.

Структура магистерской диссертации: «Перечень условных обозначений, символов и терминов», «Введение», «Общая характеристика работы», основная часть, «Заключение», «Список использованной литературы» и «Приложение».

ABSTRACT

Master thesis, 57 pages, 18 images, 24 references, 2 appendices.

Keywords: GRAPH, K-UNIFORM HYPERGRAPH, BIOINFORMATICS, CLICK, GRAPH COLORING, CLICK PROBLEM, SEARCH FOR MAXIMUM CLIPS, PARALLEL ALGORITHM, CLICK EXTENSIONS.

The object of research is the design and implementation of an algorithm for finding linear neutralizing motifs in molecular data, which is a solution to the problem of finding a list of maximum clicks in a graph to find biomarkers for detecting cancer or viral infection.

The purpose of the work is to study and analyze existing approaches to solving the problem of finding the maximum clicks and to design and implement an algorithm for finding the list of maximum clicks.

In the course of the work, the analysis of existing approaches to solving the problems of finding the maximum clicks was carried out, the justification for the use of maximum clicks to find biomarkers for detecting cancer or viral infection, a sequential and parallel algorithm for finding the list of maximum clicks was designed and implemented.

The result of the work is an application implemented in Java and Scala using Spark and the Fractal framework, which allows finding the list of maximum clicks in a reasonable time using a list of edges.

The field of application is medicine, software for diagnosing cancer or viral infection.

The structure of the master thesis: "List of symbols and terms", "Introduction", “Abstract”, the main part, "Conclusion", "List of used literature" and "Appendix".

ГЛАВА 1
ОСОБЕННОСТИ ЗАДАЧИ

1.1 Введение в задачу

Кликой неориентированного графа называется подмножество его вершин, любые две из которых соединены ребром. Иными словами, это полный подграф первоначального графа. Размер клики определяется как число вершин в ней. Максимальная клика — это клика, которая не может быть расширена путём включения дополнительных смежных вершин, то есть нет клики большего размера, включающей все вершины данной клики. Наибольшая максимальная клика — это максимальная клика, размер которой наибольший. Размер клики определяется количеством вершин, входящих в эту клику. В графе, изображенном на рисунке 1, существует три клики — {*A, B, F*}, {*C, D, E*} и {*B*, *C*, *E*, *F*}. При этом клика {*B*, *C*, *E*, *F*} является наибольшей.

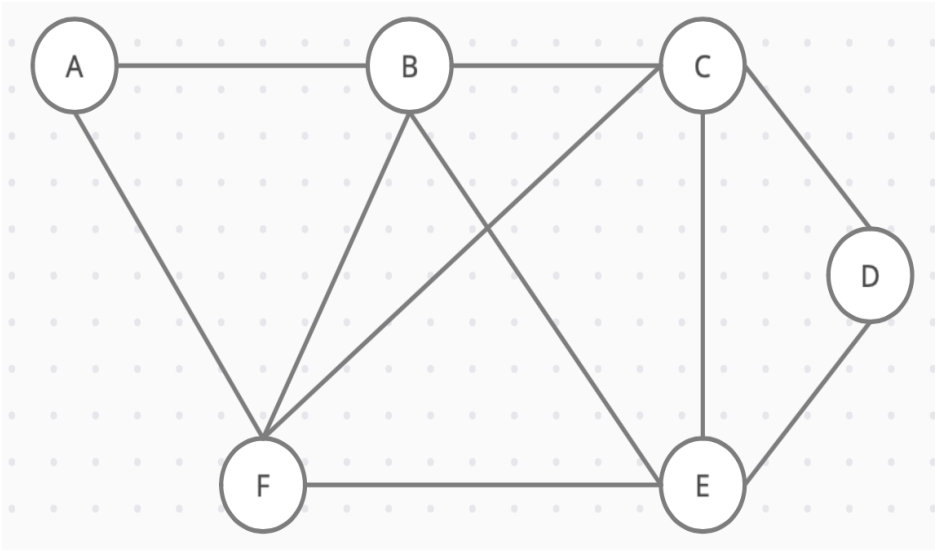

Рисунок 1. Пример графа.

Поиск максимальной клики — NP-полная задача [12], и вычислительно сложно получить эффективное точное решение, или хотя бы приближенное [13]. Несмотря на это, существует большое количество задач, которые могут быть сформулированы в виде задачи поиска максимальной клики. К настоящему времени достигнут значительный теоретический прогресс в разработке и анализе точных алгоритмов для поиска максимальной клики или максимального независимого множества (которое является максимальной кликой дополнительного графа), например, [16], [17] и [18]. Следует упомянуть, что алгоритм с теоретически лучшей сложностью не обязательно работает быстрее на практике. Следовательно, для решения задачи, поставленной в

магистерской работе, требуется разработать точный алгоритм поиска максимальных клик, который на практике работает очень быстро.
Однако, прежде чем начать поиск максимальной клики, необходимо построить граф, где будет происходить поиск этой клики.

1.2 Построение графа

Предположим, P — набор пептидов длины L. Для целого числа k, k-часть пептида $p \in P$, соответствующая подмножеству $J = \{j_1, ..., j_k\}$, представляет собой пару (s, r), где s представляет собой подстроку p, образованную аминокислотами в положениях J, а r — вектор чисел промежуточных положений между последовательными элементами J. Например, для p = KKEGLHD, k = 4 и J = {2, 4, 6, 7} соответствующая k-часть имеет вид (KGHD, (1, 1, 0)). Значение k обычно определяется пользователем. Так как пептиды в реальных данных обычно короткие (L < 10), можно запускать предложенный алгоритм с разными значениями k (k = 4 или 5 обычно достаточно).

Вместо того, чтоб рассматривать весь набор P, можно рассмотреть набор K всех уникальных k-частей всех пептидов. По множеству K, строится граф $G = G(K)$. Вершины представляют собой k-части, вершины смежные, если они различаются в двух аминокислотах, учитывая интервалы.

Пример графа пересечений для трех k-частей *v1, v2* и *v3* изображен на рисунке 2.

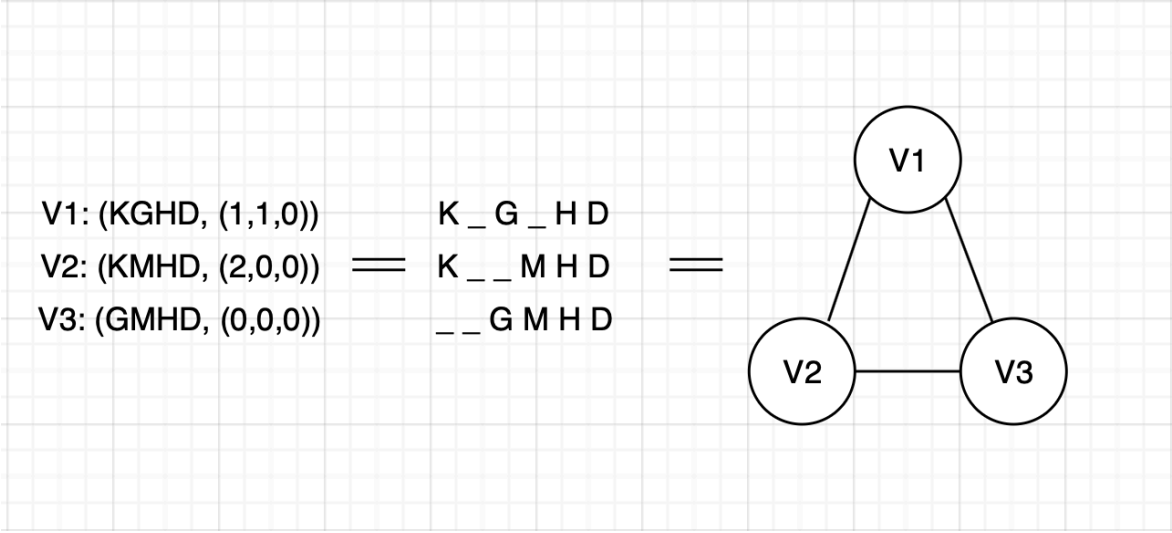

Рисунок 2. Пример построенного графа.

Для каждой вершины v создается набор (k — 1)-частей H(v), полученный удалением каждой из аминокислот из k-части. Например, для k-части *v1* из рисунка 1 $H(v1)$ имеет вид {((GHD, (1,0)), (KHD, (3,0)), (KGD, (1,2)), (KGH, (1,1))}. Вершины *v1* и *v2* смежные, если $H(v1)$ и $H(v2)$ содержат одинаковую (k-1)-часть, то есть $H(v1) \cap H(v2) \neq \varnothing$. В примере $H(v1)$ и $H(v2)$ содержат одинаковую часть (KHD, (3,0)).

Кликой в графе G является набор k-частей, любые две из которых соединены ребром, вместе с этим, клики представляют собой нейтрализующие мотивы. В то время как в общем случае генерация всех возможных максимальных клик, то есть клик, которые не могут быть расширены путём добавления смежных вершин, требует экспоненциального времени, в данном случае можно эффективно решить эту проблему, используя комбинаторные свойства графа G. По построению G является графом пересечений линейного k-униформного гиперграфа.

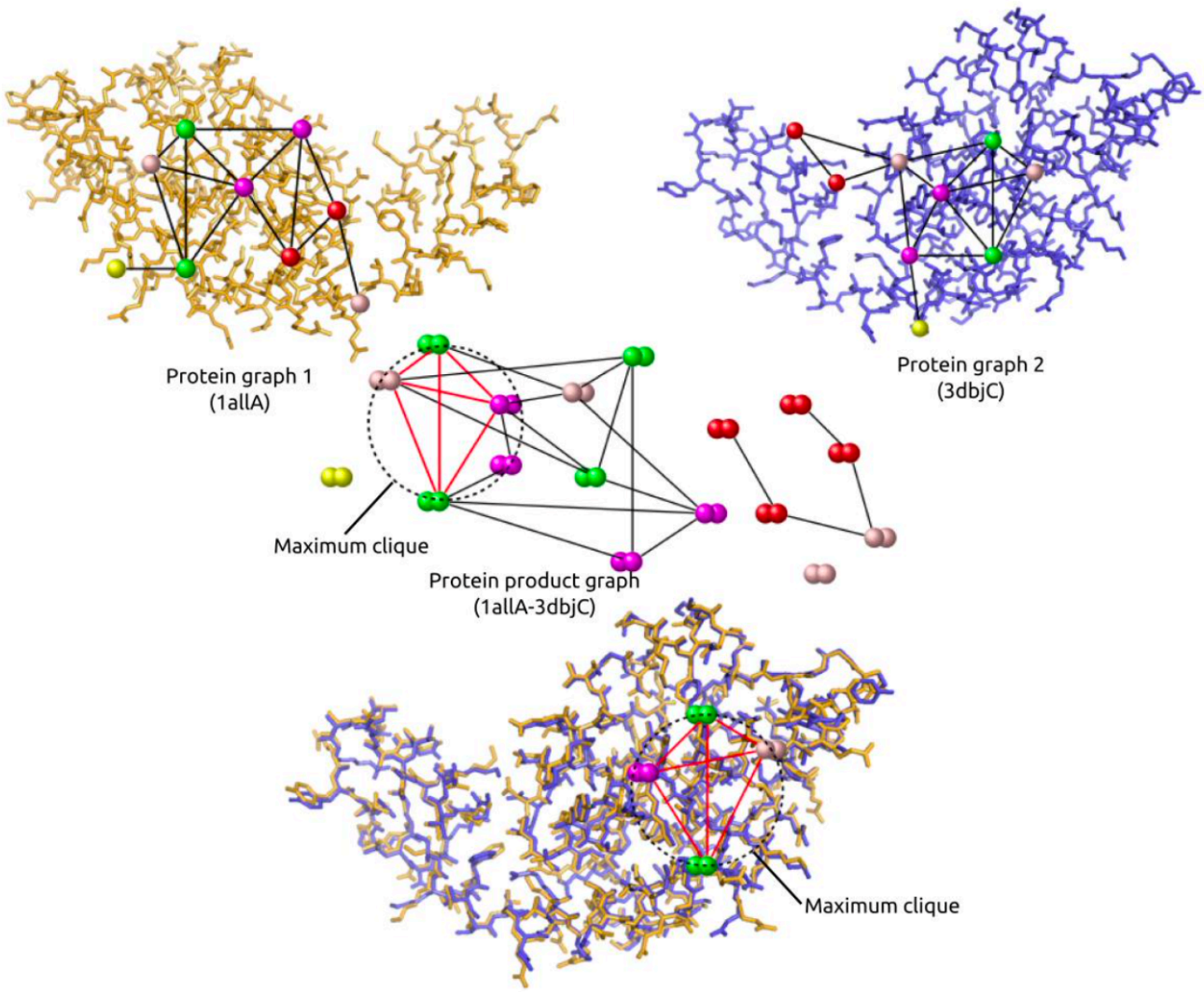

Рисунок 3. Биологическое применение клик.

На рисунке 3 показано сравнение структуры белков с использованием алгоритма поиска максимальных клик. Сначала белки преобразуются в графики белков (сверху рисунка) — для наглядности показана только часть каждого графика белков. Вершины окрашены в соответствии с их физико-химическими свойствами: акцепторный (красный), донорный (зеленый), стэкинг (розовый) и алифатический (желтый). График белкового продукта построен из обоих графиков белков (в центре). Максимальная клика из четырех вершин, соединенных красными краями, указывает на сходство между белками. Наконец, два сравниваемых белка накладываются друг на друга (внизу) в соответствии с наилучшим выравниванием вершин, представленных максимальной кликой.

После того, как клики-кандидаты найдены, из каждой клики будет собираться мотив. По определению, вершины клики соответствуют подпосседовательностям, которые идеально перекрываются друг с другом. Таким образом, их можно склеить друг с другом по пересечениям, и получить мотив. Мотивы, в свою очередь, соответствует так называемым эпитопам - частям вредоносной молекулы, к которым могут цепляться антитела, вырабатываемые иммунной системой. В итоге получен набор мотивов-кандидатов M. Затем выбираются надежные мотивы из полученного набора. Самый простой критерий — это выбрать мотивы, соответствующие наибольшим кликам. Чтобы исключить секвенирование химер (организмов, состоящих из генетически разнородных клеток) важно обеспечить, чтобы *k*-части в клике имели высокие значения. Таким образом, будут выбираться мотивы *M* с самыми высокими значениями d (*M*) = n(*M*) * c(*M*), где n(*M*) — размер соответствующей клики, а c(*M*) — количество k-частей этой клики.

1.3 K-униформный гиперграф

Введем необходимые определения и теоремы. Реберным графом *L(G)* называется граф, представляющий соседство рёбер графа *G*. Реберный граф *L(H)* гиперграфа *H* определяется условиями:

1) вершины графа *L(H)* биективно соответствуют ребрам гиперграфа *H*;
2) две вершины смежны в *L(H)* тогда и только тогда, когда соответствующие ребра гиперграфа *H* пересекаются.

Гиперграф называется *k*-униформным, если каждое его ребро имеет размер *k*, то есть содержит ровно *k* вершин. Гиперграф называется линейным, если любая пара гиперрёбер имеет в пересечении максимум одну вершину. Класс графов пересечений ребер линейных *k*-униформных гиперграфов обозначается L_k^l.

Клика, содержащая не менее k вершин, называется k-кликой.

Семейство $Q = (C_1, C_2, ..., C_q)$ клик графа G называется крауссовым k-разбиением графа, а клики C_i - кластерами этого разбиения, если:

1) каждое ребро графа G входит в один кластер C_i;
2) Каждая вершина графа G входит не более, чем в k кластеров разбиения Q.

Теорема 1 [10]. Каждая максимальная $\geq (k^2 - k + 2)$-клика графа является кластером каждого его крауссова k-разбиения.

Теорема 2 [11]. Граф принадлежит классу $G \in L_k^l$, если и только если существует крауссово k-разбиение этого графа.

В силу определения краусова k-разбиения окружение каждой вершины графа $G \in L_k^l$ содержит не более чем k попарно несмежных вершин. Поэтому, если степени вершин графа достаточно велики, то каждая вершина графа G входив в некоторую k-большую клику. Класс L_2^l реберных графов простых графов изучен очень хорошо, в частности, существуют эффективные алгоритмы для решения задачи распознавания $G \in L_2^l$. Ситуация существенно меняется, если k равен не двум, а трём и выше. В [10] доказано, что для фиксированного $k \geq 3$ задача распознавания $G \in L_k^l$ является NP-полной.

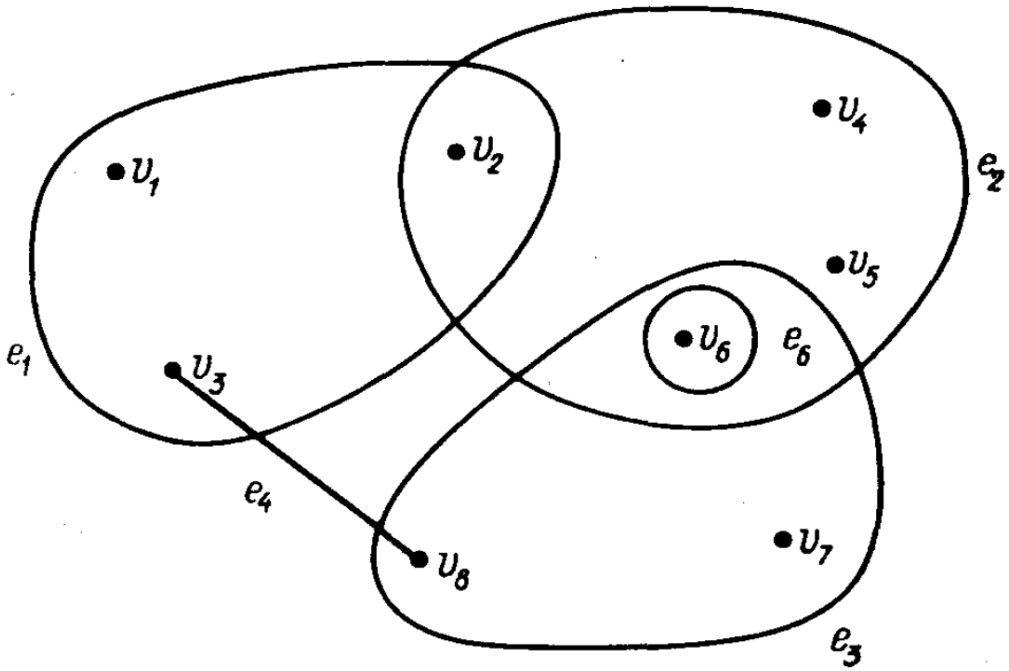

Рисунок 4. Пример гиперграфа.

На рисунке 4 показан пример гиперграфа, где e_1-e_4 — ребра графа, а v_1-v_7 — вершины графа.

Выводы

В первой главе было рассмотрено следующее:

- введение в задачу;
- алгоритм построения графа;
- биологический смысл клик;
- общий алгоритм работы;
- основные теоремы и определения.

ГЛАВА 2
АЛГОРИТМ ПОИСКА МАКСИМАЛЬНЫХ КЛИК

2.1 Существующие подходы

На решение задачи поиска максимальной клики было потрачено огромное количество усилий. Решение задачи поиска максимальной клики в литературе можно разделить на три группы. Первая группа включает централизованные последовательные алгоритмы. Вторая группа стремится распараллелить поиск по доступным процессорам или ядрам на одной машине, предлагая многопоточные подходы. Третья группа взяла на себя направление распределения экземпляра графа по нескольким машинам для обеспечения более высокой масштабируемости.

Алгоритмы из первой группы можно разделить на две группы: приближенные методы, включающие алгоритмы стохастического локального поиска, и точные алгоритмы. Приближенные алгоритмы могут решать большие и сложные задачи, но не могут гарантировать оптимальность их решений. Точные алгоритмы гарантируют оптимальность найденных решений. Поскольку задача поиска максимальной клики является NP-трудной, эффективных алгоритмов с точным полиномиальным временем еще не найдено. Однако было приложено много усилий для имплементации быстрых алгоритмов на практике. Одна из самых успешных парадигм для решения данной задачи называется методом ветвей и границ [7].

Метод ветвей и границ — общий алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. Метод является развитием метода полного перебора, в отличие от последнего — с отсевом подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Вторая группа алгоритмов исследовала распараллеливание поиска по ядрам на одной машине. Depolli [8] впервые представил очень эффективный централизованный алгоритм, объединив существующие блоки различных алгоритмов. Они объединили улучшенную приближенную раскраску из [9], начальное упорядочение вершин из [10] и использование битовых строк из [11]. Затем они эффективно распараллелили алгоритм в многопоточной среде. Rossi в [12] предложил параллельный алгоритм, ориентированный на большие разреженные графы. Они использовали характеристики социальных и информационных сетей и применяли агрессивные методы удаления. Далее они распараллелили поиск максимальной клики. Хотя они предложили универсальное распараллеливание, которое также можно использовать в

распределенных фреймворках, реализация авторов и эксперименты проводились только в многопоточной среде.

Третья группа алгоритмов использует распределенные фреймворки для достижения более высокой масштабируемости. Например, Pardalos в [13] предложили алгоритм max-clique на основе MPI. Однако модель MPI не обеспечивает отказоустойчивости. Кроме того, он не такой масштабируемый, как другие распределенные модели, такие как MapReduce. Распределенные алгоритмы max-clique, основанные на MapReduce, были предложены в [14], [15]. В [16] Peng впервые предложил наивный распределенный алгоритм. Они полагались на наивное случайное разбиение и не использовали никаких приемов отсечения. Xiang [17] затем предложил стратегию рекурсивного разделения, называемую BMC-разделением, которая обеспечивает хорошее распределение нагрузки. Затем они использовали разбиение BMC в эффективном и масштабируемом алгоритме MapReduce. Мощность их подхода была продемонстрирована путем решения, впервые, одного из больших графов из теста DIMACS, с использованием кластера из 128 машин.

В магистерской работе будет реализован точный алгоритм, основанный на методе ветвей и границ. Стандартный последовательный алгоритм поиска максимальной клики, реализующий метод ветвей и границ представлен в Алгоритме 1.

Алгоритм 1

```
 1.  w = 0
 2.  C = {}
 3.  P = G.V
 4.  extend(C, P)
6.
7.  Function extend(C, P):
8.        for v in P:
9.             if |Γp(v)| + |C| < w then
10.                 return
11.            endif
12.            C = C ∪ {v}
13.            P′ = Γp(v)
14.            if |P′| > 0
15.                 extend(C, P′)
16.            else
17.                 w = max(w, |C|)
18.            end if
19.            C = C - {v}
20.            P = P - {v}
```

```
21.       end for
```

Переменная w — глобальная переменная для размера максимальной клики, C — сет вершин в текущей клике и P — сет еще не обработанных вершин. На каждом шаге берется вершина из сета не обработанных вершин, если граничное условие выполняется, вершина добавляется в текущую клику, в сете остаются только вершины, смежные с текущей. Далее, рекурсивно вызывается функция расширения. После работы алгоритма в переменной C будет максимальная найденная клика.

В методе ветвей и границ наиболее важным моментом является эффективное сокращение пространства поиска с низкими накладными расходами. Фактически, значительное сокращение пространства поиска довольно легко достигается с высокими накладными расходами, но это приведет к увеличению общего времени работы.

Кандидатом сокращающего критерия может быть раскраска графа, а именно количество цветов, необходимых для раскрашивания графа, это верхняя граница размера максимальной клики в графе G. Действительно, так как клика размера k является полным подграфом, для ее раскраски необходимо ровно k цветов. Также, раскраска вершин графа даёт самые узкие границы для метода ветвей и границ [18].

2.2 Алгоритм раскраски графа

Алгоритм раскраски графа назначает цвета (или числа) вершинам в графе так, что две смежные вершины имеют разные цвета (или числа). Количество цветов, присвоенных алгоритмом, используется алгоритмом поиска максимальной клики в качестве граничного условия (как размер максимальной клики). Хотя раскраска графа это NP-сложная задача [1], существуют приближенные алгоритмы, основанные на жадной раскраске [16]. Сложность таких алгоритмов — $O(n^2)$ (n — количество вершин в графе). Пример такого алгоритма представлен на рисунке 5.

На первом шаге выбирается вершина A (как вершина с наибольшей степенью), ей присваивается текущий доступный цвет — 1. Далее вершина A добавляется в сет раскрашенных вершин U. Все соседи вершины A удаляются из Vcopy, а вершина A удаляется из V. Так как Vcopy пуст, то в него заново добавляются все вершины из V. Текущий цвет меняется на 2 (увеличивается на 1). Далее, из Vcopy выбирается вершина B, ей присваивается текущий доступный цвет (2), после этого B добавляется в U. Соседи вершины B удаляются из Vcopy, и B удаляется из V. Эти шаги выполняются пока V не пуст, тогда граф раскрашен и алгоритм прекращает свою работу.

Количество использованных цветов, то есть 4 в этом примере, является верхней границей размера максимальной клики, которая может быть найдена в

данном графе. Это число используется в алгоритме поиска максимальной клики в качестве границ в поисковом дереве.

На рисунке 5 показана визуализация алгоритма раскрашивания графа. Шаги идут слева направо, сверху вниз. На рисунке обведена обрабатываемая на каждом шаге вершина.

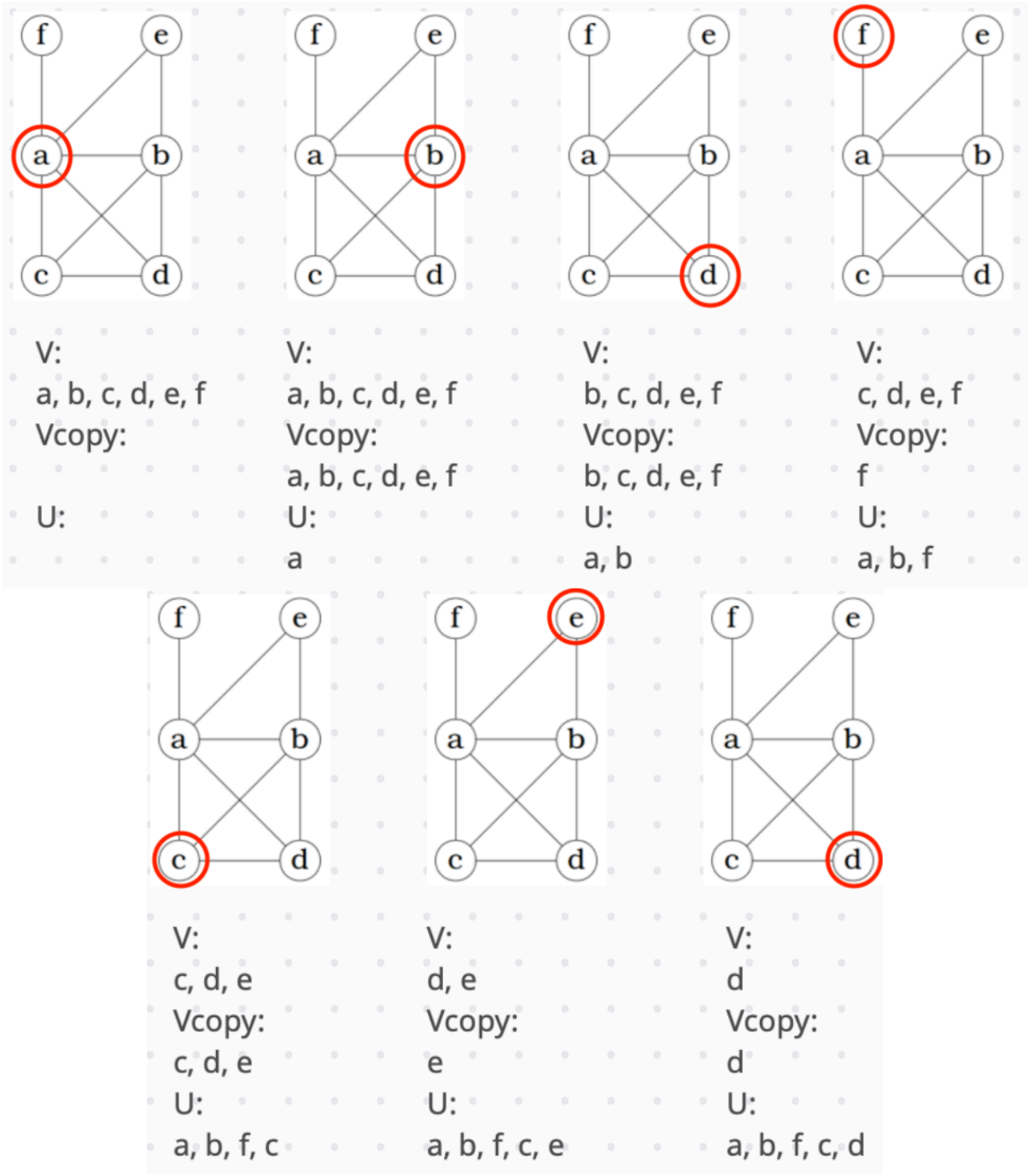

Рисунок 5. Визуализация алгоритма раскрашивания графа.

Tomita в [9] показал, что первоначальная сортировка вершин перед применением метода ветвей и границ может сильно повлиять как на

пространство поиска, так и на общее время выполнения. Он предложил алгоритм MCQ, который изначально сортирует вершины по степени и применяет простую начальную нумерацию / раскраску, где он устанавливает col[V[i]] = i для i $\leq \Delta$(G) и col[V[i]] = Δ(G) +1 для Δ(G) + 1 $\leq$ i $\leq$ |V|.

Изначальный порядок вершин в алгоритме раскраски графа значительно влияет на выполнение алгоритма поиска максимальной клики [11]. Раскрашивание уже, то есть для раскрашивания необходимо меньше цветов, если вершины расположены в не возрастающей последовательности их степеней. Таким образом, если степень u больше или равно степени v, u идет перед v. Однако остается некоторая свобода в упорядочивании вершин, имеющих одинаковые степени. Tomita в [10] описал эффективное начальное упорядочение вершин одной и той же степени с введением расширенной степени, определяемой для данной вершины как сумма степеней всех ее смежных вершин. При предварительной обработке вычисляется расширенная степень, за которой следует начальная сортировка по невозрастающей степени и невозрастающей расширенной степени для вершин равных степеней. После того, как вершины отсортированы, они перенумеровываются, так что вершина с наивысшей степенью имеет индекс один, вершина со второй высшей степенью — индекс два и так далее. Матрица смежности, то есть матрица размера n × n для графа с n вершинами, в котором элемент a_{ij} не равен нулю, если между вершинами i и j существует ребро, затем восстанавливается с помощью перенумерованных вершин. Следствием перенумерации является лучшая локализация доступа к памяти и, следовательно, более эффективное использование кэш-памяти. Кроме того, вводится дополнительный набор вершин, который поддерживает исходный порядок вершин на протяжении всего выполнения алгоритма поиска максимальной клики.

2.3 Список максимальных клик

Результатом текущей реализации алгоритма является наибольшая максимальная клика в графе. Однако, в исходной задаче данной магистерской работы необходимо найти список длины N (N задается пользователем) наибольших максимальных клик в графе. На данном этапе достаточно эффективно будет запустить поиск максимальных клик N раз, но перед каждой итерацией необходимо удалять уже найденные клики из графа. В [3] показано, что клики в графе не могут пересекаться по ребрам, то есть ребро графа может принадлежать только одной клике. Следовательно, для удаления клики из графа, достаточно удалить все ребра, принадлежащие этой клике. После этого необходимо перекрасить граф, так как степени, а следовательно, и порядок, вершин изменились, и количество используемых цветов тоже может измениться (уменьшиться). Алгоритм 2 показывает общую схему работы алгоритма поиска максимальных клик.

Алгоритм 2

```
Input: N, G
     1. C = {}
     2. G’ = G
     3. while len(C) < N:
     4.   max_color = color(G’)
     5.   clique = get_max_clique(G’, max_color)
     6.   C.add(clique)
     7.   G’ = cut_clique(G, clique)
```

На вход поступает граф G и количество искомых клик. В ходе работы алгоритма максимальные найденные клики сохраняются в сет C.

2.4 Улучшения существующих алгоритмов

Стандартный последовательный алгоритм поиска максимальной клики реализовывается через рекурсию. Данная особенность ограничивает размер максимальной искомой клики максимальной глубиной рекурсии, которая может зависеть от опций компилятора, от того, на какой платформе выполняется, в какой операционной системе и с какими настройками. Использование данного алгоритма подразумевает использование практически не контролируемого ресурса: стека вызовов. Это не только не предсказуемо, но и очень мало для задачи в данной магистерской работе. Известно, что любой рекурсивный алгоритм можно переписать без использования рекурсии. Не рекурсивная версия данного алгоритма представлена в Алгоритме 3.

Алгоритм 3

```
Input: G

    5.    w = 0
 6.  P = G.V
 7.  layer = first_extends(P)
 8.  cliques = {}
 9.  while True:
 10.    new_layer = extends_all(layer)
 11.    if new_layer == {}:
 12.      break
 13.    layer = new_layer
 14. end while
 15.
 16. Function first_extends(P):
 17.   subgraphs = {}
 18.   for v in P:
 19.     subgraphs.add({v}, Гр(v))
```

```
20.   end for
21.   return subgraphs
22.
23. Function extends_all(subgraphs):
24.   all_extends = {}
25.   for s, c in subgraphs:
26.     all_extends.add_all(extend(s, c))
27.   end for
28.  return all_extends
29.
30. Function extend(C, P):
31.   new_extends = {}
32.   for v in P:
33.     if |Гp(v)| + |C| > w - 1 then
34.       extend.add(C ∪ {v}, Гp(v))
35.     end if
36.   end for
37. return new_extends
```

Алгоритм 3 принимает на вход граф G. В переменной w хранится текущий максимальный размер клики. В переменной P — все вершины графа, функция `first_extends` производит расширение клики для каждой вершины, то есть каждая вершина берется в качестве клики, а ее список смежности — в качестве вершин кандидатов. Функция `extends_all` принимает массив пар — клика, список вершин кандидатов и производит расширение для каждой пары, возвращая новый массив пар. Самое интересное в этом алгоритме — функция расширения (`extend`). Эта функция принимает текущую клику и список ее вершин кандидатов и производит расширение клики, то есть для каждой вершины делает следующее: добавляет эту вершину в клику и фильтрует список кандидатов так, чтоб в нем остались только смежные с ней вершины. Таким образом в списке кандидатов всегда остаются вершины, смежные со всеми вершинами в клике. Данное действие показано на рисунке 6.

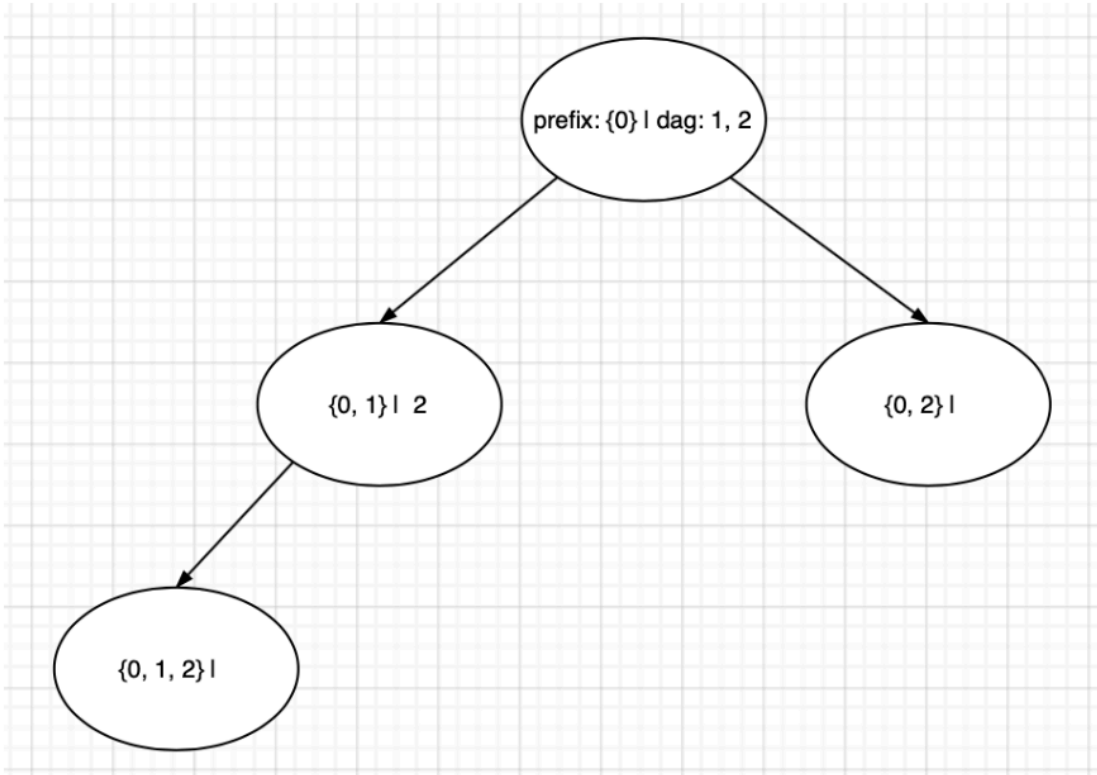

Рисунок 6. Схема работы функции extend.

Слева от вертикальной черты расположен сет вершин, которые образуют клику, а справа расположен сет вершин кандидатов на расширение клики. При переходе на следующий шаг по стрелкам список кандидатов фильтруется, и в нем остаются только актуальные вершины.

Однако, текущая версия не принимает размер клики, из-за этого может появиться существенное количество лишних расширений для вершин с заведомо недостаточными степенями (степень вершины меньше текущего размера искомой клики). Чтобы это исправить, достаточно добавить новый аргумент в алгоритм — размер искомой максимальной клики.

Алгоритм 4

```
Input: G, Size

1.   P = G.V
2.   layer = first_extends(P, Size)
3.   cliques = {}
4.   while True:
5.     new_layer = extends_all(layer, Size)
6.     if new_layer == {}:
7.      break
8.     layer = new_layer
```

```
9.   end while
10.
11.Function first_extends(P, Size):
12.  subgraphs = {}
13.  for v in P:
14.    if |Гр(v)| >= Size:
15.      subgraphs.add({v}, Гр(v))
16.  end for
17.return subgraphs
18.
19.Function extend(C, P, Size):
20.  new_extends = {}
21.  for v in P:
22.     if |Гр(v)| + |C| >= Size:
23.       filtered_N = {}
24.       for u in Гр(v):
25.         if |Гр(u)| + |C| + 1 >= Size:
26.           filtered_N.add(u)
27.       extend.add(C ∪ {v}, filtered_N)
28.     end if
29.  end for
30.return new_extends
```

Функция `extends_all` и основной поток алгоритма практически не изменились. Основные изменения произошли в функциях `extend` и `extends_all`: теперь все вершины проверяются перед добавлением в клики или в сет кандидатов на расширение.

Алгоритмы 3 и 4 не имеют зависимости от стека вызовов, однако, сейчас появилась новая проблема. Большие графы ведут к огромному числу подграфов, что потребляет большое количество памяти. Для решения этой проблемы понадобится специальная структура данных и алгоритм сохранения подграфов. Реализация данной структуры будет рассмотрена в третьей главе.

2.5 Параллельная версия алгоритма

В этом разделе представляется новый алгоритм параллельного поиска максимальных клик, основанный на алгоритме, описанном в предыдущем разделе. Блок-схема параллельного алгоритма показана на рисунке 7. В алгоритме используются методы многопоточности, которые поддерживаются в большинстве языков программирования без дополнительных библиотек или другой поддержки программного обеспечения. Это делает алгоритм переносимым на другие языки и операционные системы. Он может работать на большинстве современных многоядерных компьютерных архитектур. Параллельный алгоритм ведет себя идентично последовательному алгоритму

при выполнении на одном ядре, но для его выполнения требуется немного больше времени из-за добавленного кода управления потоками. Однако при выполнении на нескольких ядрах он легко компенсирует эти накладные расходы, если проблема максимальной клики, которую нужно решить, нетривиальна, например, графы с высокой плотностью.

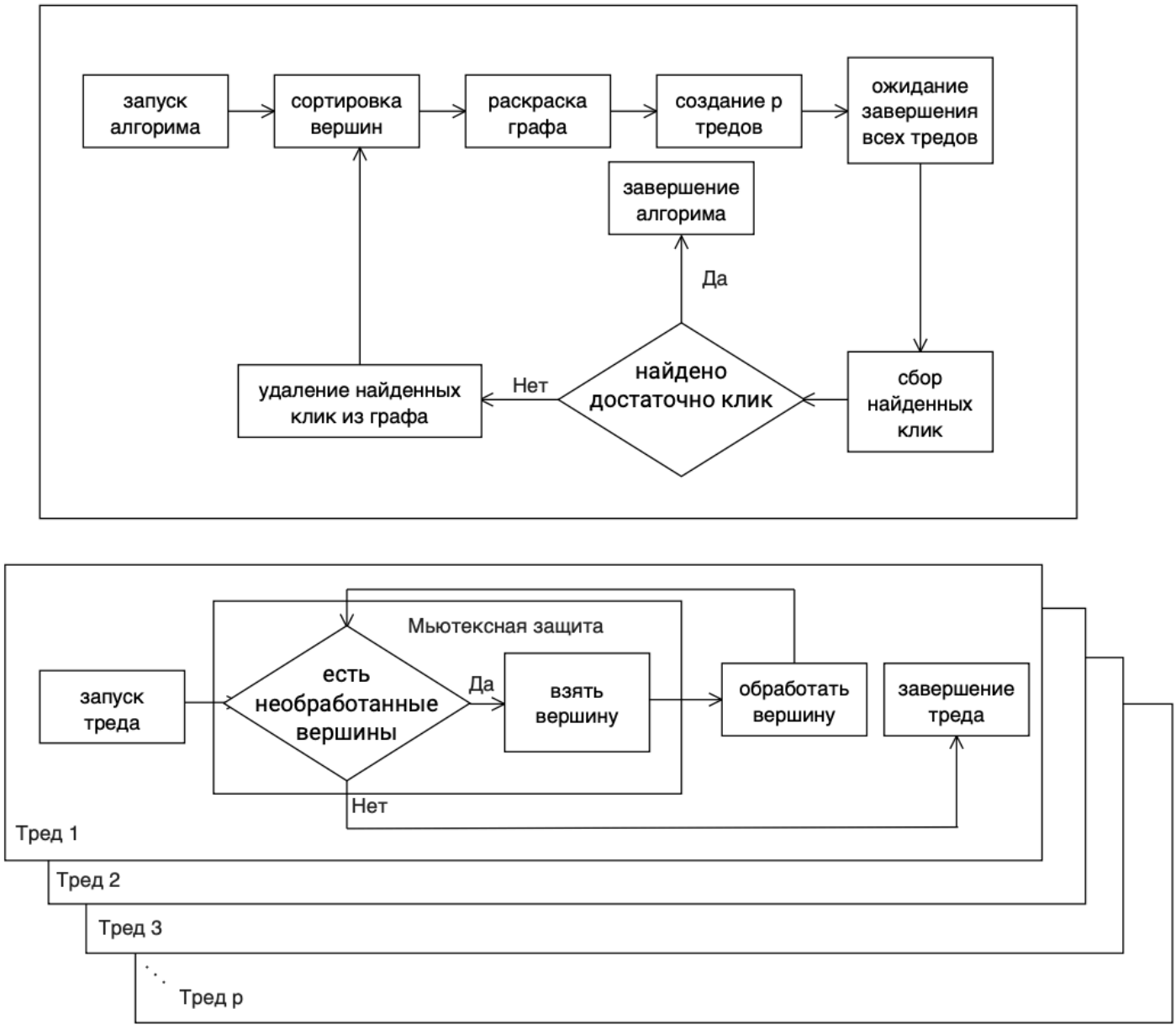

Рисунок 7. Блок-схема параллельного алгоритма.

Основной алгоритм (сверху на рисунке 7) начинается с этапа предварительной обработки, состоящего из первоначального упорядочивания и сортировки вершин. Этот шаг не распараллеливается, поскольку время его выполнения по сравнению с общим временем, необходимым для поиска

максимальной клики, имеет значение только на небольших и легко решаемых графах. Такие графы не являются целевыми графами данного алгоритма, и в них часто клики можно искать более эффективно с помощью последовательного алгоритма.

Алгоритм переходит к созданию p рабочих потоков, количество которых может быть установлено во время выполнения. Рабочие потоки создаются и запускаются последовательно, каждый поток на своем собственном ядре.

Следующий идет шаг основного алгоритма — дождаться завершения всех потоков и освободить их ресурсы. В результате работы — новые найденные клики. Далее необходимо проверить, что клик найдено достаточно (то есть больше либо равно числу N, задаваемому пользователем). Если клик меньше необходимого числа, найденные клики удаляются из графа и алгоритм повторяется с первого шага.

Каждый созданный поток (внизу на рисунке 7) входит в цикл и сначала проверяет наличие доступных заданий (вершин). Если доступных заданий нет, значит, вся работа уже назначена, и поток завершается. Если задание доступно, поток получает его и начинает обработку вершины. Последним и наиболее трудоемким действием является решение задания, то есть поиск максимальной клики в рабочем множестве вершин полученного задания. Это делается алгоритмом, который на вход принимает уже не весь граф, а рабочее множество вершин задания текущего треда.

Параллельная версия алгоритма будет реализована с помощью фреймворка Fractal (https://github.com/dccspeed/fractal) это высокопроизводительная система для поддержки распределенного анализа графовых шаблонов. Fractal рассматривает распределенную и параллельную среду, организованную одним мастером приложения и множеством рабочих. Как показано на рисунке 8, пользователь взаимодействует с Fractal, отправляя команды мастеру приложения (a). Мастер содержит механизм выполнения, который управляет базовыми ресурсами кластера и координирует выполнение шагов алгоритма (b).

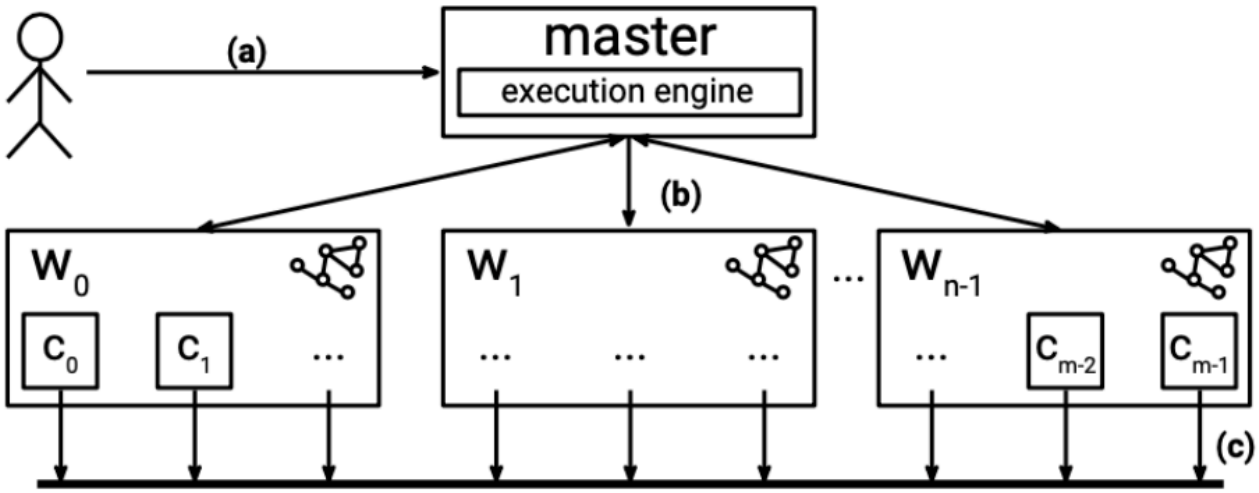

Рисунок 8. Схема архитектуры Fractal.

Коммуникация происходит между мастером и рабочими, но возможна и между рабочими. При запуске приложения происходит инициализация мастера, в которую входит создание необходимого количества рабочих потоков, а также чтение входного графа в памяти. Входные графы могут храниться в локальной файловой системе или в HDFS. Затем каждый рабочий отправляет регистрационное сообщение мастеру. Когда мастер подтверждает регистрацию рабочих, он передает их адреса. Таким образом, каждый рабочий знает, как связаться с других рабочими. Мастер создает набор необработанных вершин и рассылает рабочим их вершины. Если же рабочий закончил работу, он обращается к мастеру за новой.

Выводы

Во второй главе были рассмотрены следующие пункты:

- обзор существующих подходов и алгоритмов;
- описаны проблемы существующих алгоритмов;
- на основе существующих подходов был разработан алгоритм решения задачи поставленной в магистерской диссертации. Произведены необходимые улучшения и изменения;
- разработана параллельная версия алгоритма.

ГЛАВА 3
РЕАЛИЗАЦИЯ АЛГОРИТМА ПОИСКА СПИСКА МАКСИМАЛЬНЫХ КЛИК

3.1 Генерация данных

Прежде, чем приступить к реализации алгоритма поиска максимальных клик, необходимо создать генератор графов для проверки и тестирования алгоритма.

Как описано в [24] реальные графы являются 4-униформными графами с количеством вершин порядка 100000. Вершинами в таких графах являются 4-меры (только 4 аминокислоты существуют в последовательностях длиной 7). По схеме, описанной в первой главе, будет генерироваться набор пептидов, из которых далее создается набор K всех уникальных k-частей всех сгенерированных пептидов. Все 4-меры сравниваются по очереди. Если у двух 4-меров больше или равно 3 общих букв (аминокислот) в одинаковых позициях, то они являются смежными вершинами. Например, *a* — это A-TG-Y, *b* — это L-TG-Y, а *c* — это A-QT-Y. В этом примере *a* смежная с *b* и *c*, а *b* и *c* не являются смежными.

Алгоритм генерации последовательностей выглядит следующим образом:

Алгоритм 5

```
Input: preDefStr, n

 1.  retVal = new Vector
 2.  randLetIndx = 0
 3.  randD = 0.0
 4.  for i in (0, n):
 5.    temp = ""
 6.    for c in preDefStr:
 7.     if c == "-":
 8.       randLetIndx = random()
 9.       temp += aminoAcidLets.charAt(randLetIndx)
10.    else:
11.      randD = random.double()
12.      if randD > 0.5:
13.        temp = temp + preDefStr.charAt(j)
14.      else:
15.        randLetIndx = random()
16.        temp += aminoAcidLets.charAt(randLetIndx)
17.    retVal.add(temp)
18. return retVal
```

Алгоритм принимает аргументы `preDefStr` и `n`, где `preDefStr` — строка вида “A-TG-Y”, где буквы представляют собой аминокислоты, а n — сколько 4-меров, умноженных на 35, будет сгенерировано. 35 — это число сочетаний из 7 по 4. Таким образом, можно контролировать количество вершин в сгенерированном графе с шагом в 35.

Алгоритм работает по следующей логике: при прохождении по строке, если текущий символ прочерк — выбирается случайная аминокислота, если же буква — с вероятностью 50% выбирается аминокислота из `preDefStr` или случайная. В итоге получается список 4-меров, которые являются вершинами графа.

Далее необходимо построить список смежности. Для этого достаточно сравнить все сгенерированные 4-меры и, если у двух 4-меров есть 3 общие аминокислоты, стоящие на одинаковых позициях, соответствующие вершины смежные.

Алгоритм 6

```
Input: sequences

    adjacencyList = []
    for i in (0, sequences.len):
      for j in (0, sequences.len):
        if i == j:
          continue
        score = 0
        for k in (0, 7):
          if sequences[i][k] == sequences[j][k] and
            sequences[i][k] != ‘-‘:
          score += 1
      if score >= 3:
       adjacencyList[i].add(j)
  return adjacencyList
```

На вход поступает список сгенерированных 4-меров, функция построения сравнивает все 4-меры друг с другом, строит ребра, если критерий смежности выполнен, то возвращает список смежности сгенерированного графа.

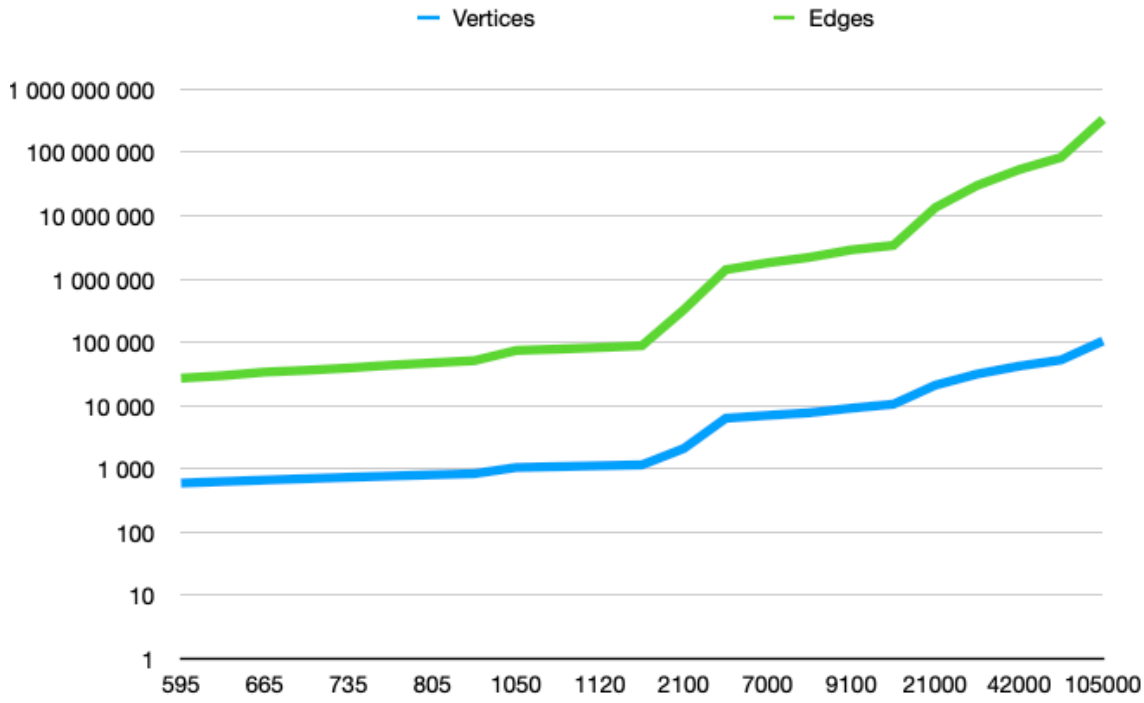

Рисунок 9. Количество ребер и вершин в сгенерированных графах.

На рисунке 9 показаны свойства сгенерированных графов. Шкала ординат — логарифмическое количество вершин и ребер, а шкала абсцисс — количество сгенерированных 4-меров. Размер реальных графов — около 100 000 вершин, поэтому максимальный сгенерированный граф имеет 105 000 вершин и 333 242 026 ребер.

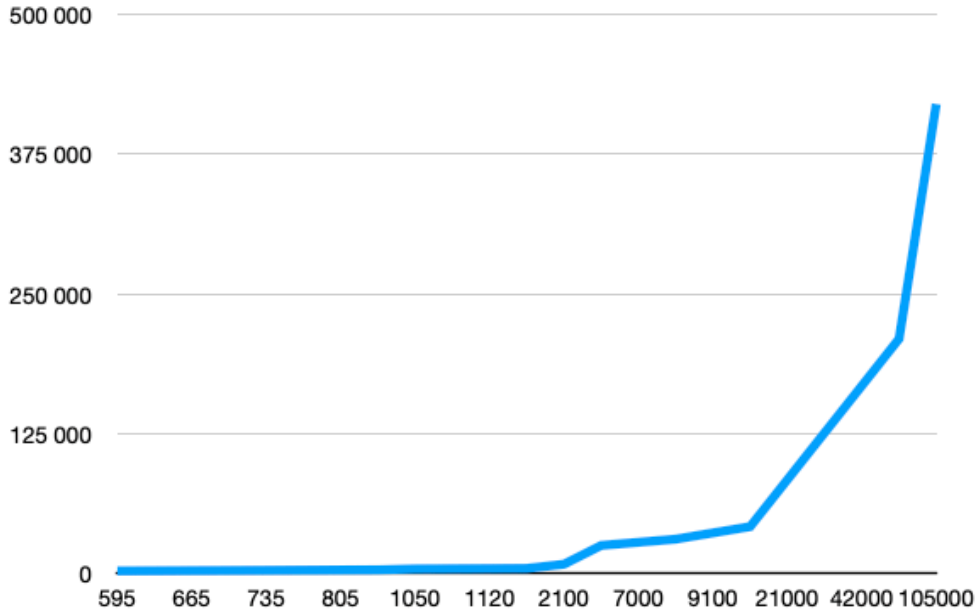

Рисунок 10. Размер максимальной клики, в зависимости от размера графа.

На рисунке 10 показаны размеры максимальных клик в сгенерированных графах. Как видно по графику, размер клики увеличивается пропорционально размеру графа.

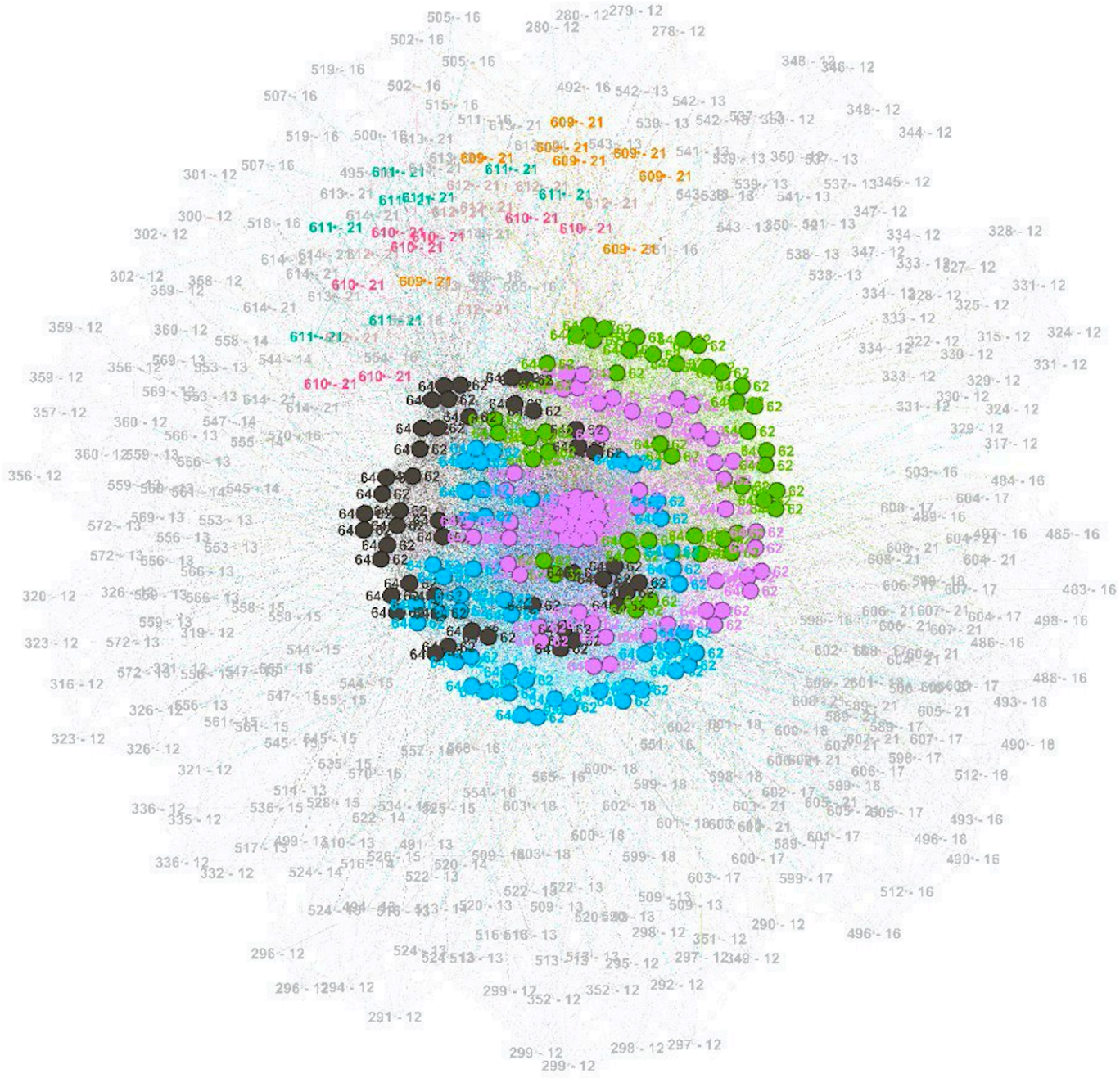

Рисунок 11. Пример сгенерированного графа.

На рисунке 11 показан пример графа с 525 вершинами и раскрашенными четырьмя максимальными кликами. На рисунке видно, что клики расположены достаточно близко друг к другу, поэтому граф будет кластеризоваться особым образом: из графа G (V, E) для каждой вершины строится новый граф G’, где V’ = {v} ∪ N(v), E’ = {(u, v) ∈ E | u ∈ V’, v ∈ V’}. То есть для каждой вершины

строится граф, содержащий всех ее соседей, и начало и конец всех вершин принадлежат множеству соседей текущей вершины.

3.2 Основные требования новой структуры данных

Обозначим функциональные и нефункциональные требования к новой структуре. Структура должна:

1) хранить текущие клики и список вершин-кандидатов (подграф);
2) не занимать много места в памяти;
3) иметь возможность удалять ветви расширений;
4) иметь возможность возвращаться к предыдущим кликам и их кандидатам.

Итак, практически всем требованиям удовлетворяет дерево следующего вида:

```
1. class ComputationTree
2. List<ComputationTree> children
3. ComputationTree parent
4. ComputationResult head
5.
6. class ComputationResult
7. SubgraphEnumerator candidates
8. Subgraph subgraph
```

Класс, который содержит поле родитель своего же типа и список детей своего же типа, а также ссылку на текущую клику и ее кандидатов. Удаление ветвей можно реализовать простым удалением элемента из списка детей. Возвращение к предыдущим кликам осуществляется с помощью поля родитель. Полный код приведен в приложении В.

3.3 Уменьшение потребляемой памяти

Произведем тестовый запуск на сгенерированном графе, в котором 105 000 вершин и 333 242 026 ребер. На рисунке 12 показан график потребляемой приложением памяти.

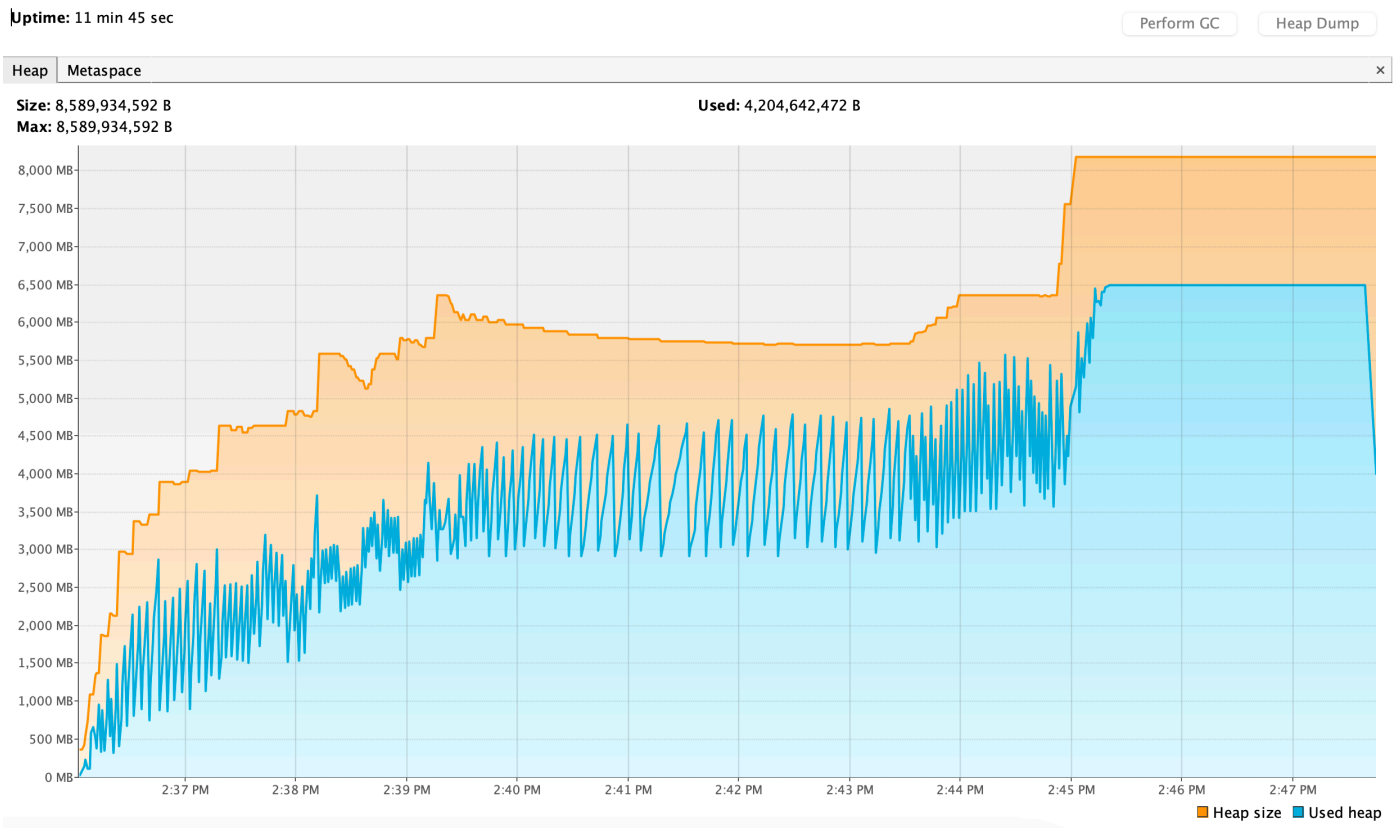

Рисунок 12. График потребляемой памяти

Итоговая структура занимает даже больше места, чем предыдущие варианты алгоритма, что ведет к OutOfMemoryError. Так как приложение завершилось аварийно, полный размер потребляемой памяти не известен. Решить OutOfMemoryError можно с помощью ленивых расширений и расширений в глубину, а не в ширину. Идея данного алгоритма заключается в том, что программе не нужно запоминать и хранить состояния всех ветвей и узлов в дереве расширений, достаточно знать только состояние текущей расширяемой вершины. Рассмотрим построение дерева расширений для тестового графа, изображенного на рисунке 13.

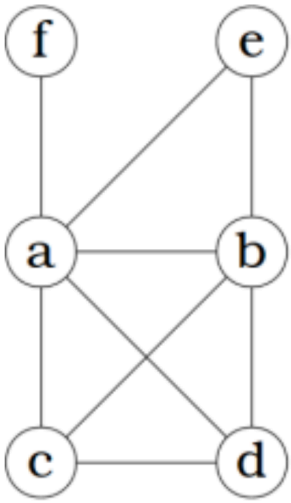

Рисунок 13. Тестовый граф

На рисунке 14 показано, как будет выглядеть дерево расширений.

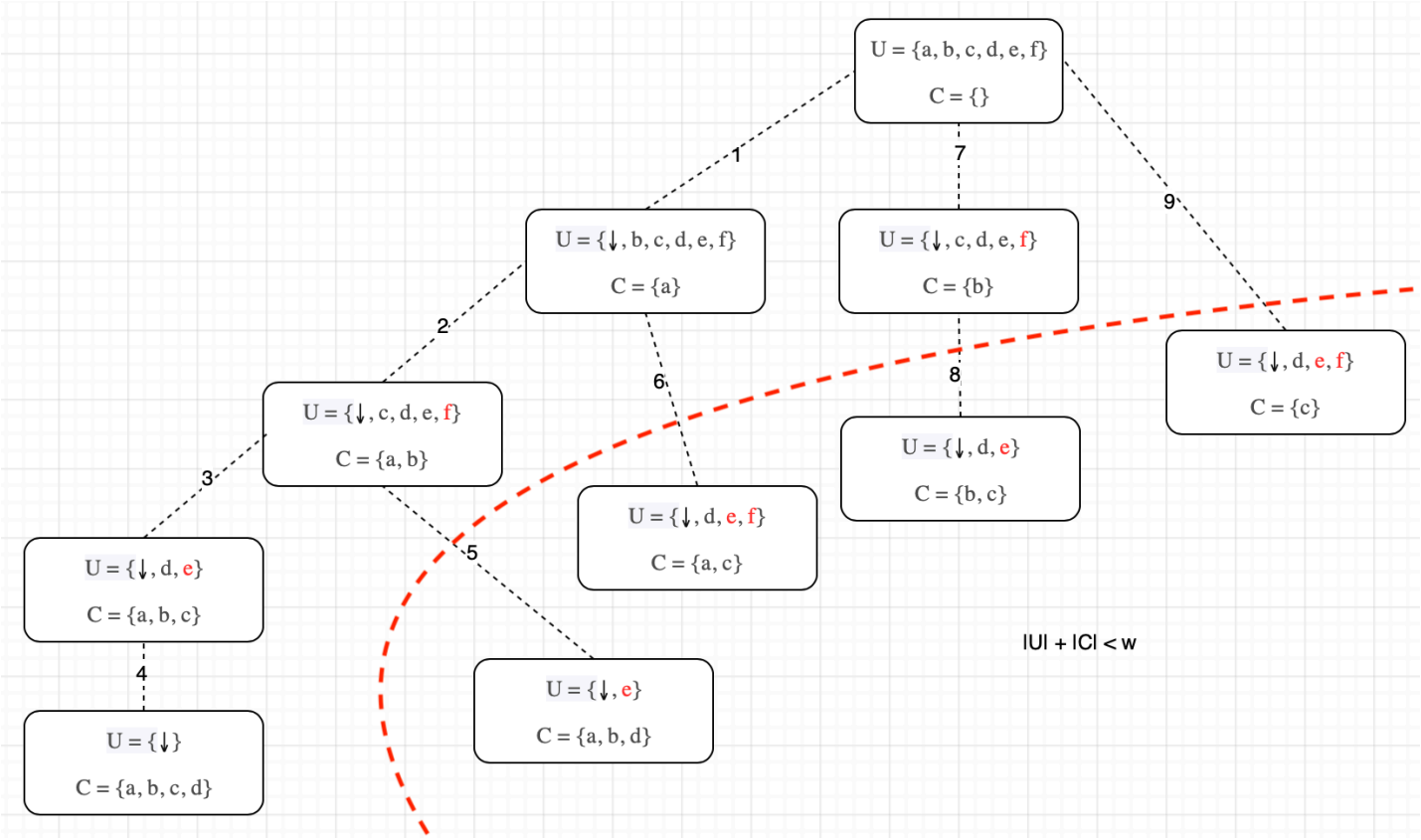

Рисунок 14. Дерево расширений.

Все, что ниже красной пунктирной линии, отсекается проверкой на максимальный возможный размер клики |U| + |C| < w. Цифры на пунктирных линиях показывают порядок расширений клик. Стрелка вниз обозначает вершину, которая добавляется в клику, а красный шрифт показывает вершины,

которые удаляются из списка вершин-кандидатов. Данный алгоритм описан в алгоритме 7.

Алгоритм 7

```
Input N - количество искомых клик
 1.  w = 0
 2.  P = G.V
 3.  layer = first_extends(P)
 4.  cliques = {}
 5.  head = layer.head
 6.  done = false
 7.  while cliques.size < N and not done:
 8.  new_head = extends_first(head)
 9.  if new_head == null:
 10. head, done0 = get_new_head(head)
 11. done = done0
 12. end while
 13.
 14. Function first_extends(P):
 15. subgraphs = {}
 16. for v in P:
 17. subgraphs.add({v}, Γp(v))
 18. end for
 19. return subgraphs
 20.
 21. Function extends_first(subgraphs):
 22. all_extends = {}
 23. s, c = subgraphs.head
 24. return extend(s, c)
 25.
 26. Function extend(C, P):
 27. new_extends = {}
 28. for v in P:
 29. if |Γp(v)| + |C| > w - 1 then
 30. new_extends.add(v)
 31. end if
 32. end for
 33. return (P, new_extends)
 34.
 35. Function get_new_head(head):
 36. child = head.get_new_child()
 37. if child == null:
 38. back = true
```

```
39. child == head
40. while child.has_parent() && back:
41. child_first = child.visit()
42. if child_first != null:
43. head = child_first
44. back = false
45. end if
46. end while
47. if head == null:
48. done = true
49. else
50. head = child
51. end if
52. return (head, done)
```

Реализуем Алгоритм 7 и запустим приложение с исходным тестовым графом. Рисунок 15 показывает результат запуска.

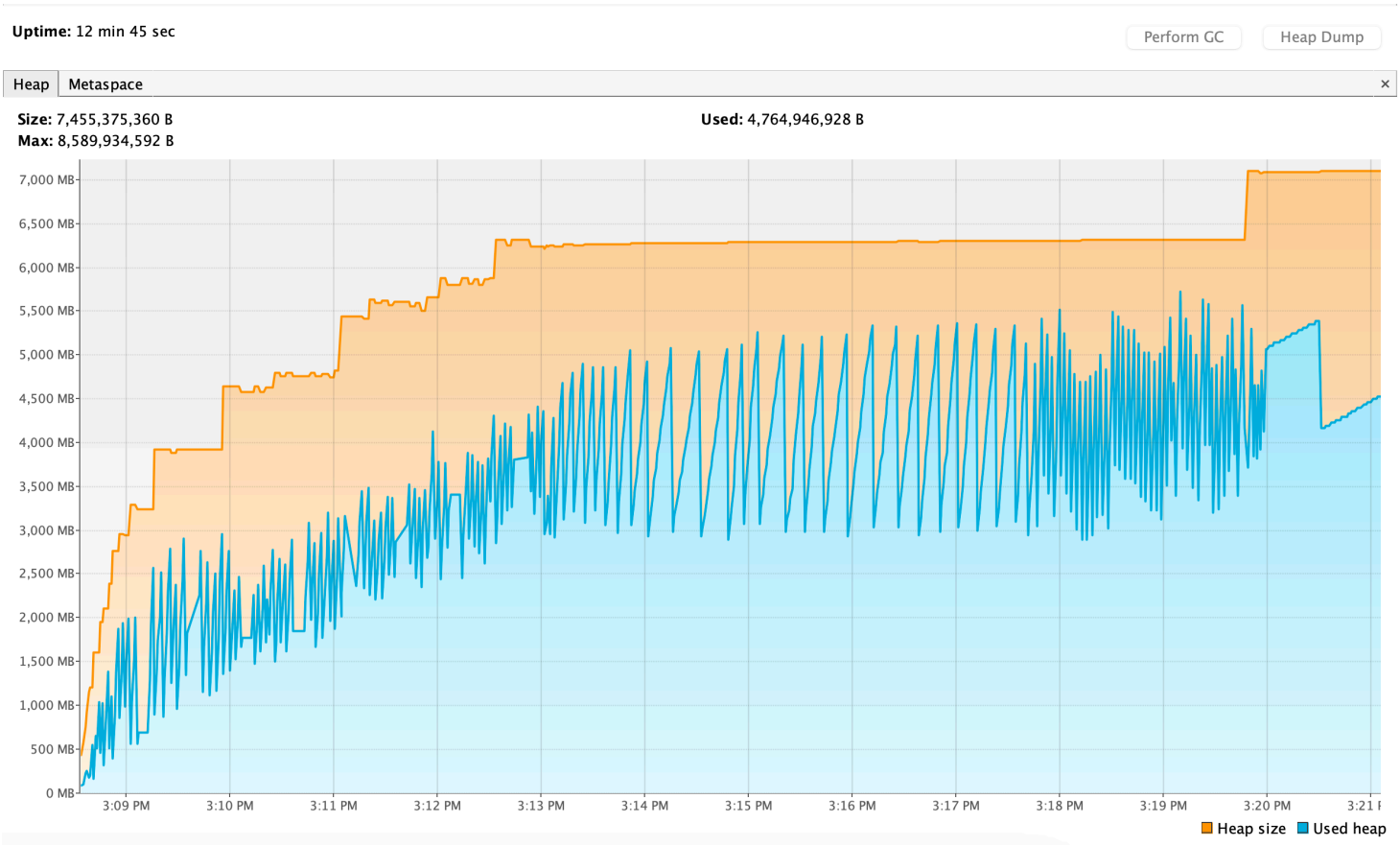

Рисунок 15. График потребляемой памяти с ленивым расширением.

Как видно по графику, OutOfMemoryError уже не появляется, но структура всё еще занимает много места — 5500 MB.

3.4 Chronicle Map

Уменьшить размер потребляемой памяти можно с помощью Chronicle Map. Chronicle Map (https://github.com/OpenHFT/Chronicle-Map) — это сверхбыстрое не блокирующее хранилище ключей и значений в памяти, предназначенное для приложений с малой задержкой или многопроцессорных приложений. Размер Chronicle Map ограничен не оперативной памятью, а доступной емкостью диска. Промежуточные подграфы и список вершин-кандидатов будут храниться в Chronicle Map, а в дереве расширений будут содержаться только ключи к соответствующим значениям. Внесем небольшие изменения в Алгоритм 7, добавив работу с Chronicle Map. Изменятся всего две функции, поэтому для краткости приведем только их в Алгоритме 8.

Алгоритм 8

```
1.  Function extend(C, P, map):
2.  for v in P:
3.  if |Гp(v)| + |C| > w - 1 then
4.  key = generate_key()
5.  map.put(key, v, Гv(P))
6.  end if
7.  end for
8.  return key
9.
10. Function generate_key():
11. return generate_random_int()
12.
13. Function extends_first(subgraphs, map):
14. s, c = map.get(subgraphs.head)
15. return extend(s, c)
```

Произведем повторный запуск программы на тех же данных, но с Chronicle Map. График потребляемой памяти показан на рисунке 16.

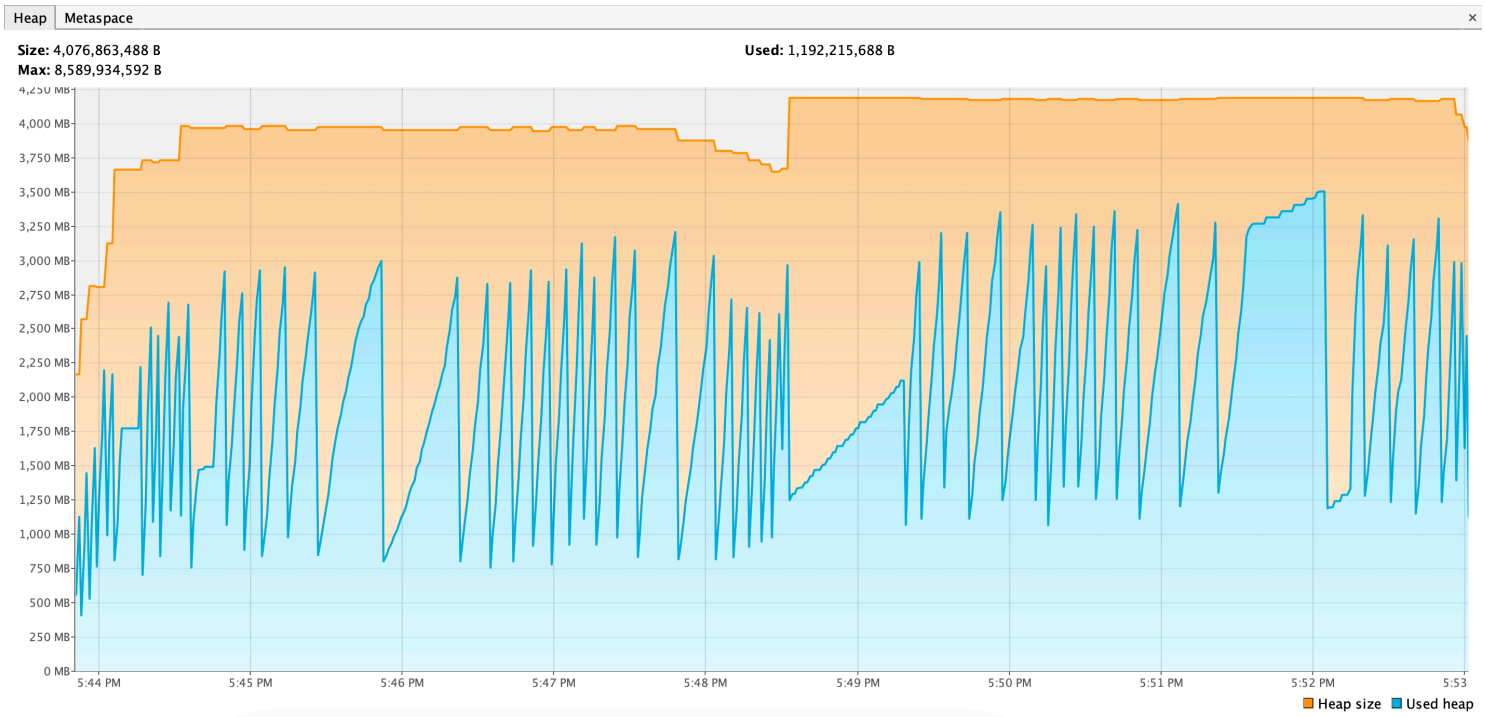

Рисунок 16. График потребляемой памяти с Chronicle Map.

Итоговая реализация занимает практически в два раза меньше места. Полная реализация данного алгоритма приведена в приложении А.

3.5 Параллельная версия

В параллельной версии потоки исследуют несколько ветвей дерева поиска одновременно, каждая ветвь начинается в отдельной начальной вершине, как показано на рисунке 17 в примере графа с рисунка 13. Поток 1 исследует ветвь, имеющую вершину a в качестве корня и все другие вершины в качестве кандидатов. Поток 2 исследует другую ветвь, полученную первым разделением задания, и имеет вершину b в качестве корня и все другие вершины, кроме вершины a, в качестве кандидатов. Количество вершин-кандидатов в рабочем наборе U уменьшается с каждым дополнительным потоком, и соответственно уменьшается среднее время выполнения, поэтому в первую очередь ищутся сложные, наиболее трудоемкие ветви. Когда потоки, исследующие сложные ветки, завершают свою работу, остаются только те потоки, которые исследуют простые ветки, что обеспечивает низкое время простоя и хорошую балансировку нагрузки. Аналогично рисунку 14, всё что под пунктирной линией отсекается проверкой на максимальный возможный размер клики $|U| + |C| < w$.

В отличие от последовательного алгоритма (рисунок 14), параллельный алгоритм исследует две ветви одновременно (рисунок 17). Это уменьшает количество шагов, то есть с 9 до 7 в последовательном алгоритме, и приводит к более быстрому времени выполнения (ускорению) алгоритма.

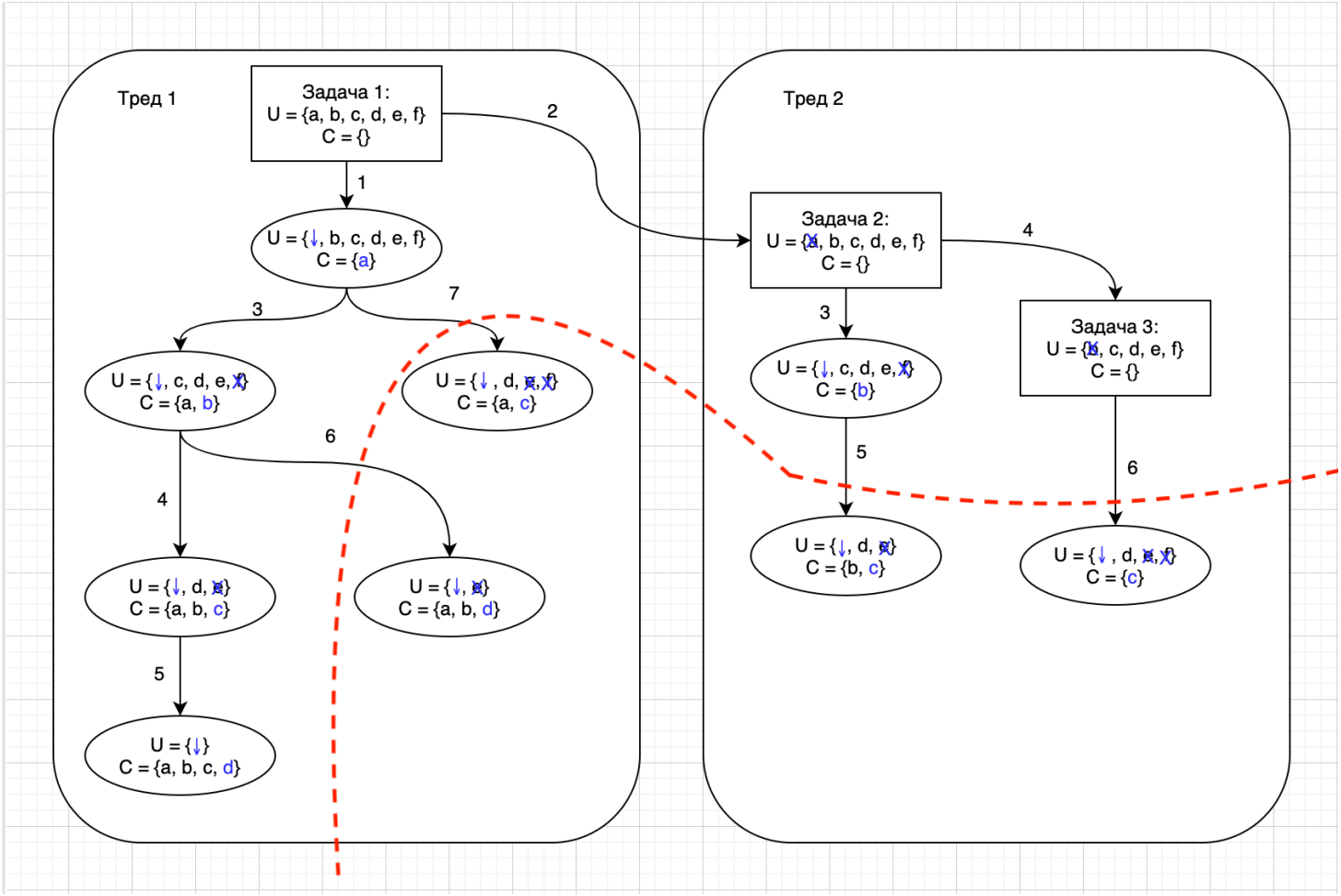

Рисунок 17. Параллельная версия дерева расширений.

3.6 Результат работы

Алгоритм был реализован с помощью языков Scala и Java и фреймворков Spark и Fractal. Программа запускалась на MacBook Pro (2018) с процессором 2.6 GHz 6-Core Intel Core i7. На рисунке 18 на шкале ординат показано время работы в секундах в логарифмическом масштабе, в зависимости от размера графа (количество вершин) на шкале абсцисс.

В тестах производился поиск 5 максимальных клик. Клики в графах реального размера, а именно графы с 105000 вершинами и 333000000 ребрами, вычисляются на данном компьютере за 1834 секунды или за 30 минут, что является отличным результатом на данном этапе.

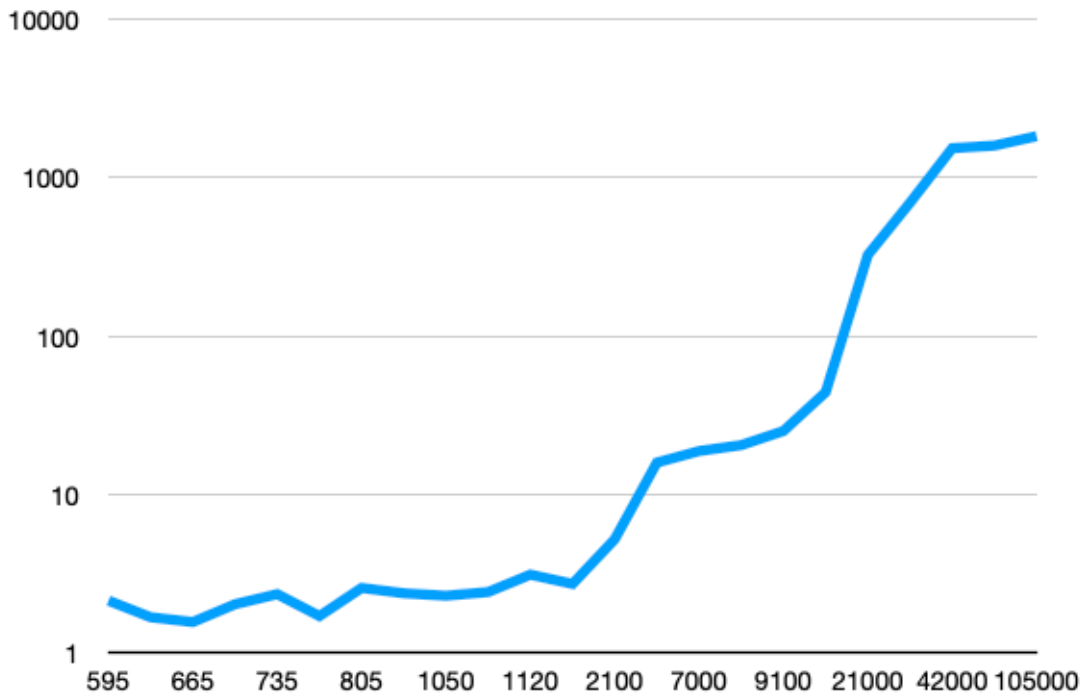

Рисунок 18. Время работы приложения, в зависимости от размера графа.

3.7 Валидация алгоритма

Для того, чтобы оценить и проверить реализованный алгоритм, были сгенерированы образцы последовательностей разной длины и были имплантировали предопределенные мотивы в образцы. Далее алгоритм был применен к смоделированным данным. После успешного прохождения всех тестов была инициирована проверка на реальных данных из Los Alamos National Labs HIV-1, NCBI Short Read Archive, European Nucleotide Archive, The Cancer Genome Atlas и других публичных репозиториев. В данный момент алгоритм проходит валидацию в университете штата Джорджия.

Выводы

В третьей главе были рассмотрены следующие пункты:

- реализация алгоритма построения графа и генерация тестовых графов по 4-мерам;
- проектирование и реализация специальной структуры данных для хранения промежуточных результатов (вершин кандидатов и промежуточных клик);
- реализация первой версии алгоритма и реализация существенных улучшений данного приложения;
- разработка параллельной версии алгоритма;
- начальная валидация реализованного алгоритма.

ЗАКЛЮЧЕНИЕ

Молекулы можно описать как графы с вершинами, представляющими атомы или группы атомов, и ребрами, представляющими связи. Обнаружение сходства между молекулами, может быть выражено как проблема поиска максимальной клики в графах, полученных из сравниваемых молекул. В частности, алгоритм поиска списка максимальных клик позволяет обнаруживать структурные сходства белков, которые полезны при классификации белков или прогнозировании функций белков, и выполнять поиск структурных сходств в небольших соединениях в больших базах данных химических соединений, таких как ZINC, что является ключом к разработке новых лекарств.

В рамках данной диссертационной работы проанализированы материалы и существующие подходы по проблеме поиска максимальных клик. Было изучено большое число существующих алгоритмов. На основе этих данных был спроектирован и реализован, с помощью языков программирования Java и Scala, а также фреймворков Fractal и Spark, последовательный и параллельный алгоритм поиска списка максимальных клик. Так же была изучена и реализована схема построения графа по набору пептидов.

Данное решение было успешно протестировано на сгенерированном наборе графов, показало приемлемое время работы на графах реального размера. В данный момент реализованное приложение проверяется на реальных данных из Los Alamos National Labs HIV-1, NCBI Short Read Archive, European Nucleotide Archive, The Cancer Genome Atlas и других публичных репозиториев.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Об алгоритме распознавания графов пересечений ребер линейных 3-униформных гиперграфов // [Электронный ресурс]. — 2019. Режим доступа : https://scholar.google.com/scholar_host?q=info:D-PF7LvdT9QJ:scholar.google.com/&hl=en&as_sdt=0,11&output=viewport&pg=55. — Дата доступа : 10.09.2019.
2. Efficient Maximum Clique Computation over Large Sparse Graphs // [Электронный ресурс]. — 2019. Режим доступа : https://dl.acm.org/citation.cfm?id=3330986. — Дата доступа : 15.11.2019.
3. Why is maximum clique often easy in practice? // [Электронный ресурс]. — 2019. Режим доступа : http://www.optimization-online.org/DB_FILE/2018/07/6710.pdf. — Дата доступа : 20.11.2019.
4. Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time // [Электронный ресурс]. — 2019. Режим доступа : https://arxiv.org/abs/1006.5440. — Дата доступа : 20.10.2019.
5. Clustering Motifs via Maximal Cliques with Parallel Computing Design // [Электронный ресурс]. — 2019. Режим доступа : https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4972426/. — Дата доступа : 21.10.2019.
6. Barker, E. J.; Buttar, D.; Cosgrove, D. A.; Gardiner, E. J.; Kitts, P.; Willett, P.; Gillet, V. J. Scaffold hopping using clique detection applied to reduced graphs. — J. Chem. Inf. Model. 2006, 46, 503-511. — Дата доступа: 09.01.2021.
7. Butenko, S.; Wilhelm, W. E. Clique-detection models in computational biochemistry and genomics. Eur. J. Oper. Res. 2006, 173, 1-17. — Дата доступа: 09.01.2021.

   6. Artymiuk, P. J.; Poirrette, A. R.; Grindley, H. M.; Rice, D. W.; Willett, P. A graph- theoretic approach to the identification of three-dimensional patterns of amino acid side- chains in protein structures. J. Mol. Biol. 1994, 243, 327-344. — Дата доступа: 09.01.2021.
8. Artymiuk, P. J.; Poirrette, A. R.; Rice, D. W.; Willett, P. The use of graph theoretical methods for the comparison of the structures of biological macromolecules. In Molecular Similiarity II, Sen, K. D., Ed. Springer: Berlin Heidelberg, 1995; pp 73-103. — Дата доступа: 19.01.2021.
9. Konc, J.; Janezic, D. ProBiS: a web server for detection of structurally similar protein binding sites. Nucleic Acids Res. 2010, 38, W436-W440. — Дата доступа: 29.01.2021.

10. Irwin, J. J.; Shoichet, B. K. ZINC-a free database of commercially available compounds for virtual screening. J. Chem. Inf. Model. 2005, 45, 177-182. — Дата доступа: 01.02.2021.
11. Raymond, J. W.; Gardiner, E. J.; Willett, P. Rascal: Calculation of graph similarity using maximum common edge subgraphs. Comput. J. 2002, 45, 631-644. — Дата доступа: 09.02.2021.
12. Мотив (молекулярная биология) // [Электронный ресурс]. — 2019. Режим доступа : https://en.wikipedia.org/wiki/Sequence_motif. — Дата доступа : 03.09.2019.
13. Перез-Чернов А.Х. Специальные структуры данных для задач на графах, связанных с понятием клики или с модульными декомпозициями / А.Х Перез-Чернов, С.В. Суздаль — Минск: Вестник Белорусского государственного университета, 2007. — с.103-108.
14. Distributed graph decomposition algorithms on Apache Spark // [Электронный ресурс]. — 2019. Режим доступа: http://dmgroup.cs.iupui.edu/files/student_thesis/AritraThesis.pdf. — Дата доступа: 11.11.2019.
15. Finding the maximum clique in massive graphs // [Электронный ресурс]. — 2019. Режим доступа : https://dl.acm.org/citation.cfm?id=3137660. — Дата доступа: 11.11.2019.
16. Claude Berge. Graphs And Hypergraphs. — Amsterdam, 1973. — 528 c.
17. Naik R.N. Intersection graph of k-uniform linear hypergraphs / Naik R.N., Rao S.B., Shrikhande S.S., Singhi N.M— Ann. Discrete Math, 1980. — c. 275-279.
18. Matjaž Depolli, Roman Trobec, Janez Konc. Exact Parallel Maximum Clique Algorithm for General and Protein Graphs. — 2013. Режим доступа: http://pubs.acs.org/doi/abs/10.1021/ci4002525. — Дата доступа 05.01.2021.
19. Amr Elmasry, Ayman Khalafallah, Moustafa Meshry. A Scalable Maximum-Clique Algorithm Using Apache Spark. — 2015. Режим доступа: https/www.computer.org/csdl/pds/api/csdl/proceedings/download-article/12OmNyaGeJp/pdf. — Дата доступа: 05.01.2021.
20. Pablo San Segundo, Diego Rodríguez-Losada, Miguel Hernando. An improved bit parallel exact maximum clique algorithm. — 2011. Режим доступа: https://publication/235696204_An_improved_bit_parallel_exact_maximum_clique_algorithm. — Дата доступа: 06.01.2021.
21. Etsuji Tomita. A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments. — 2013. Режим доступа: https://www.researchgate.net/publication/270441228. — Дата доступа: 06.01.2021.

22. Janez Konc. An improved branch and bound algorithm for the maximum clique problem. — 2007. Режим доступа: https://www.researchgate.net/publication/266859824. — Дата доступа 07.01.2021.
23. Pablo San Segundo, Diego Rodrıguez-Losada, Agustin Jimenez. An exact bit-parallel algorithm for the maximum clique problem. — 2010. Режим доступа: https://www.researchgate.net/publication/246239829. — Дата доступа: 07.01.2021.
24. Skums P.: Discovery of binding motifs and epitopes, 2019. . — Дата доступа: 01.09.2019.

ПРИЛОЖЕНИЕ А

Исходный код алгоритма

```
val N = c.getConfig.getInteger("top_N", 1)
val graph = c.getConfig.getMainGraph[MainGraph[_, _]]()

val resize = () => {
                Refrigerator.neigh_sizes      =
KClistEnumerator.neigboursSizes
                        Refrigerator.size         =
Refrigerator.neigh_sizes.get(Refrigerator.neigh_sizes.size
- Refrigerator.idx) + 1

  KClistEnumerator.size = Refrigerator.size
       logWarning(s"Refrigerator  size:  $
{Refrigerator.size.toString};")
}

KClistEnumerator.getColors(graph)

var first = true
val repeatOrClean = () => {
  var continue = true
  if (Refrigerator.result.size == N) {
    continue = false
  } else {
     //well, there are no cliques, reduce clique size and
try again
    Refrigerator.inc()
    resize()
  }

  if (Refrigerator.size < 6 && !first) {
    continue = false
  }

  iter.clearDag()
  iter.resetCursor()

  first = false
  continue
}
```

```
val execEngine = c.getExecutionEngine.asInstanceOf[SparkFromScratchEngine[S]]
var currComp = c.nextComputation()
while (currComp != null) {
  currComp.setExecutionEngine(execEngine)
  currComp.init(config)
  currComp.initAggregations(config)
  currComp = currComp.nextComputation
}
lastStepConsumer = new LastStepConsumer[S]()
var ret_main : ComputationResults[S] = null

while (repeatOrClean()) {

  val start = System.currentTimeMillis

  val ret = processCompute(iter, c)
      logWarning(s"processCompute time: ${(System.currentTimeMillis - start) / 1000.0}s")

  val computationTree = new ComputationTree[S](c, null)

  for (r <- ret.getResults) {
        computationTree.adopt(new ComputationTree[S](computationTree, c.nextComputation(), r))
  }

  var result = computationTree
  var done = ret.size() == 0
  var repeat = false

  val addClique = (s : S) => {
        Refrigerator.result = s.getVertices :: Refrigerator.result
    if (Refrigerator.result.size == N) {
      done = true
    } else {
      repeat = false
      logWarning("FOUND!")
      logWarning(s.getVertices.toString)
      val start = System.currentTimeMillis
      graph.removeCliques(List(s.getVertices))
              logWarning(s"removeCliques time: ${(System.currentTimeMillis - start) / 1000.0}s;")
```

```
      KClistEnumerator.dropColors()
      KClistEnumerator.getColors(graph)

      Refrigerator.idx = 1
      resize()
    }
  }

  val repeatLoop = () => {
    if (done) {
      false
    } else if (result != null && !repeat) {
      val (r, d) = next_children(result)
      result = r
      done = d
      !done
    } else {
      true
    }
  }

  while (repeatLoop()) {
    val start0 = System.currentTimeMillis

    var subgraph : S = result.head.subgraph
          var saved_iter : SubgraphEnumerator[S] =
result.head.enumerator

    if (!done && !repeat) {
                if (result.head.getResultType ==
ResultType.SERIALIZED) {
        //Ok, we have serialized iter and sub
                          val (e, s, ser_time) =
read_iter(result.head.serializedFileIter,
result.head.serializedFileSub, c)
        saved_iter = e
        subgraph = s
        KClistEnumerator.loads += 1
            logWarning(s"deser iter: ${result.id}; size: $
{s.getVertices.size}; dag size: ${e.getDag.size}; deser
time: ${ser_time / 1000.0}s; ")
                } else if (result.head.getResultType ==
ResultType.VERTEX) {
        //Ok, we have only vertex, need to extend
        iter.maybeRemoveLastWord()
```

```
              val (next_iter, extend_time) = extend(iter,
result.head.vertex)
        val ser = System.currentTimeMillis
        val (e, s) = copyIter(next_iter, iter.getSubgraph)
        val ser_time = System.currentTimeMillis - ser
        val orphan = new ComputationResult[S](e, s)
        //result.setHead(new ComputationResult[S](e, s))
        saved_iter = e
        subgraph = s

                logWarning(s"extend ${result.head.vertex};
extend_time: ${extend_time / 1000.0}s; copy iter: $
{ser_time / 1000.0}s; id: ${result.id}")
                 result = new ComputationTree[S](result,
iter.getComputation.nextComputation(), orphan)
              } else if (result.head.getResultType ==
ResultType.SUBGRAPH) {
        //Ok, we have subgraph, rebuild iter
        val vertices = ListBuffer.empty[Int]
        vertices += result.head.vertex
        var p = result.parent
          while (p.head != null && p.head.getResultType !=
ResultType.REGULAR) {
          vertices += p.head.vertex
          p = p.parent
        }
        subgraph = if (p.head != null) {
                          SparkConfiguration.deserialize[S]
(SparkConfiguration.serialize(p.head.subgraph))
        } else {
                          SparkConfiguration.deserialize[S]
(SparkConfiguration.serialize(iter.getSubgraph))
        }
                    vertices.reverseIterator.foreach(v =>
subgraph.addWord(v))

        val (e, time) = sub2iter(c, subgraph)
        saved_iter = e
        KClistEnumerator.rebuilds += 1
           logWarning(s"rebuild subgraph and iter; size: $
{subgraph.getVertices.size}; dag size: ${e.getDag.size};
rebuild time: ${time / 1000.0}s; id: ${result.id}")
      }
    }
```

```
    if (subgraph.getVertices.size() == Refrigerator.size) {
      addClique(subgraph)
    } else {
      if (writePath) {
        logWarning(subgraph.toOutputString + " -- dag size:
" + saved_iter.getDag.size().toString)
      }

      val nextComp = result.nextComputation
      nextComp.setSubgraphEnumerator(saved_iter)
      subgraph.nextExtensionLevel()
      val results = nextComp.compute(subgraph).getResults
      subgraph.previousExtensionLevel()

      if (results.length == 1) {
        // so here is the logic:
             // on the previous iteration we checked all
candidates, if the results.length == 1, we have only one
candidate
          // so, starting from this point we only need to
find first candidate from next candidates
        // because the number may only falling

        repeat = true

        //check if we already have a clique
         //the first one should always be with subgraph and
enumerator
        val s = results.get(0).subgraph
        val dag = results.get(0).enumerator.getDag

        if (KClistEnumerator.isClique(dag)) {
              logWarning(s"KClistEnumerator.isClique for $
{s.getVertices}!")

          for (i <- dag.keySet()) {
            s.addWord(i)
          }
          addClique(s)
          repeat = false
        }

        if (repeat) {
          result.setHead(results.get(0))
          result.updateId()
```

```
          result.updateLevel()
        }
      } else {
        if (writePath && results.isEmpty) {
          logWarning("|--> X")
        }

        //result.head.reset()

        for (orphan <- results) {
                    val c = new ComputationTree[S](result,
nextComp.nextComputation(), orphan)

          result.adopt(c)
        }
        repeat = false
      }

      val stepTime = System.currentTimeMillis - start0
      if (true || result.level % 100 == 0) {
              logWarning(s"handling ${result.id}, level ${
result.level}, " +
          s"time: ${stepTime / 1000.0}s; " +
           s"extend_time_all: ${extend_time_all / 1000.0}s;
" +
                s"ser_time_all: ${ser_time_all / 1000.0}s;
colors: ${colors_all / 1000.0}s;")

             //colors_vertices_all: ${colors_vertices_all};
colors_neighbours_all: ${colors_neighbours_all};")
      }
      extend_time_all = 0
      ser_time_all = 0
      colors_all = 0
      colors_vertices_all = 0L
      colors_neighbours_all = 0L
    }
  }

  val elapsed = System.currentTimeMillis - start
     logInfo(s"WorkStealingMode internal=${
config.internalWsEnabled()}" +
    s" external=${config.externalWsEnabled()}")
  logInfo(s"InitialComputation step=${c.getStep}" +
    s" partitionId=${c.getPartitionId} took ${elapsed} ms")
```

```
  ret_main = ret
}
graph.closeMap()
ret_main
```

Исходный код поиска следующего подграфа

```
private def next_children(current: ComputationTree[S]):
(ComputationTree[S], Boolean) = {
  var result = current
  var done = false

  //visit first available child if possible
  val child = result.visit()
  if (child == null) {
    //well, we have no children, go back to parent
    result.killChildren()
    result.head.reset()

    var back = true
    while (result.hasParent && back) {
      result = result.parent
      val child0 = result.visit()
      if (child0 != null) {
        result = child0
        back = false
      }
    }
    if (result.parent == null) {
      //oh, this is init parent, we have visited all nodes
      done = true
    }
  } else {
    //we go deeper, result is child
    result = child
  }

  (result, done)
}
```

Исходный код алгоритма ленивого расширения

```
private def hasNextComputation(iter: SubgraphEnumerator[S],
c: Computation[S], nextComp: Computation[S]):
ComputationResults[S] = {
  val graph = c.getConfig.getMainGraph[MainGraph[_, _]]()
  val states = KClistEnumerator.getColors(graph)
  val size = Refrigerator.size - 1

  val result = new ComputationResults[S]
  val data_path = c.getConfig.getString("dump_path", "")
  val getOnlyFirst = iter.isGetFirstCandidate
  var found = false
    var extendNeeded = iter.getSubgraph.getNumVertices >
KClistEnumerator.EXTENDS_THRESHOLD
  var log = ""

  val prefixSize = iter.getSubgraph.getVertices.size()
    val maxPossibleSize = prefixSize + max(0,
iter.getAdditionalSize - 1)

  while (!(found && getOnlyFirst) && iter.hasNext) {
    val u = iter.nextElem()

    val (uniqColors, elapsed) = FractalSparkRunner.time {
      val dag = iter.getDag

      val neigh_colors = ListBuffer.empty[Int]
      neigh_colors += states(u)

      if (!dag.containsKey(u)) {
        val neighbours = graph.getVertexNeighbours(u)
        val cursor = neighbours.cursor()
        while (cursor.moveNext()) {
          neigh_colors += states(cursor.elem())
        }
      } else {
        val dagNeighbours = dag.get(u)
        val cursor = dagNeighbours.cursor()
        while (cursor.moveNext()) {
          neigh_colors += states(cursor.elem())
        }
      }
      colors_neighbours_all += neigh_colors.size
      //k-clique contains k colors
      neigh_colors.distinct.size
    }
```

```
    colors_vertices_all += 1
    colors_all += elapsed

    val isFirstComputation = maxPossibleSize == 0
      val isSizeOk = !(isFirstComputation && uniqColors <
size || !isFirstComputation && maxPossibleSize < size)

    if (isSizeOk && uniqColors + prefixSize > size) {
      found = true

      if (prefixSize == 0) {
        extendNeeded = false
        KClistEnumerator.graphCounter += 1
      }

      if (extendNeeded) {
        val (next_iter, extend_time) = extend(iter, u)
        if (result.size() > 0) {
          var ser_time: Long = 0
           val (iterName, subName, s) = save_iter(next_iter,
iter.getSubgraph, data_path)
          ser_time = s
          result.add(iterName, subName)
          KClistEnumerator.dumps += 1

          ser_time_all += ser_time

          if (false && writePath) {
             val str = if (getOnlyFirst) "only first " else
""
            logWarning("|--> " + str + u.toString)
          }

                          log += s"\n$u: dump to file ${
KClistEnumerator.dumps.toString} dag size: ${iter.getDag.size};
sub size: ${iter.getSubgraph.getVertices.size()}" +
                s"; extend_time: ${extend_time / 1000.0}s;
ser_time: ${ser_time / 1000.0}s; get colors: ${elapsed /
1000.0}s;"
        } else {
          var ci = ""
          val copy = System.currentTimeMillis
          if (getOnlyFirst) {
            ci = "just add; "
```

```
            result.add(next_iter, iter.getSubgraph)
          } else {
              val (iterNew, subNew) = copyIter(next_iter,
iter.getSubgraph)
            ci = "copy iter; "
            result.add(iterNew, subNew)
          }
          val copy_time = System.currentTimeMillis - copy

           log += s"\n$u: $ci dag size: ${iter.getDag.size};
sub size: ${iter.getSubgraph.getVertices.size()}" +
                s"; extend_time: ${extend_time / 1000.0}s;
copy_time: ${copy_time / 1000.0}s; get colors: ${elapsed /
1000.0}s;"
        }
      } else {
        if (writePath) {
          log += s"\n$u: add vertex"
        }
        result.add(u, prefixSize == 0)
      }
    }
  }

   if (prefixSize != 0 && !extendNeeded && result.size() >
0) {
      //let's extend, because we have only one appropriate
vertex
         val subgraph = SparkConfiguration.deserialize[S]
(SparkConfiguration.serialize(iter.getSubgraph))
    iter.set(c, subgraph)
          val (next_iter, extend_time) = extend(iter,
result.get(0).vertex)
    log += s"\nextend first time: ${extend_time / 1000.0}s"
    result.get(0).reset()
    if (result.size() == 1) {
      next_iter.shouldRemoveLastWord = false
      next_iter.setGetFirstCandidate(true)
    }
    result.get(0).enumerator = next_iter
    result.get(0).subgraph = subgraph
    result.get(0).setResultType(ResultType.REGULAR)
  }
  iter.extend = true
```

```
    if (result.size() > 0) logWarning(s"Length: ${result.size()}")
  if (log != "") logWarning(log)
  result
}
```

ПРИЛОЖЕНИЕ В

Исходный код дерева расширений

```
public class ComputationTree<S extends Subgraph> {
    Computation<S> nextComputation;

    public void setHead(ComputationResult<S> head) {
        this.head = head;
    }

    ComputationResult<S> head;
    ComputationTree<S> parent;
    private List<ComputationTree<S>> children;
    private int firstNotVisited = 0;
    public int level;
    private static int idCounter = 0;
    public int id;

          ComputationTree(ComputationTree<S> parent,
Computation<S> nextComputation, ComputationResult<S> head)
{
        this.nextComputation = nextComputation;
        this.head = head;
        this.children = new ArrayList<>();
        this.parent = parent;
        if (parent == null) {
            this.level = 0;
        } else {
            this.level = parent.level + 1;
        }

        updateId();
    }

          ComputationTree(Computation<S> nextComputation,
ComputationResult<S> head) {
        this(null, nextComputation, head);
    }

    public ComputationTree<S> visit() {
        if (hasUnvisited()) {
                                ComputationTree<S> child =
children.get(firstNotVisited);
```

```
            //children.set(firstNotVisited, null);
            firstNotVisited++;
            return child;
        } else {
            return null;
        }
    }

    public boolean hasUnvisited() {
        return firstNotVisited < this.children.size();
    }

    public void adopt(ComputationTree<S> orphan) {
        children.add(orphan);
    }

    public boolean hasParent() {
        return parent != null;
    }

    public void killChildren() {
        children = new ArrayList<>();
    }

    public void updateId() {
        id = idCounter++;
    }

    public void updateLevel() {
        level++;
    }
}
```

Исходный код ComputationResult

```
enum ResultType {
    VERTEX,
    SUBGRAPH,
    SERIALIZED,
    REGULAR
}

public class ComputationResult<S extends Subgraph> {
    SubgraphEnumerator<S> enumerator;
    S subgraph;
    String serializedFileIter = "";
    String serializedFileSub = "";
    int vertex = -1;

    public void setResultType(ResultType resultType) {
        this.resultType = resultType;
    }

    public ResultType getResultType() {
        return resultType;
    }

    private ResultType resultType;

      ComputationResult(SubgraphEnumerator<S> enumerator, S
subgraph) {
        //System.out.println("ResultType.REGULAR");
        resultType = ResultType.REGULAR;
        this.enumerator = enumerator;
        this.subgraph = subgraph;
    }

    ComputationResult(S subgraph) {
        //System.out.println("ResultType.SUBGRAPH");
        resultType = ResultType.SUBGRAPH;
        this.subgraph = subgraph;
    }

       ComputationResult(String serializedFileIter, String
serializedFileSub) {
        //System.out.println("ResultType.SERIALIZED");
        resultType = ResultType.SERIALIZED;
```

```
        this.serializedFileIter = serializedFileIter;
        this.serializedFileSub = serializedFileSub;
    }

    ComputationResult(int v, boolean first) {
        if (first) {
            //System.out.println("ResultType.VERTEX");
            resultType = ResultType.VERTEX;
            vertex = v;
        } else {
                    //System.out.println("ResultType.SUBGRAPH
ONE");
            resultType = ResultType.SUBGRAPH;
            vertex = v;
        }
    }

    void reset() {
        subgraph = null;
        enumerator = null;
        serializedFileIter = "";
        serializedFileSub = "";
    }

}
```