МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование кафедры)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Системное программирование и компьютерные технологии
(направленность (профиль)/специализация)

БАКАЛАВРСКАЯ РАБОТА

на тему «Оценка производительности алгоритмов поиска минимального остовного дерева»

Студент	Д.Б. Сабиров	
	(И.О. Фамилия)	(личная подпись)
Руководитель	Э.В. Егорова	
	(И.О. Фамилия)	(личная подпись)
Консультанты	Н.В. Андрюхина	
	(И.О. Фамилия)	(личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент, А.В. Очеповский _____
(ученая степень, звание, И.О. Фамилия) (личная подпись)

«_____» _____ 20___ г.

Тольятти 2019

АННОТАЦИЯ

Бакалаврскую работу выполнил студент Тольяттинского государственного университета, Сабиров Данил Борисович. Название работы: «Оценка производительности алгоритмов поиска минимального остовного дерева». Актуальность работы заключается в том, что при решении некой практической задачи требуется найти минимальное остовное дерево (MST) некоторого граф. Однако не всегда однозначно ясно какой из алгоритмов MST следует использовать для достижения максимальной производительности. Для решения этой задачи был произведен анализ, разработка и тестирование последовательных и параллельных алгоритмов поиска минимального остовного дерева.

Объектом исследования является процесс выбора оптимального алгоритма поиска минимального остовного дерева для графа определенного вида.

Предметом исследования являются алгоритмы Крускала, Прима, Борувки.

Цель работы: проанализировать однопоточные и многопоточные алгоритмы поиска минимального остовного дерева и сравнить их производительность на различных графах.

В первой главе рассмотрена задача поиска минимального остовного дерева и алгоритмы решения этой задачи.

Во второй главе произведен анализ параллельных алгоритмов поиска минимального остовного дерева, а также их реализация.

В третьей главе произведено тестирование и анализ полученных результатов.

Бакалаврская работа выполнена на 48 страницах, состоит из введения, трех глав, заключения, списка литературы, состоящего из 21 литературных источников, 35 рисунков и 5 таблиц.

ABSTRACT

This bachelor's graduation work is by the student of Togliatti state university, Sabirov Danil Borisovich. The work's title is "Evaluation of performance for minimal spanning tree searching algorithms".

Nowadays, IT specialists often face a problem of finding a minimal spanning tree for a certain graph in a practical situation. It is usually extremely difficult to choose the best MST algorithm for a specific situation to reach the peak performance. To solve this problem, an analysis, implementation and tests for both single thread and multi thread algorithms for minimal spanning tree searching, were made.

The object of this work is a process of choosing an appropriate minimal spanning tree algorithm for a specific situation.

The subjects of this work are Kruskal, Boruvka and Prim's algorithms.

The goal of the graduation work is to analyze single threaded and multi threaded minimal spanning tree searching algorithms and compare their performances for different graphs.

This work includes three chapters.

The first chapter shows a problem of searching the minimal spanning tree and algorithms for solving this problem.

Second chapter displays parallel minimal spanning tree searching algorithms analysis and their implementations.

The third chapter is focused on testing and analyzing the results.

The graduation thesis is on 48 pages, consist of an introduction, three chapters, conclusion, 21 references, 35 figures and 5 tables.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Минимальным остовным деревом (MST) связного взвешенного графа называется его связный подграф, состоящий из всех вершин исходного графа и некоторых его ребер, причем сумма весов ребер минимально возможная. Нахождение такого дерева встречается в многих прикладных задачах и решение этой задачи позволяет минимизировать затраты на связь компонент принятых за вершины графа.

Примером такой задачи может являться поиск способа соединения городов дорогами так чтобы их общая длинна или стоимость была минимальной. Также задача поиска минимального островного дерева возникает в компьютерных сетях, а именно в протоколе STP, который устраняет петли в сети выбирая при этом лучшие по скорости соединения.

Актуальность данной работы заключается в том, что появляются новые области, где возникает задача поиска MST и сложно определить по получившемуся графу какой алгоритм решит эту задачу оптимально.

Объектом исследования процесс выбора оптимального алгоритма поиска минимального остовного дерева для графа определенного вида.

Предметом исследования являются различные алгоритмы поиска минимального остовного дерева и их модификации.

Цель работы: оценка производительности на различных графах однопоточных и многопоточные алгоритмы поиска минимального остовного дерева.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- проанализировать однопоточные алгоритмы поиска MST;
- реализовать однопоточные алгоритмы MST;
- разработать генераторы тестовых графов;
- составить тесты;
- протестировать однопоточные алгоритмы MST;
- проанализировать многопоточные алгоритмы MST;

- реализовать многопоточные алгоритмы MST;
- протестировать многопоточные алгоритмы MST;
- провести анализ полученных на этапе тестирования данных.

Работа состоит из введения, трех глав, заключения и списка литературы. Введение отражает основные характеристики работы определяет тему работы и ее актуальность, описывает объект и предмет исследования, цели и задачи, которые необходимо рассмотреть в настоящем документе.

В первой главе рассмотрена задача поиска минимального остовного дерева и алгоритмы решения этой задачи.

Во второй главе произведен анализ параллельных алгоритмов поиска минимального остовного дерева, а также их реализация.

В третьей главе произведено тестирование и анализ полученных результатов.

1 ПОСТАНОВКА И АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧИ ПОИСКА МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА

1.1 Постановка задачи поиска минимального остовного дерева

Представим, что есть некая компьютерная сеть, соединяющая различное вычислительное оборудование. Каждое соединение имеет свою скорость, с которой по нему могут передаваться данные. Для корректной работы этой сети требуется отсутствие в ней циклов. Наша задача состоит в том, чтобы каждый вычислительный узел был доступен в сети, и суммарная скорость была максимальной. Такую задачу можно описать как задачу поиска максимального остовного дерева, где узлы — это вычислительное оборудование, а ребра соединение между оборудованием с весом равному скорости соединения. Знак веса ребра мы заменим на противоположный и получим задачу поиска минимального остовного дерева. Далее мы увидим почему мы можем так сделать.

Сформулируем задачу в общем виде. Пусть дан связанный, взвешенный, неориентированный граф $G = (V, E)$, где V – множество вершин, а E – множество ребер этого графа. Для каждого ребра $(u, v) \in E$ задан вес $w(u, v)$, задающий стоимость соединения вершины u и v. Задача состоит в нахождении такого подграфа $T \subseteq G$, который соединяет все вершины и общий вес, вычисляемый по формуле 1 минимален.

$$w\ T\ = \sum_{(u,v)\in T} w(u, v) \qquad 1)$$

Задача поиска минимального остовного дерева и есть задача поиска дерева T. На рисунке 1 изображен некий неориентированный, взвешенный граф G и жирной линией выделены его ребра входящий в минимальный остов.

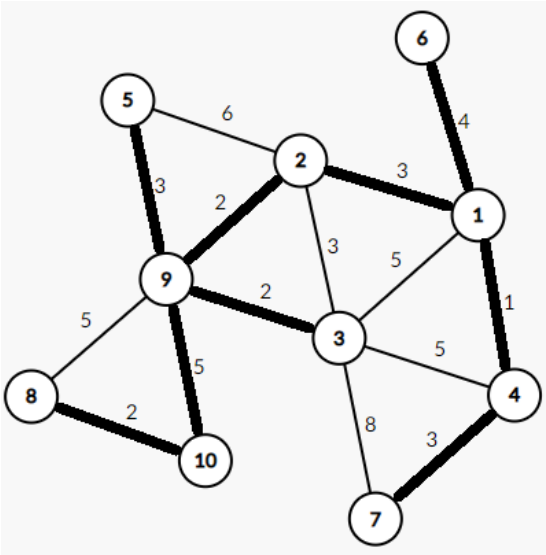

Рисунок 1 – Минимальное остовное дерево связанного графа

Суммарный вес всех ребер, входящих в минимальный остов равен 25. Далее приведем некоторые свойства минимального остова.

Свойства минимального остова:

1. Минимальный остов уникален, если веса всех рёбер различны. В противном случае, может существовать несколько минимальных остовов.
2. Минимальный остов является также и остовом с минимальным произведением весов рёбер.
3. Минимальный остов является также и остовом с минимальным весом самого тяжелого ребра.
4. Остов максимального веса ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова.

Минимальных остовных деревьев может быть несколько в одном графе [1]. Так в предыдущем примере мы могли вместо ребра (8,9) взять ребро (9, 10). Суммарный вес дерева не изменился бы.

1.2 Построение минимального остовного дерева в общем виде

Общая схема работы алгоритмов построения минимального остовного дерева имеет следующий вид. Существует связанный неориентированный граф

$G = (V, E)$ с весовой функцией $w\ u, v$ и мы хотим найти минимальное остовное дерево для этого графа.

Искомый остов получается из подграфа A. Граф A строится постепенно и изначально является пустым. На каждом шаге алгоритм добавляет одно новое ребро. Граф A всегда является подграфом некоторого минимального остова. Добавляемое ребро $e = (u, v)$ выбирается так чтобы $A \cup (u, v)$ являлось подмножеством минимального остовного дерева. Такое ребро называется *безопасным* [5]. На рисунке 2 изображена блок-схема алгоритма построения минимального остовного дерева в общем виде.

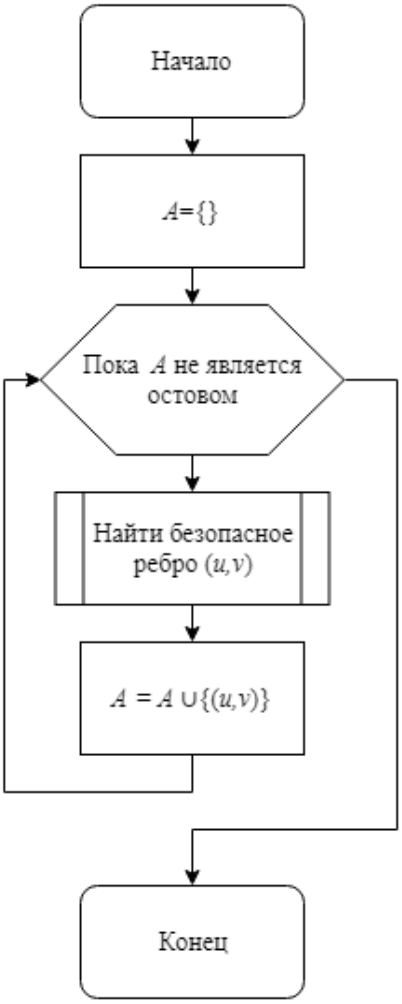

Рисунок 2 – Блок-схема алгоритма построения минимального остовного дерева в общем виде

Далее рассмотрим конкретные алгоритмы поиска минимального остовного дерева.

1.3 Исследование алгоритмов поиска минимального остовного дерева

1.3.1 Исследование алгоритма Крускала и построение его модели

Алгоритм Крускала — эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм описан Джозефом Крускалом в 1956 году.

В начале считаем, что множество ребер в минимальном остове пусто. Затем на каждой итерации выбираем ребро $e(u, v)$ минимального веса $w(e)$. Если при добавлении этого ребра не образовывается цикл, то мы его добавляем иначе ищем следующее ребро. Когда число ребер, включенных в остов, будет равно $n - 1$, где n число вершин в графе, алгоритм завершится. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса [3].

Как видно алгоритм является жадным, поскольку на каждом шаге он добавляет ребро минимального веса.

Опишем модель алгоритма. Пусть дан связный неориентированный граф $G = (V, E)$, где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево T.

1. Изначально множество T полагаем пустым $T = \emptyset$.
2. Формируем множество $Comp = \{C_1, \cdots, C_n\}$, элементами которого являются множества вершин, соответствующих компонентам исходного остовного леса. Каждая такая компонента состоит из единственной вершины.
3. Сортируем множество ребер E исходного графа по возрастанию весов и формируем очередь Q, элементами которой являются ребра G.
4. Если множество $Comp$ содержит более одного элемента и очередь Q не пуста, переходим на шаг 5, иначе на шаг 7.
5. Извлекаем из очереди Q ребро e. Если вершины на концах ребра e лежат в разных компонентах C_i и C_j из $Comp$, то переходим к шагу 6, иначе пропускаем извлеченное ребро и возвращаемся на шаг 4.

6. Объединяем множества вершин C_i и C_j (полагаем $C_{new} = C_i \cup C_j$), удаляем из множества $Comp$ множества C_i и C_j и добавляем в $Comp$ множество C_{new}. Добавляем ребро e в множество T. Возвращаемся на шаг 4.

7. Прекращаем работу. Множество T есть множество ребер минимального остовного дерева.

Приведем пример работы алгоритма. На рисунке 3 изображен граф G.

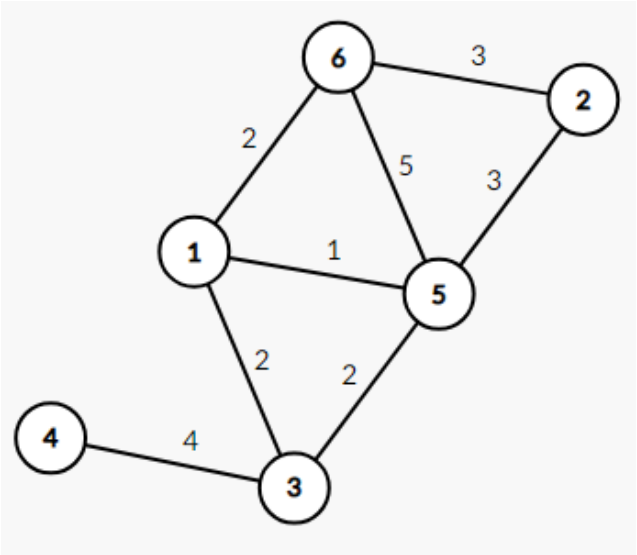

Рисунок 3 – Граф G на котором будет запущен алгоритм Крускала

Выпишем все ребра в порядке возрастания их весов. Результат приведен в таблице 1.

Таблица 1 – Ребра E графа G в порядке возрастания весов

Номер ребра (j)	Ребро (u, v)	Вес ребра (w)
1	(1,5)	1
2	(1,3)	2
3	(1,6)	2
4	(3,5)	2
5	(2,5)	3
6	(2,6)	3
7	(3,4)	4
8	(5,6)	5

На шаге 1 берем ребро (1,5) и добавляем в наш остов (рисунок 4).

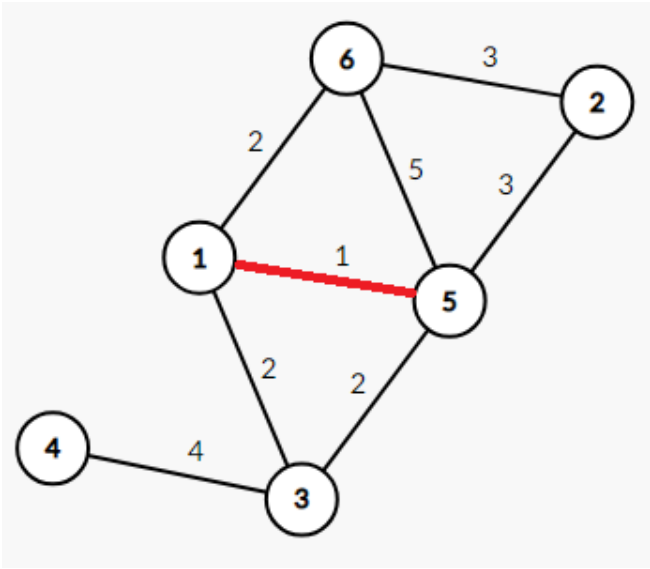

Рисунок 4 – Дерево *T* после первого шага алгоритма Крускала

На шаге 2, 3 берем ребра (1,3), (1,6) и добавляем в наш остов (рисунок 5).

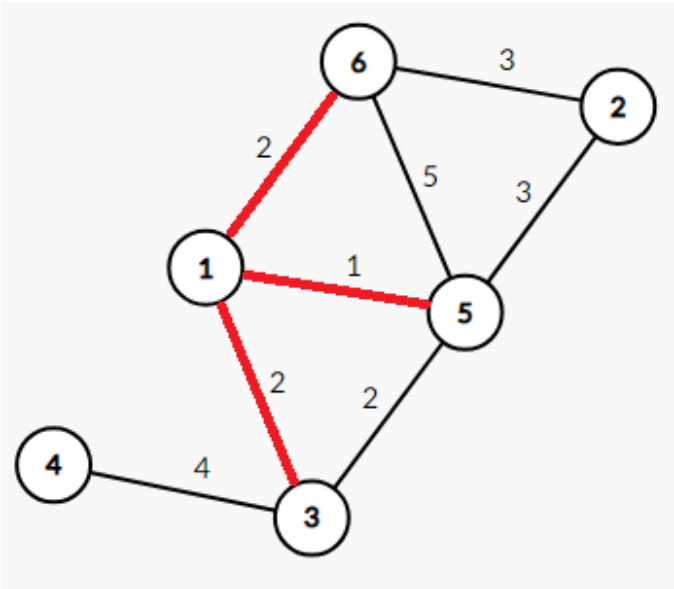

Рисунок 5 – Дерево *T* после третьего шага алгоритма Крускала

На шаге 4 мы не можем взять ребро (3,5) так как тогда образуется цикл из ребер 1, 2 и 4. Следовательно, пропускаем ребро 4 и проверяем ребро (2,5). Текущее ребро не создает противоречий, значит мы включаем его в остов (рисунок 6).

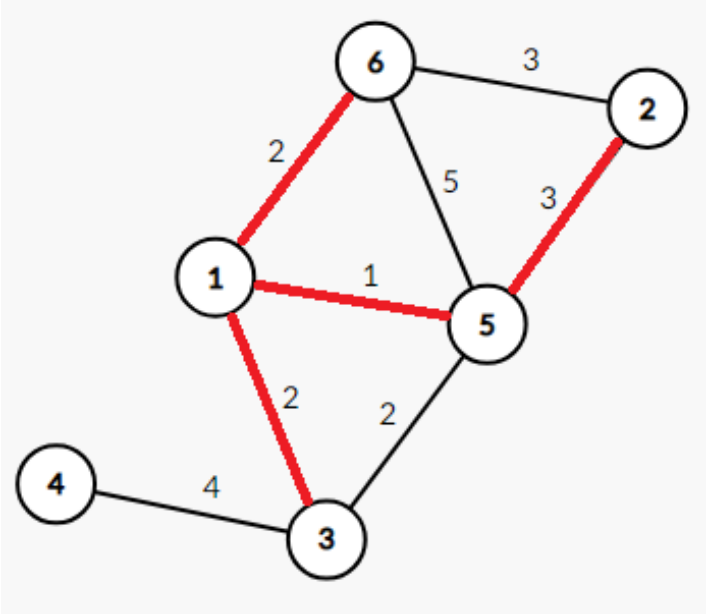

Рисунок 6 – Дерево T после четвертого шага алгоритма Крускала

На шаге 5 мы проверяем ребро $(2,6)$ оно нам не подходит так как будет образован цикл из ребер 1, 3, 5, 6. Ребро (3, 4) мы включаем в минимальный остов (рисунок 7).

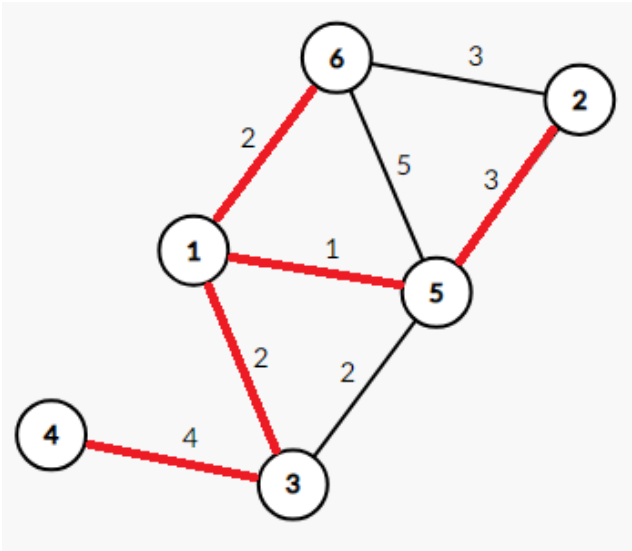

Рисунок 7 – Остов на на пятом шаге алгоритма Крускала

Так как включили $n-1$ ребро, алгоритм можно завершить. Минимальный остов найден.

Асимптотика алгоритма Крускала сильно зависит от используемой сортировки и при использовании быстрой сортировки она равна $O(|E| * log_2(E))$. Далее рассмотрим алгоритм Прима.

1.3.2 Исследование алгоритма Прима и построение его модели

В конце 1950-х годов Эдгар Дейкстра и Прим, работая и публикуя свои результаты независимо друг от друга, предложили следующий алгоритм построения минимального остовного дерева. Алгоритм Прима обладает тем свойством, что ребра в множестве A всегда образует связанное дерево.

Работа алгоритма начинается с выбора любой вершины в графе $G = (V, E)$. Далее из всех ещё не добавленных ребер в множество A и соединяющих дерево A и оставшиеся вершины графа выбирается ребро с наименьшим весом. Выбранное ребро добавляется в множество A. Эта операция повторяется $n - 1$ раз, где n число вершин в графе G. После завершения алгоритма получается остов минимальной стоимости [3, 8].

Опишем модель алгоритма. Пусть дан связный неориентированный граф $G = (V, E)$, где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево $T = (R, A)$.

1. Изначально множество ребер A и множество вершин R полагаем пустыми $A = \emptyset, R = \emptyset$.
2. Выбирается случайная вершина r и добавляется в множество R
3. Из $E - A$ просматриваются все ребра инцидентные вершинам из R
4. Среди таких ребер выбирается ребро $e = (u, v)$, где u вершина из R, а v вершина из $R - A$ значение функции $w(e)$ которой наименьшее
5. Ребро e добавляется к множеству A
6. Вершина v добавляется в множество R
7. Если $A < n - 1$, где n число вершин в графе G то переходим на шаг 3
8. Искомый остов найден

Приведем пример работы алгоритма. На рисунке 8 изображен граф G.

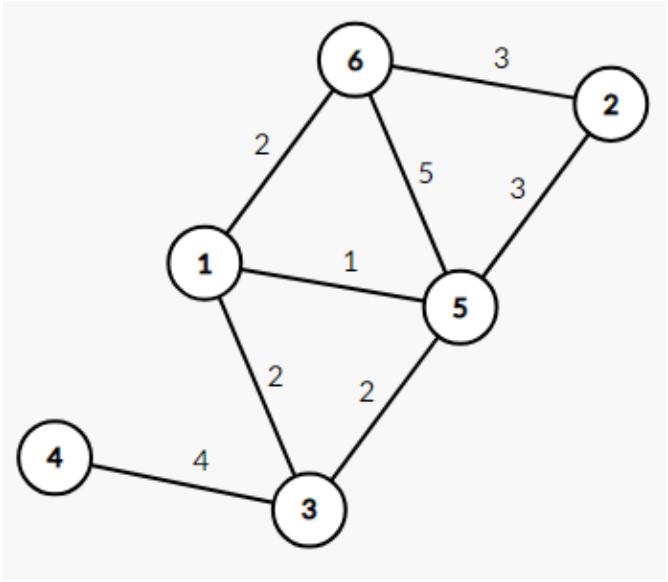

Рисунок 8 – Граф G на котором будет запущен алгоритм Прима

Выберем случайную вершину $r = 6$. Просмотрим все ребра исходящие из r среди таких минимальный вес имеет ребро $(6, 1)$. Добавим его в остов (рисунок 9).

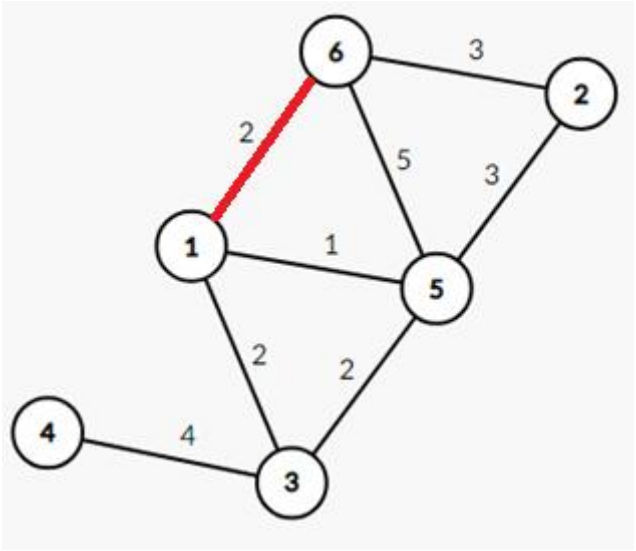

Рисунок 9 – Дерево T после первого шага алгоритма Прима

Теперь у нас есть две вершины 1 и 6. Рассмотрим все инцидентный им ребра. Среди них наименьший вес имеет ребро (1, 5). Добавим его в остов (рисунок 10).

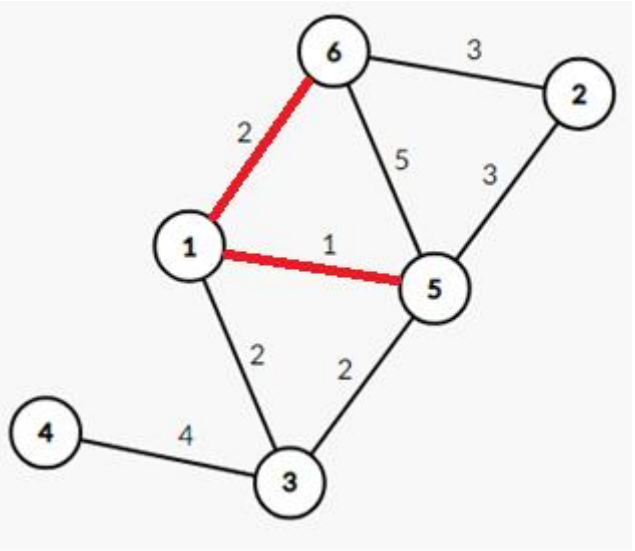

Рисунок 10 – Дерево T после второго шага алгоритма Прима

Аналогично продолжим на каждой итерации выбирать безопасное ребро минимального веса. В итоге получим остов, изображенный на рисунке 11.

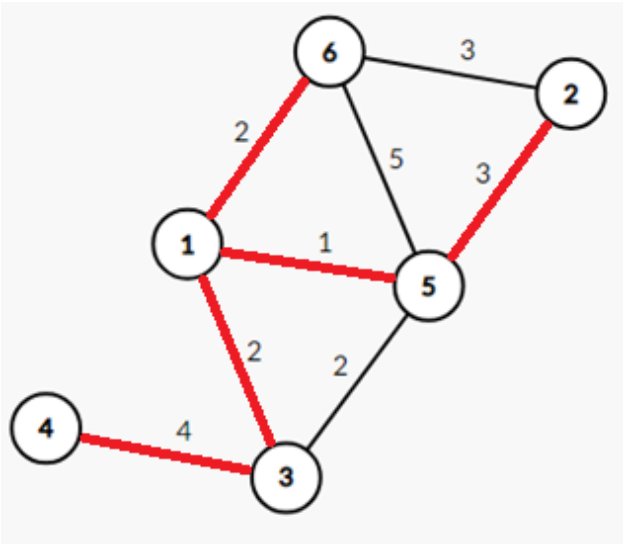

Рисунок 11 – Минимальный остов, полученный алгоритмом Прима

В результате минимальный остов найден.

Существует две реализации данного алгоритма. Первая реализация предназначена для разреженных графов. Она основана на структуре данных красно-чёрное дерево и позволяет находить минимальное ребро за $O\ log_2(V)$ операций, но требует на каждом шаге выполнить пересчет всей структуры за $O(V * log_2(V))$ операций. Асимптотика такой реализации равна $O(E *$

$log_2\ |V|\ )$. Вторая реализация предназначена для полных графов и на каждой итерации выполняет поиск минимального ребра за $O(\ V\ )$ операций. Асимптотика такой реализации равна $O(|V|^2)$.

Перейдем к рассмотрению алгоритма Борувки.

1.3.3 Исследование алгоритма Борувки и построение его модели

Алгоритм Борувки – это алгоритм поиска минимального остовного дерева в взвешенном связанном неориентированном графе. Впервые был опубликован в 1926 году Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии.

Алгоритм Борувки представляет граф как лес поддеревьев MST. На первом этапе каждая вершина принадлежит отдельному дереву. Далее каждое дерево выбирает минимальное ребро соединяющие текущее дерево с другим. Если минимальных ребер несколько выбирается ребро с наименьшим порядковым номером это очень важно для правильной работы алгоритма. На следующем этапе все выбранные ребра добавляются в MST и алгоритм повторяется до тех пор, пока не останется одно дерево. Оставшееся дерево и будет искомым минимальным остовным деревом [5].

Опишем модель алгоритма. Пусть дан связный неориентированный граф $G = (V, E)$ где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево $T = (R, A)$.

1. Изначально имеем множество поддеревьев MST $Comp = \{C_1, C_2, \dots, C_{|E|}\}$, где C_i содержит вершину i.
2. Для каждого поддерева из множества $Comp$ выбирается ребро соединяющие текущую компоненту с другой и имеющие наименьший вес. Если веса равны, то выбираем ребро с наименьшим номером.
3. Компоненты объединяются и из оставшихся компонент формируется новое множество $Comp$.
4. Если $Comp\ > 1$ перейти к шагу 2 иначе к шагу 5.
5. Минимальный остовным деревом будет являть дерево C_1.

Приведем пример работы алгоритма. На рисунке 12 изображен исходный граф G.

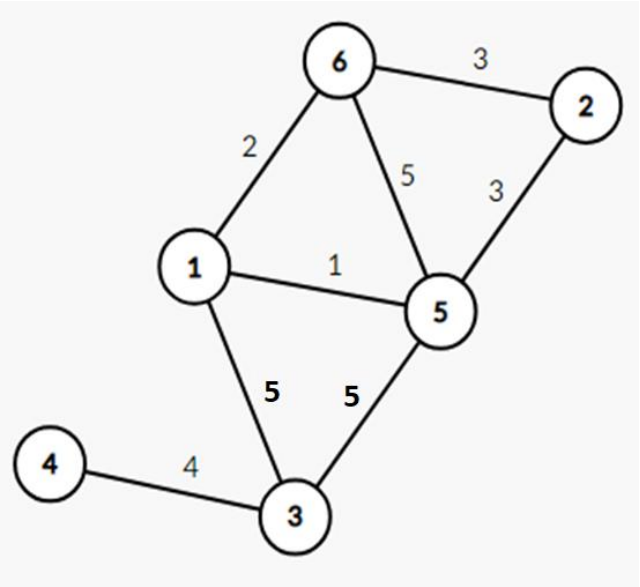

Рисунок 12 - Граф G на котором будет запущен алгоритм Борувки

Представим каждую вершину как поддерево MST. На рисунке 13 изображены поддеревья MST и различные поддеревья обозначены разным цветом.

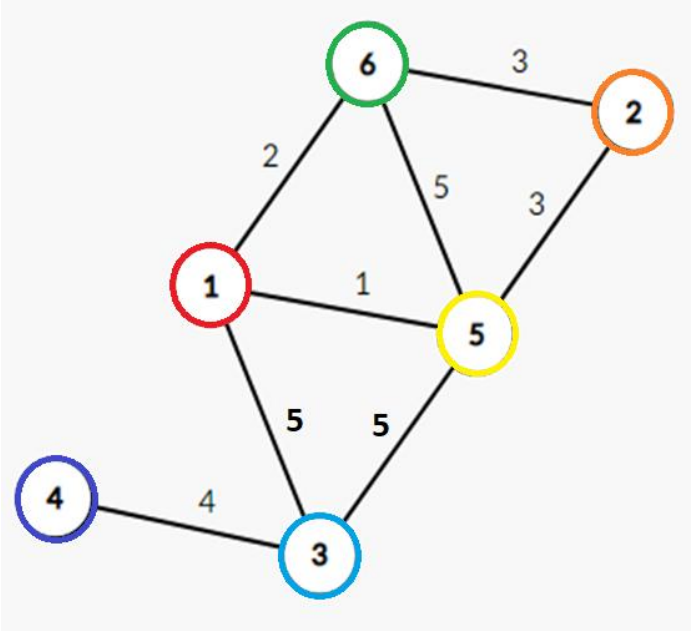

Рисунок 13 – Разбиение вершин по различным поддеревьям

Далее для каждого поддерева выберем минимальное ребро соединяющие текущие поддерево с другим поддеревом (рисунок 14).

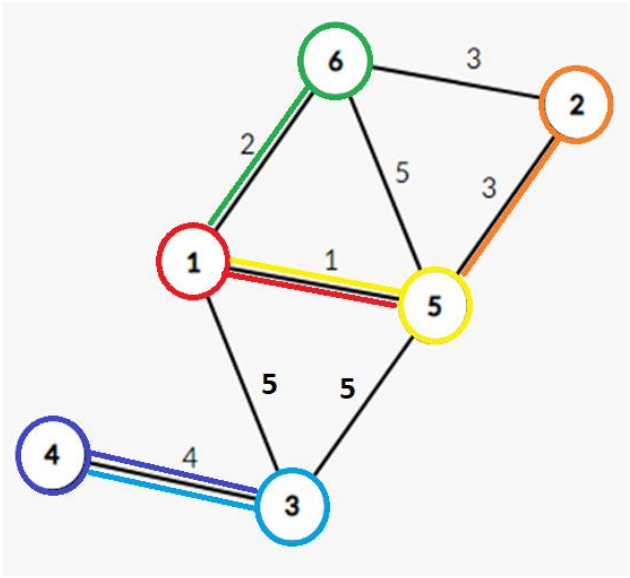

Рисунок 14 – Ребра выбранные каждым поддеревом

Теперь все ребра необходимо добавить в список ребер MST и объединить связанные компоненты (рисунок 15).

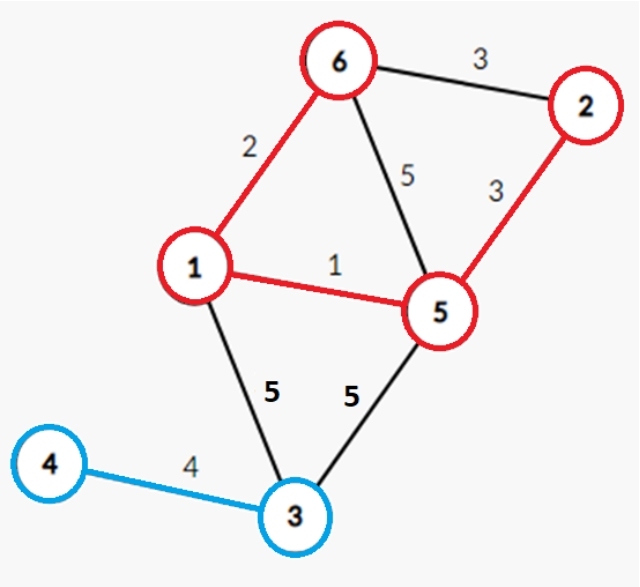

Рисунок 15 – Новые поддеревья после объединения

После одной итерации алгоритма Борувки число поддеревьев сократилось с 6 до 2. Повторим операцию выбора минимальных ребер и получим результат, изображенный на рисунке 16.

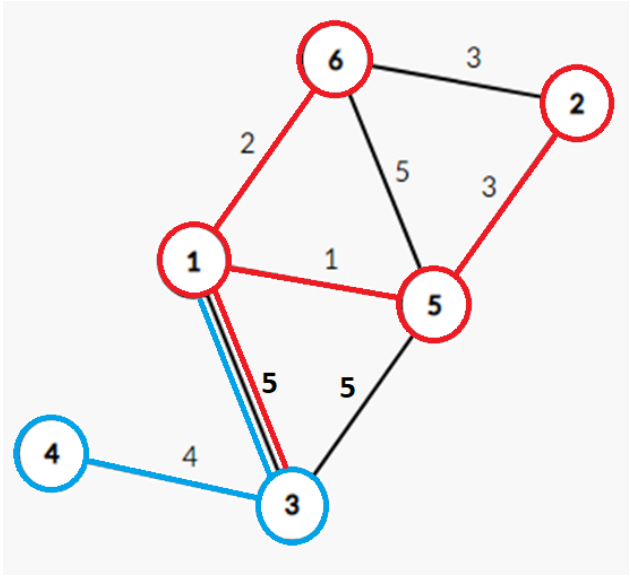

Рисунок 16 – Выбранные ребра после второй итерации алгоритма Борувки

Оставшиеся две компоненты объединяются и получается искомое минимальное остовное дерево (рисунок 17).

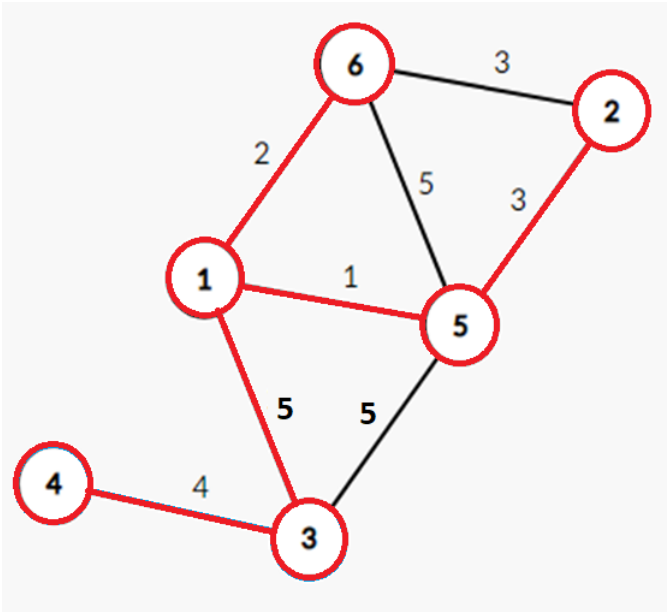

Рисунок 17 – Минимальное остовное дерево, найденное алгоритмом Борувки

Можно заметить, что на каждой итерации алгоритма Борувки число поддеревьев сокращается как минимум вдвое. Следовательно алгоритм в худшем случае выполнит $O(log_2\ V\ )$ итераций. На каждой итерации мы в

худшем случае просмотрим все ребра. Получается итоговая оценка алгоритма Борувки $O(\ E\ *log_2\ \ V\ \ )$ [2].

В результате были описаны модели основных алгоритмов поиска минимального остовного дерева для дальнейшего понимания результатов тестирования алгоритмов и поиска мест возможной организации параллельных вычислений.

Далее перейдем к разработке моделей параллельных версий алгоритмов, описанных выше.

2 ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ПОИСКА МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА

2.1 Подход применяемый к организации параллельных вычислений

Следующие несколько действий которые могут помочь определить эффективные способы организовать параллельные вычисления:

- проанализировать существующие вычислительные схемы и определить места где они могут быть разделены на подзадачи так, что каждая подзадача может быть выполнена независимо от других
- определить для полученного набора подзадач алгоритм информационного взаимодействия
- разработать параллельную вычислительную схему и выполнить разделения подзадач между процессорами

Важно равномерно разделять подзадачи между процессорами так чтобы каждый процессор имел примерно равную вычислительную нагрузку. Такой разделение позволяет максимально эффективно использовать вычислительные ресурсы и минимизировать простаивания процессоров в ожидании, когда какой-то процессор все ещё занят вычислениями. Также при проектировании параллельной вычислительной схемы следует стремиться к минимизации информационных связей между подзадачами [14].

Следует учитывать тот факт, что организация параллельных вычислений может привести к дополнительному времени для разделения задачи и распределении данных, что в свою очередь может привести к дополнительному росту потребляемой памяти. Возможны ситуации, когда из-за слишком большого числа мест где необходимо синхронное взаимодействие параллельная схема может работать дольше последовательной [19].

2.2 Оценка возможности организовать параллельные вычисления в алгоритме Крускала

Алгоритм Крускала является жадным алгоритмом и каждый этап вычислений требует в нем строгой последовательности. Каждое ребро в списке

должно быть обработано после того как будут обработаны все предыдущие ребра. Следовательно, организовать параллельную обработку ребер не представляется возможности, но можно распараллелить предобработку ребер. Так как для работы алгоритма необходимо чтобы ребра были отсортированы по не убыванию их весов. Можно этап сортировки ребер выполнять в нескольких потоках.

Модель параллельного алгоритма Крускала не будет сильно отличаться от его последовательной версии. Отличия будут заключаться в измененном алгоритме сортировки, который может выполнять сортировку данных независимо на разных процессорах [10].

Модель параллельного алгоритма Крускала будет иметь следующий вид. Пусть дан связный неориентированный граф $G = (V, E)$, где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево T.

1. Изначально множество T полагаем пустым $T = \emptyset$.
2. Формируем множество $Comp = \{C_1, \cdots, C_n\}$, элементами которого являются множества вершин, соответствующих компонентам исходного остовного леса. Каждая такая компонента состоит из единственной вершины.
3. Сортируем множество ребер E исходного графа, используя одну из параллельный сортировок, по возрастанию весов и формируем очередь Q, элементами которой являются ребра G.
4. Если множество $Comp$ содержит более одного элемента и очередь Q не пуста, переходим на шаг 5, иначе на шаг 7.
5. Извлекаем из очереди Q ребро e. Если вершины на концах ребра e лежат в разных компонентах C_i и C_j из $Comp$, то переходим к шагу 6, иначе пропускаем извлеченное ребро и возвращаемся на шаг 4.
6. Объединяем множества вершин C_i и C_j (полагаем $C_{new} = C_i \cup C_j$), удаляем из множества Comp множества C_i и C_j и добавляем в $Comp$ множество C_{new}. Добавляем ребро e в множество T. Возвращаемся на шаг 4.

7. Прекращаем работу. Множество T есть множество ребер минимального остовного дерева.

Асимптотическая оценка такого алгоритма будет дополнительно зависеть от числа процессоров, на которых будет выполняться сортировка. Получаем следующую асимптотическую оценку $O(\frac{|E|*log\ \ E}{p} + |E| * a\ \ E\ , |V|\ )$, где p – число процессоров, а a – функция, обратная к функции Аккермана.

2.3 Особенности реализации алгоритма Крускала

Для алгоритма Крускала важно отметить используемую в нем сортировку так, как в нем она является самой затратной частью по потребляемым вычислительным ресурсам. В последовательном и параллельном алгоритме использована сортировка block_indirect_sort из библиотеки boost. Block_indirect_sort – это новый параллельный алгоритм сортировки разработанный Франциско Тапиа. Особенность этого алгоритма заключается в высокой скорости работы и низким потребление памяти. В документации к библиотеке boost приводится сравнение этого алгоритма с параллельными алгоритмами сортировки из библиотеки OpenMP и из библиотеки Intel Threading Building Blocks. Результаты измерений при сортировке массива длиной в 100 000 000 из 64 битных чисел изображены на рисунке 18.

Algorithm	Time (secs)	Memory used
Open MP Parallel Sort	1.1990	1564 MB
Threading Building Blocks (TBB)	1.6411	789 MB
Block Indirect Sort	0.9270	790 MB

Рисунок 18 – Результаты изменения параллельных сортировок

В худшем случае выбранный алгоритм имеет асимптотику $O(n * log_2(n))$, где n число сортируемых элементов. Число потребляемой памяти равно $O(b * |P|)$, где b размер блока, который вычисляется по таблице 2, а P число используемых процессоров [20].

Таблица 2 – Определение размера блока в алгоритме block_inderect_sort

Размер элемента массива в байтах	1 - 15	16 - 31	32 - 63	64 - 127	128 - 255	256 - 511	512 <
Размер блока (b)	4096	2048	1024	768	512	256	128

Ещё одним важным замечанием является то, что в текущей реализации алгоритма Крускала используется структура данных «система непересекающихся множеств» (DSU) для возможности за время $O\ log_2\ n$, где n число вершин в графе производить объединения компонент [17].

2.4 Оценка возможности организовать параллельные вычисления в алгоритме Прима

Алгоритм Прима аналогично, как и алгоритм Крускала требует строгой последовательности выполнения итераций алгоритма. Параллельно вычислять отдельные итерации алгоритма не представляется возможным, но на каждой итерации действия выполняются независимо. Так, когда требуется найти ребро соединяющие вершину из текущего остова с вершиной, ещё не включенной в остов можно чтобы каждый процессор искал такие ребра среди своего множества вершин, а затем на отдельном процессоре объединить полученные результаты [18].

Опишем модель параллельного алгоритма Прима. Пусть дан связный неориентированный граф $G = (V, E)$, где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево $T = (R, A)$.

1. Изначально множество ребер A и множество вершин R полагаем пустыми $A = \emptyset, R = \emptyset$.

2. Присвоим каждому процессору P_i множество вершин $V_i = \{v_{\ i-1\ *k+1} \dots v_{i*k}\}$, где $k = \frac{|V|}{|P|}$.

3. Выбирается случайная вершина r и добавляется в множество R

4. На каждом процессоре P_i выбирается ребро $e_i = v_f, v_s$, $f \neq s$, $v_f, v_s \in V_i, v_f \in R$, $v_s \notin R$ такое, что значение функции $w(e_i)$ минимально

5. Среди найденных ребер выбирается ребро $e = u, v$, $u \in R, v \notin R$ имеющие наименьшее значение функции w

6. Ребро e добавляется к множеству A

7. Вершина v добавляется в множество R

8. Если $A < n - 1$, где n число вершин в графе G то переходим на шаг 3

9. Искомый остов найден

Асимптотическая оценка такого алгоритма будет зависеть от числа процессоров, на которых будет выполняться алгоритм. Всего будет $O(V)$ итераций алгоритма. Каждая итерация будет выполняться за $O(\frac{V}{P})$ операций, где $|P|$ число процессоров. Итоговая асимптотика равна $O(\frac{V * V}{P})$.

2.5 Оценка возможности организовать параллельные вычисления в алгоритме Борувки

Изучая алгоритм Борувки можно заметить его часть, которая легко поддается распараллеливанию. На каждой итерации алгоритма нам необходимо найти для каждой компоненты минимальное ребро соединяющие текущую компоненту с другой. Поиск такого ребра для каждой компоненты выполняется независимо. Следовательно, можно каждому процессору присвоить свой набор компонент и запустить на них поиск оптимальных ребер.

У такого подхода есть несколько недостатков:

- требуется дополнительно отбирать ребра для каждой компоненты;
- возможны случаи, когда нагрузка на процессоры распределена не равномерно.

Эти недостатки можно устранить, изменив подход к поиску ребер. Мы будем выполнять поиск не для каждой компоненты в отдельности, а сразу для всех одновременно. Будем перебирать ребра и для каждой вершины выбирать

минимальное ребро. Тогда нам достаточно присвоить каждому процессору по $\frac{|E|}{|P|}$ ребер и среди них найти безопасные ребра. В таком случае у нас могут возникать ситуации, когда несколько процессоров одновременно для одной вершины вносят новое значение минимального ребра. Решить эту проблеме можно путём использования блокировок или независимо на каждый процессор будет хранить свой список минимальных ребер, а затем производить объединения полученных ребер в одном потоке [10, 11, 12, 15].

Опишем модель алгоритма. Пусть дан связный неориентированный граф $G = (V, E)$ где V – множество вершин, а E – множество ребер. Вес ребра e обозначим $w(e)$, а искомое дерево $T = (R, A)$.

1. Изначально имеем множество поддеревьев MST $Comp = \{C_1, C_2, \dots, C_{|E|}\}$, где C_i содержит вершину i.
2. Присвоим каждому процессору P_i множество ребер $E_i = \{e_{\ i-1\ *k+1} \dots e_{i*k}\}$, где $k = \frac{|E|}{|P|}$.
3. На каждом процессоре P_i вычисляется следующие множество $minE_i = \{minE_{i\,1}, minE_{i\,2} \dots minE_{i\,j} \dots minE_{i\,|V|}\}$, где $minE_{i\,j}$ обозначает номер инцидентного вершине j ребра наименьшего веса или -1 если таких ребер нет.
4. На одном процессоре формируется новое множество ребер $newE = \ e_j \ \min(minE_{1\,j}, minE_{2\,j}, \dots, minE_{P\ \ j})\}$.
5. Все ребра из множества $newE$ добавляются в T.
6. Компоненты объединяются и из оставшихся компонент формируется новое множество $Comp$.
7. Из множества E удаляются все ребра соединяющие вершины в одной компоненте связанности.
8. Если $Comp\ > 1$ перейти к шагу 2 иначе к шагу 9.
9. Минимальный остовным деревом будет являть дерево C_1.

Итого, всего будет выполнено $O(log_2(\ V\ ))$ итераций так как на за одну итерацию число компонент связанности уменьшится как минимум вдвое. На каждой итерации будет выполняться параллельный перебор ребер. Такая

операция выполнится за $O(\frac{E}{P})$. Итоговая асимптотика параллельного алгоритма Борувки равна $O(\frac{E * log_2 \ V}{P})$.

Как видно в результате исследований алгоритмов поиска MST удалось найти места где вычисления могут выполняться независимо и организовать в них параллельные вычисления. Для алгоритма Крускала в реализации была использована, как уже было описано выше, параллельная сортировка из библиотеки boost, а для алгоритма Прима и Борувки использовались возможности библиотеки OpenMP. Библиотека OpenMP были выбрана из-за того, что с помощью неё можно легко организовывать параллельные циклы с автоматическим разделением нагрузки. Также число используемых процессоров определяется автоматически. Это позволяет избежать привычного падения производительности параллельного алгоритма по сравнению с последовательным, когда вычисления выполняются на небольшом количестве данных [19].

3 АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ ПОИСКА МИНИМАЛЬНОГО ОСТОВНОГО ДЕРЕВА

Тестирование проводилось на компьютере оснащённым процессором AMD Ryzen 7 2700X, имеющим 8 процессорных ядер и 16 вычислительных потоков. Управление компьютером осуществляет операционная система на базе ядра Linux. Частота работы процессора была зафиксирована на величине в 2.2 ГГц. Также была написана тестирующая система для генерации тестов и запуска алгоритмов. В основе тестирующей системы лежит утилита time которая запускает программу в отдельном процессе и по завершению её работы выводит время работы программы и максимальное число памяти которое было задействовано [21]. Каждый тест запускался с интервалом в 5 сек чтобы дать возможность операционной системе время освободить оперативную память после предыдущего алгоритма.

3.1 Графы для тестирования алгоритмов

Для тестирования алгоритмов поиска MST необходимо правильно подобрать тестовые графы. Различные виды графов позволяют находить слабые и сильные стороны алгоритмов.

Первым графом для тестирования будет граф-цепочка. Сам по себе граф цепочка является минимальным остовным деревом так, как содержит $n-1$ ребро. Такой граф позволит выделить зависимость времени поиска MST от числа вершин в графе. Так, как в этом тесте сведено к минимуму число операций на поиск минимального ребра этот тест также позволяет посмотреть на минимальное время необходимое алгоритмам на объединения компонент. На рисунке 19 изображен случайны граф-цепочка, состоящий из 10 вершин.

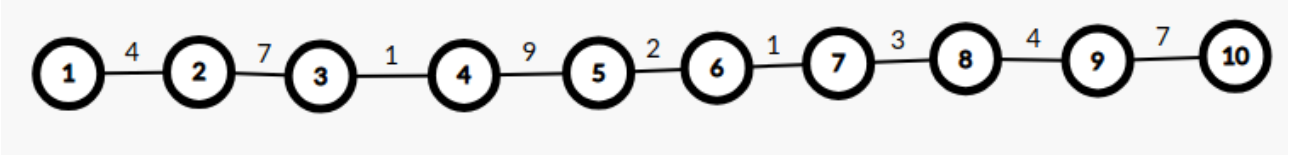

Рисунок 19 – Граф-цепочка

Следующим графом будет полный граф. Полный граф из n вершин содержит максимально возможное число ребер, а именно $\frac{n*(n-1)}{2}$. Полный граф в отличии от графа-цепочки позволит максимально выделить зависимость времени поиска MST от числа ребер. Также он может существенно повлиять на скорость выполнения алгоритмов, которые подобно алгоритму Прима рассматривают все ребра, исходящие из некоторого поддерева MST. На рисунке 20 изображен полый граф из 4 вершин и случайно расставленными на нем весами ребер.

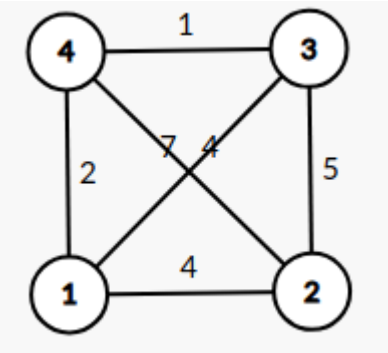

Рисунок 20 – Полный граф

Далее тестирование будет проводиться на разреженном графе. Разреженный граф – это граф где число вершин не сильно превышает число ребер. Такой граф по-прежнему, как и граф-цепочка будет иметь небольшое число ребер, но уже уйдет от формы цепочки и будет принимать случайный вид. При этом число возможных деревьев в таком графе будет больше чем одно. На рисунке 21 изображен случайный разреженный граф из 7 вершин.

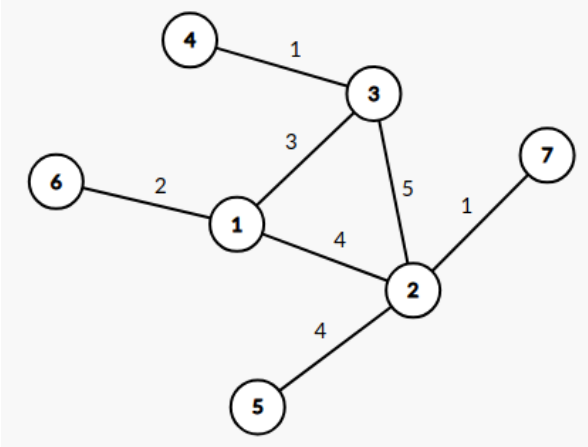

Рисунок 21 – Разреженный граф

Следующий граф зачастую используют как тест, который ломает параллельные алгоритмы. Граф-звезда представляет из себя граф, в котором все ребра исходят из одной вершины. Аналогично графу-цепочки граф-звезда содержит $n-1$ ребро и также является MST. В таком графе степень одной вершины равна $n-1$, а у всех остальных степень вершины равна 1. Такое расположение вершин может сильно повлиять на балансировку параллельного алгоритма так, как одна вершина на одном из процессоров может обрабатываться существенно дольше остальных. На рисунке 22 изображен граф-звезда из 10 вершин.

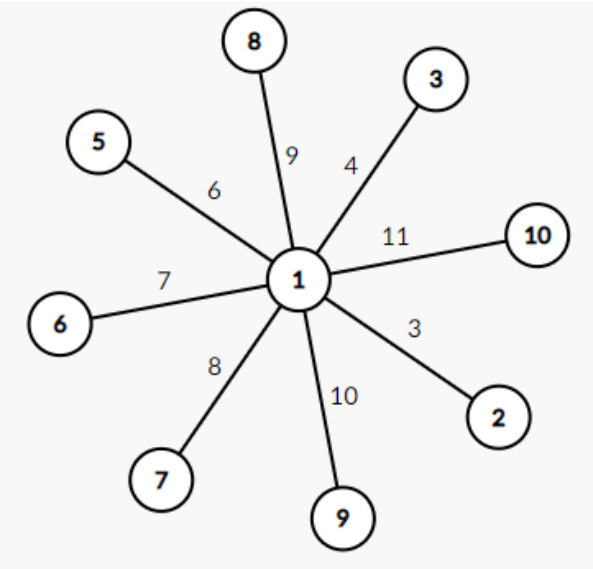

Рисунок 22 – Граф-звезда

Последним графом в тестировании будет случайных граф из n вершин и m ребер. Он поможет получить трехмерный график зависимости времени работы алгоритма от числа вершин и числа ребер в графе. Также случайный граф даёт возможность увидеть полную картину времени работы алгоритма при различной плотности графа. Пример случайного граф из 9 вершин и 14 ребер изображен на рисунке 23.

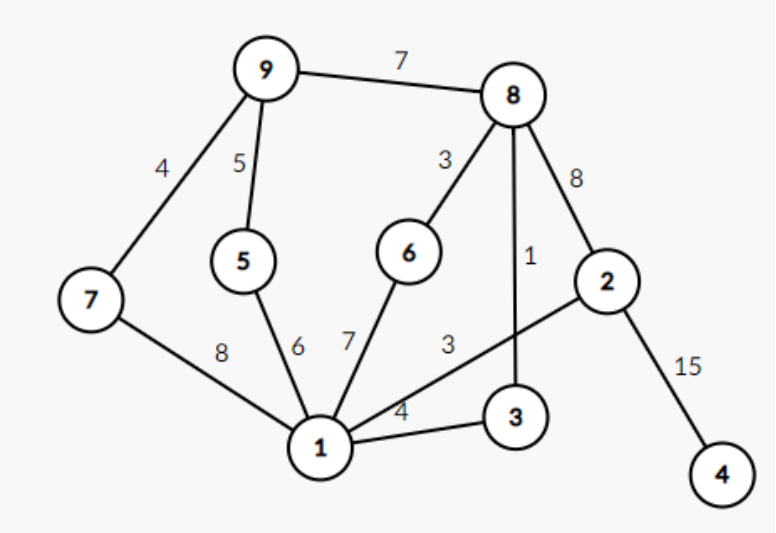

Рисунок 23 – Случайный граф

На практике могут встречаться графы совершенно разной структуры и для разных типов графов определить лучший по требуемым параметрам алгоритм.

Такой набор графов позволит в полной мере произвести сравнение алгоритмов поиска минимального остовного дерева.

3.2 Анализ последовательных алгоритмов поиска минимального остовного дерева

3.2.1 Сравнения последовательных алгоритмов поиска MST на графе цепочка

Для тестирования последовательных алгоритмов поиска минимального остовного дерева на графе цепочка было сгенерировано 30 графов. Число вершин в графе изменялось от 10^6 до $3 * 10^7$.

Ожидалось, что на данном графе лучшее время покажет алгоритм Прима так, как в тестировании использовалась его версия для разряженных графов, но в данном тесте он показал худшее время. Минимальное время работы оказалось у алгоритма Крускала. Следом за ним идет алгоритм Борувки. В целом алгоритмы не сильно различаются во времени работы и алгоритм Крускала в среднем на 12% быстрее алгоритма Борувки и на 16% быстрее алгоритма

Прима. Результаты тестирования времени работы алгоритмов изображены на рисунке 24.

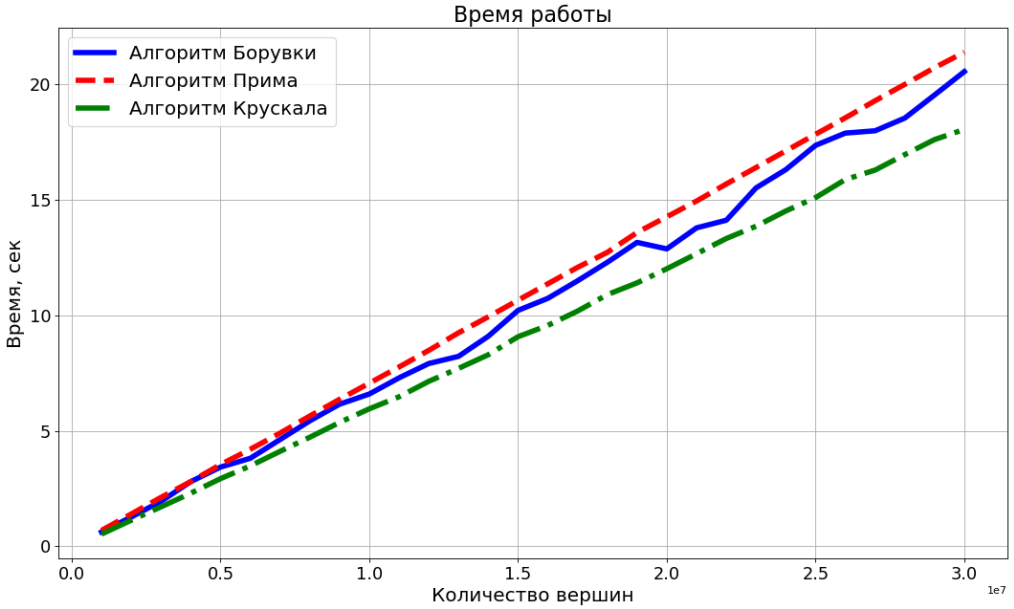

Рисунок 24 – Время работы последовательных алгоритмов поиска MST на графе цепочка

С точки зрения потребляемой памяти между алгоритмами есть существенное различие. Как и ожидалось алгоритм Крускала потребляет наименьшее количество памяти. В сравнении с ним алгоритм Борувки на 31% требует больше оперативной памяти, а алгоритм Прима на 213%. На рисунке 25 изображен график зависимости потребления алгоритмами оперативной памяти от числа вершин в графе цепочка.

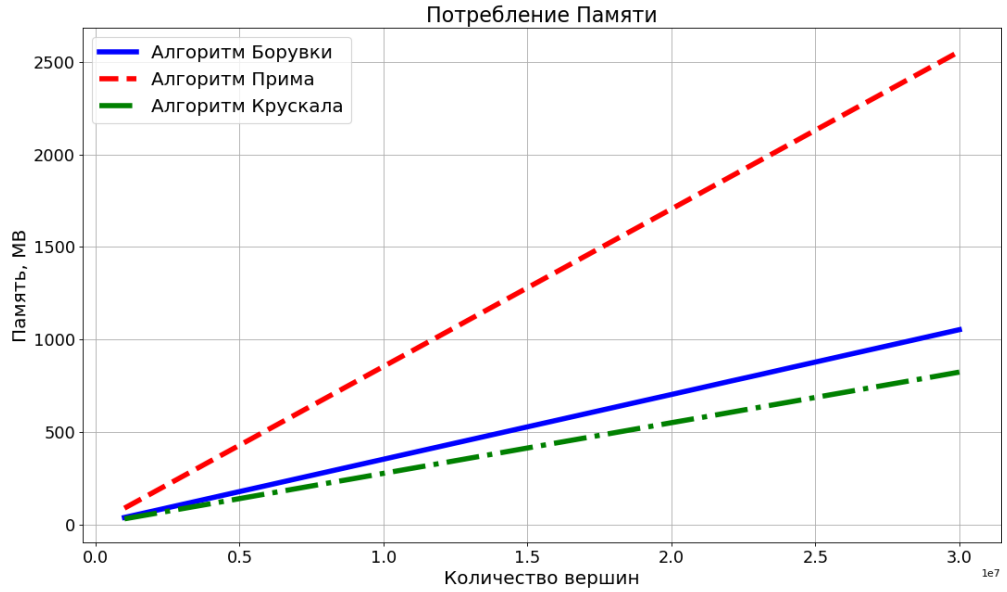

Рисунок 25 – Потребление памяти последовательных алгоритмов поиска MST на графе цепочка

На таком графе алгоритм Прима требует минимальное количество памяти так, как на каждой итерации алгоритма в set добавляется не больше двух ребер.

Основываясь на результатах, полученных в результате тестировании алгоритмов, поиск MST на графе цепочка можно сделать вывод о том, что алгоритмы в целом схожи по скорости добавления в остов новых ребер, но алгоритм Прима отличается значительно более высоким потреблением памяти несмотря на то, что граф-цепочка заданный списком смежности занимает на 25% больше памяти.

3.2.2 Сравнения последовательных алгоритмов поиска MST на полном графе

Для тестирования последовательных алгоритмов поиска минимального остовного дерева на полном графе было сгенерировано 20 графов. Число вершин в графе изменялось от 10^3 до $2 * 10^4$.

В результате тестирования алгоритмов поиска MST алгоритм Прима и алгоритм Борувки показали приблизительно одинаковый результат времени работы. Алгоритм Крускала имеет более высокую динамику роста времени работы при увеличении числа вершин по сравнению с двумя другими алгоритмами. На полном графе с 20000 вершинами алгоритм Крускала оказался на 61% медленнее чем алгоритм Прима и на 53% медленнее чем алгоритм Борувки. Такое большое время работы для алгоритма Крускала обусловлено тем, что алгоритму требуется отсортировать все $\frac{n*(n-1)}{2}$ ребер. На рисунке 26 изображен графики времени работы алгоритмов поиска MST на полном графе.

Не смотря на низкую скорость работы алгоритм Крускала требует наименьшее количество памяти. Так в сравнении с двумя другими алгоритмами алгоритм Крускала потребляет на 32% меньше оперативной памяти. Алгоритм Прима и алгоритм Борувки демонстрируют схожую между собой динамику роста потребляемой памяти по отношению к числу вершин.

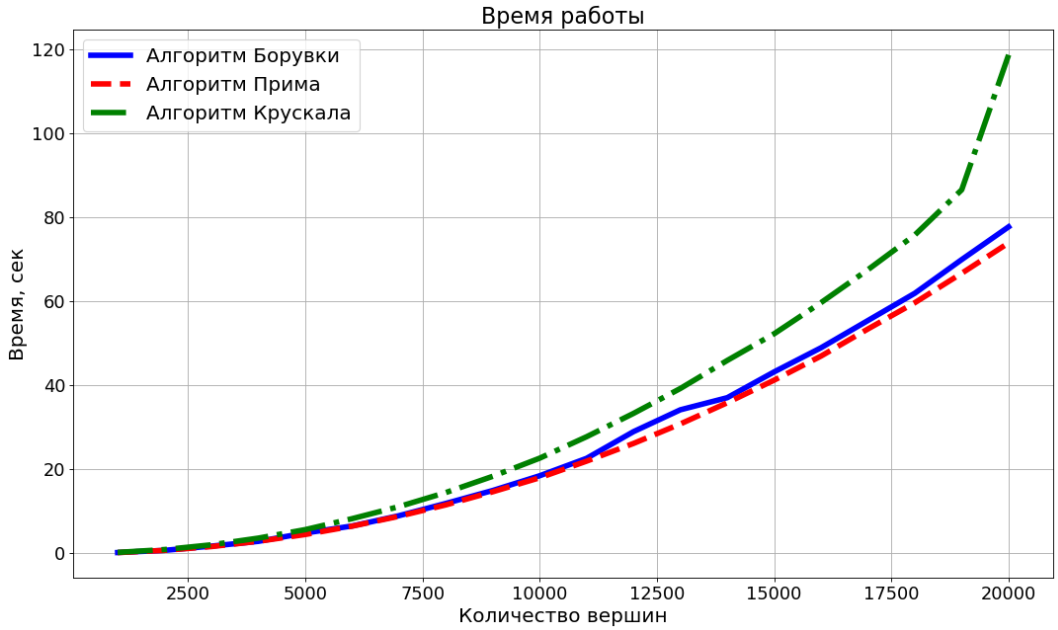

Рисунок 26 – Время работы последовательных алгоритмов поиска MST на полном графе

На рисунке 27 представлены результаты алгоритмов по числу потребляемой памяти на полном графе.

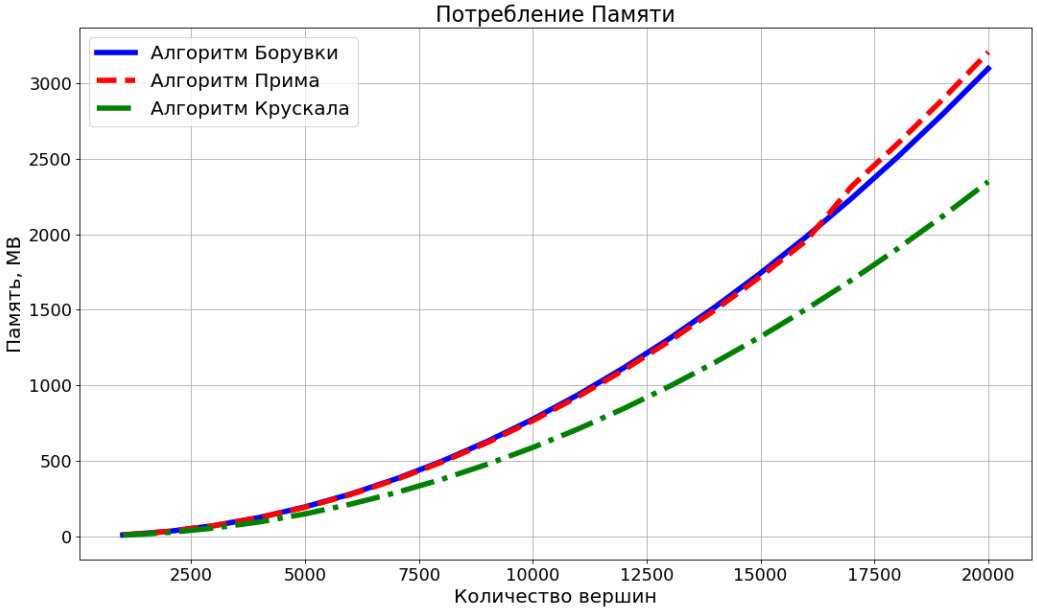

Рисунок 27 - Потребление памяти последовательных алгоритмов поиска MST на полном графе

Исходя из полученных результатов можно сделать вывод, что на полных графах или на графах близких к полным алгоритм Прима и алгоритм Борувки

имеют схожие показатели как по времени, так и по потребляемой памяти. Следовательно, выбор между ними будет основываться на удобном виде представлении графа. Аналогично если требуется минимизировать потребление памяти необходимо выбирать алгоритм Крускала.

3.2.3 Сравнения последовательных алгоритмов поиска MST на разреженном графе

Для тестирования последовательных алгоритмов поиска минимального остовного дерева на разреженном графе было сгенерировано 30 графов. Число вершин в графе изменялось от 10^6 до $3 * 10^7$.

На разреженном графе самым быстрым оказался алгоритм Крускала. Он на 154% быстрее алгоритма Борувки и 264% быстрее алгоритма Прима. Ожидалось, что на разреженном графе результаты будут аналогичны, что и на графе цепочка. Такое отличие можно объяснить тем, что алгоритму Прима приходится поддерживать set большего размера, а алгоритму Борувки необходимо перебрать большее число ребер на каждой итерации. На рисунке 28 представлены результаты тестирования времени работы последовательных алгоритмов поиска MST на разреженном графе.

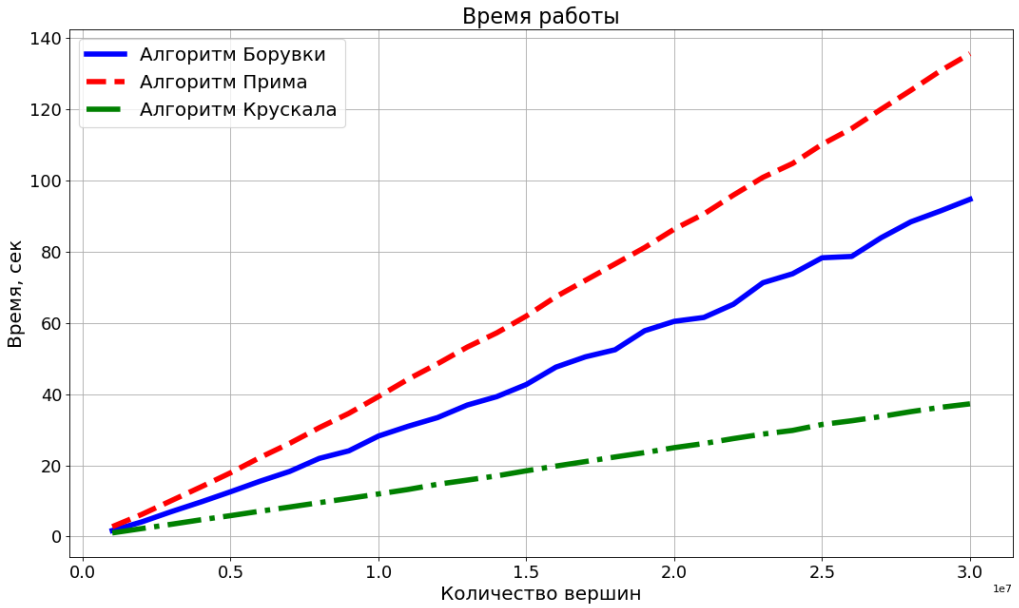

Рисунок 28 - Время работы последовательных алгоритмов поиска MST на разреженном графе

Результаты потребляемой памяти схожи с теми, что были получены при тестировании на графе цепочка. По-прежнему алгоритм Крускала потребляет наименьшее число оперативной памяти. Алгоритм Борувки требует на 25% больше оперативной памяти чем алгоритм Крускала, а алгоритм Прима на 175%. На рисунке 29 приведены полученные результаты потребляемой оперативной памяти последовательными алгоритмами поиска MST.

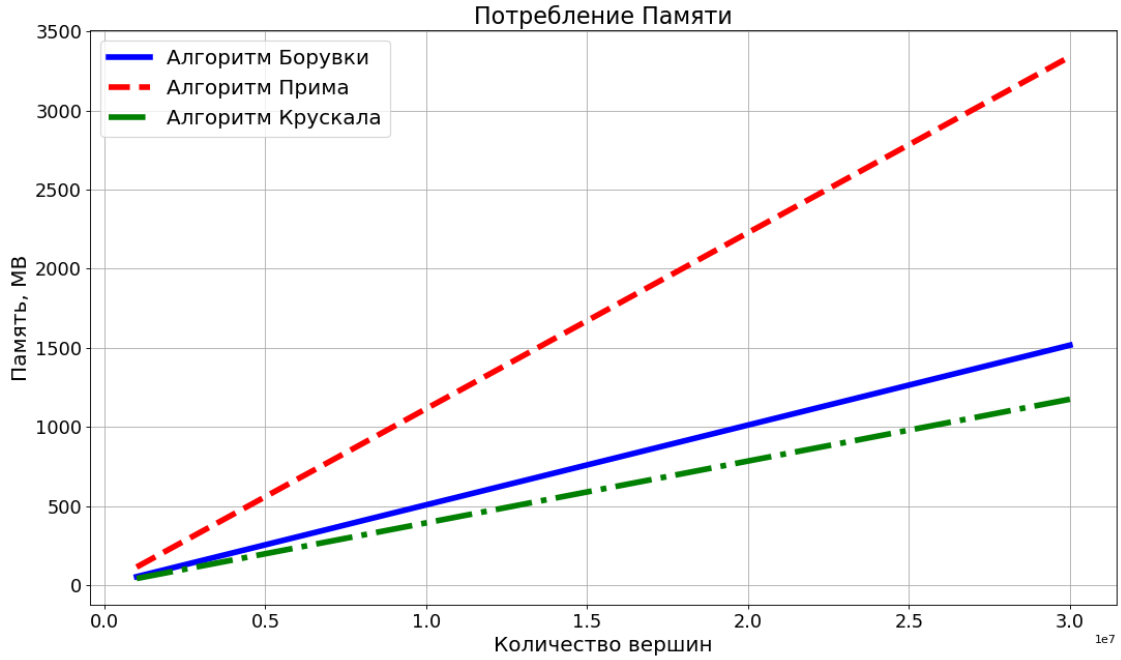

Рисунок 29 - Потребление памяти последовательных алгоритмов поиска MST на разреженном графе

Ожидалось, что на разреженном графе лучший результат покажет алгоритм Прима так, как именно используемая версия алгоритма ориентирована на применение её на таком графе.

Анализируя полученные результаты тестирования на разреженном графе можно однозначно сделать вывод, о том, что на разреженном графе лучше выбрать алгоритм Крускала так, как он показывает лучший результат по потребляемой памяти и по времени работы.

3.2.4 Сравнения последовательных алгоритмов поиска MST на графе-звезда

На графе-звезда наименьшее время выполнения показал алгоритм Борувки. Такой результат очевиден так, как алгоритм выполняет всего одну итерацию. Каждая вершина кроме центральной выбирает одно единственное направление и сразу происходит объединение всего графа. Алгоритм Прима максимум на второй итерации цикла добавит в set все ребра графа. Затем на всех оставшихся итерация ему приходится выполнять поиск минимального ребра за $O(log_2\ n\ )$, где n число ребер. Именно по этой причине алгоритм Прима показал на графе-звезда время работы хуже, чем на графе цепочка. По результатам тестирования (рисунок 30) алгоритм Крускала медленнее алгоритма Борувки на 50%, а алгоритм Прима на 500%.

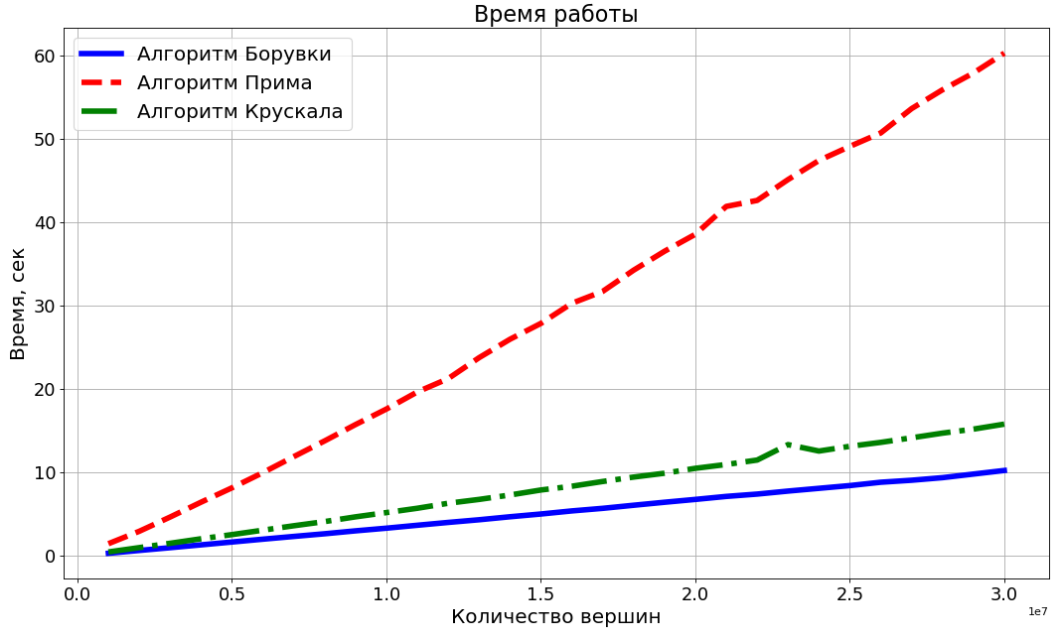

Рисунок 30 - Время работы последовательных алгоритмов поиска MST на графе-звезда

Результаты потребление оперативной памяти у алгоритма Борувки и алгоритма Крускала полностью совпадает с теми, что были получены при тестировании на графе-цепочка, а вот алгоритм Прима увеличил потребление памяти на 72%. Результаты тестирования потребления оперативной памяти представлены на рисунке 31.

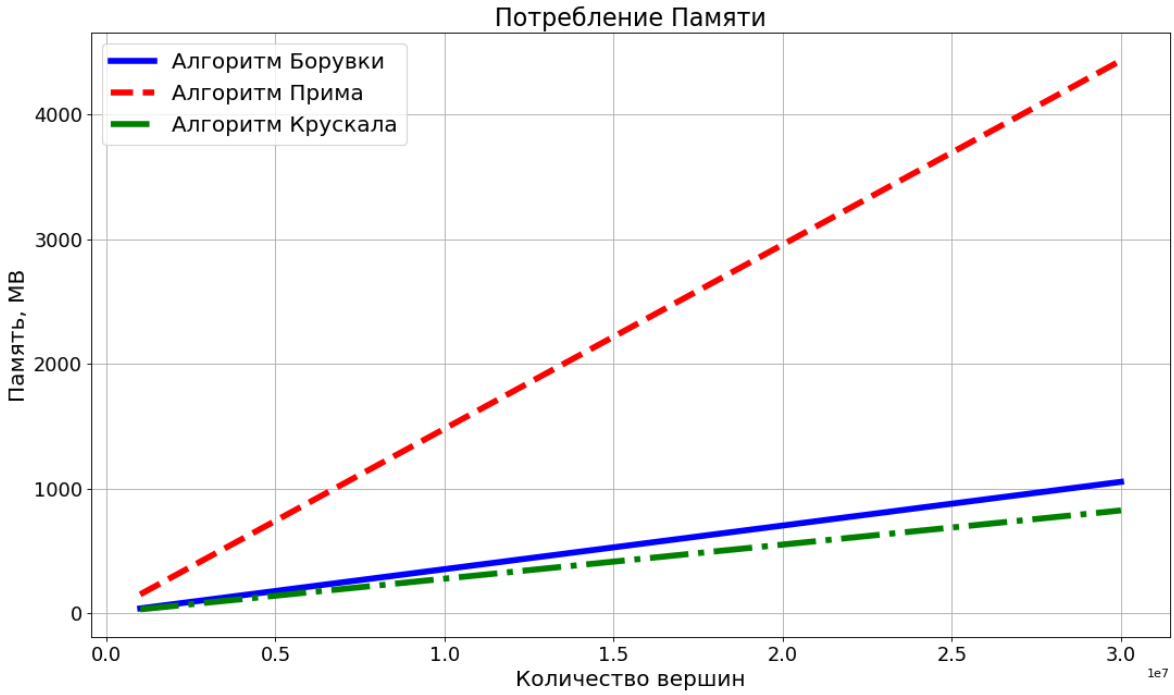

Рисунок 31 - Потребление памяти последовательных алгоритмов поиска MST на графе-звезда

Граф-звезда является идеальным для алгоритма Борувки так, как число итераций на нем сведено к минимуму. Он будет самым лучшим выбором для графов имеющих схожую структуру. Однако для алгоритма Прима вершины с высокой степенью являются долгими в обработки. Алгоритм Крускала аналогично будет предпочтительнее если требуется минимизировать потребляемую память.

3.3 Анализ параллельных алгоритмов поиска минимального остовного дерева

3.3.1 Параметры оценки параллельных алгоритмов поиска MST

Для оценки параллельных алгоритмов поиска MST будем использовать такие понятия как ускорение и эффективность.

Ускорением параллельного алгоритма называют отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма (формула 2).

$$S_p\ n\ = \frac{T_1(n)}{T_p(n)} \quad (2)$$

Эффективность параллельного алгоритма вычисляется как отношение ускорения алгоритма на число используемых процессоров (формула 3).

$$E_p\ n\ = \frac{S_p(n)}{p} \quad (3)$$

Параллельный алгоритм даже имея большое ускорение не всегда может использовать процессор эффективно. Так при росте числа процессоров падает эффективность их использования. В идеале ускорение должно ровняться числу используемых процессоров, то есть $S_p\ n\ = p$, но на практике такое линейное ускорение не всегда достижимо и зависит от многих факторов [14].

3.3.2 Оценка параллельного алгоритма Крускала

Параллельный алгоритм Крускала показывает невысокое ускорение и малую эффективность использования процессора. Такой результат связан с малой долей параллельных вычислений в алгоритме. Параллельные вычисления выполняются только во время сортировки, а дальнейшие вычисления происходят в одном потоке. Плюсом параллельного алгоритма Крускала является то что он не требует дополнительной оперативной памяти в сравнение с его последовательной версией.

В результате тестирования алгоритма Крускала на 1, 2, 4, 8 процессорах получены результаты, представленные в таблице 3 и на рисунке 32.

Таблица 3 – Результаты тестирования параллельного алгоритма Крускала

Кол-во вершин	**Последовательный алгоритм**	**Параллельные алгоритмы**								
		2 процессора			**4 процессора**			**8 процессоров**		
		T_2	S_2	E_2	T_4	S_4	E_4	T_8	S_8	E_8
10^6	1,04	0,90	1,15	0,57	0,89	1,16	0,29	0,78	1,33	0,16
$4 * 10^6$	4,72	4,12	1,14	0,57	3,99	1,18	0,29	3,54	1,33	0,16
$7 * 10^6$	8,30	7,33	1,13	0,56	6,89	1,20	0,30	6,35	1,30	0,16
$10 * 10^6$	11,96	10,63	1,12	0,56	9,81	1,21	0,30	9,12	1,31	0,16
$13 * 10^6$	15,84	14,30	1,10	0,55	13,21	1,19	0,29	12,09	1,31	0,16

Кол-во вершин	Последовательный алгоритм	Параллельные алгоритмы								
		2 процессора			4 процессора			8 процессоров		
		T_2	S_2	E_2	T_4	S_4	E_4	T_8	S_8	E_8
$16 * 10^6$	19,78	17,88	1,10	0,55	17,06	1,15	0,28	15,18	1,30	0,16
$19 * 10^6$	23,56	21,46	1,09	0,54	20,89	1,12	0,28	18,25	1,29	0,16
$22 * 10^6$	27,50	24,94	1,10	0,55	23,78	1,15	0,28	21,15	1,30	0,16
$25 * 10^6$	31,45	28,18	1,11	0,55	26,78	1,17	0,29	23,95	1,31	0,16
$28 * 10^6$	35,05	31,35	1,11	0,55	28,96	1,21	0,30	26,94	1,30	0,16

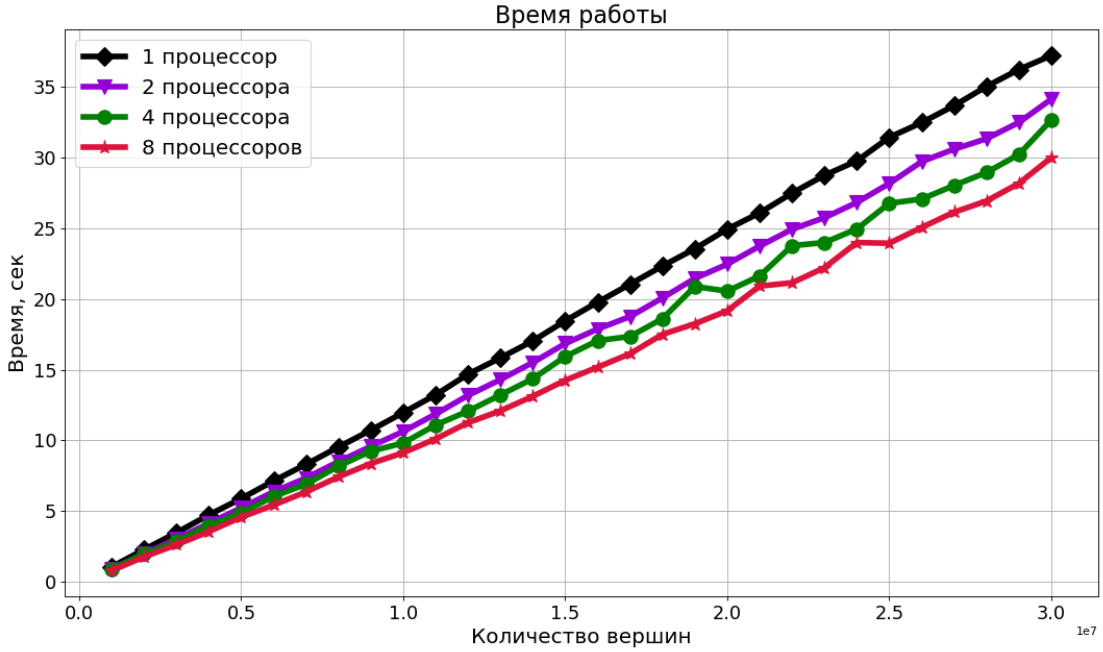

Рисунок 32 – Результаты тестирования параллельного алгоритма Крускала

3.3.3 Оценка параллельного алгоритма Борувки

Параллельный алгоритм Борувки показал ускорение лучше, чем алгоритм Крускала и на 8 ядрах удалось добиться более чем двукратного ускорения. Эффективность использования процессора также лучше, чем у алгоритма Крускала. Результаты тестирования параллельного алгоритма Борувки представлены в таблице 4 и на рисунке 33.

Таблица 4 – Результаты тестирования параллельного алгоритма Борувки

Кол-во вершин	Последовательный алгоритм	Параллельные алгоритмы								
		2 процессора			4 процессора			8 процессоров		
		T_2	S_2	E_2	T_4	S_4	E_4	T_8	S_8	E_8
10^6	1,70	1,58	1,07	0,53	1,21	1,14	0,35	1,03	1,65	0,20
$4 * 10^6$	9,72	8,29	1,17	0,58	5,99	1,62	0,40	4,96	1,95	0,24
$7 * 10^6$	18,29	15,46	1,18	0,59	11,14	1,64	0,41	8,90	2,05	0,25
$10 * 10^6$	28,20	23,20	1,21	0,60	16,60	1,69	0,42	13,12	2,14	0,26
$13 * 10^6$	36,89	31,27	1,17	0,58	22,16	1,66	0,41	17,44	2,11	0,26
$16 * 10^6$	47,54	39,15	1,21	0,60	27,49	1,72	0,43	22,74	2,09	0,26
$19 * 10^6$	57,74	47,55	1,21	0,60	34,12	1,69	0,42	26,78	2,15	0,26
$22 * 10^6$	65,17	55,73	1,16	0,58	39,93	1,63	0,40	31,28	2,08	0,26
$25 * 10^6$	78,21	65,69	1,19	0,59	45,83	1,70	0,42	36,23	2,15	0,26
$28 * 10^6$	88,28	74,45	1,18	0,59	50,53	1,74	0,43	40,29	2,19	0,27

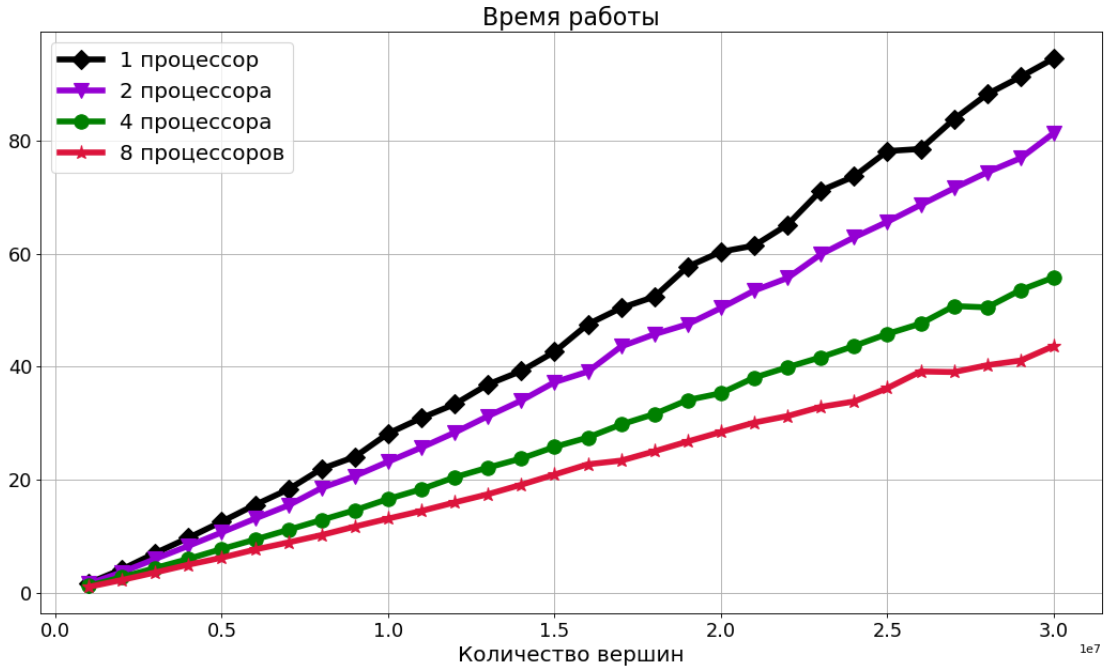

Рисунок 33 - Результаты тестирования параллельного алгоритма Борувки

Алгоритм Борувки требует для каждого ядра $O(n)$ дополнительной памяти, где n число вершин. Такое увеличение дополнительное потребление памяти может быть не рационально на графах с малым числом ребер.

3.3.4 Оценка параллельного алгоритма Прима

Параллельный алгоритм Прима не ускорялся при увеличении числа процессоров. Такой результат связан с тем, что каждый процессор получает на вход часть строки в матрице смежности равной $\frac{n}{p}$, где n число вершин, а p число процессоров. Так как порядок числа n из-за ограничений памяти не может быть больше 4, то не удается получить прироста в производительности. Результаты тестирования алгоритма Прима представлены в таб. 5 и на рис. 34.

Таблица 5 – Результаты тестирования параллельного алгоритма Прима

Кол-во вершин	Последовательный алгоритм	Параллельные алгоритмы								
		2 процессора			4 процессора			8 процессоров		
		T_2	S_2	E_2	T_4	S_4	E_4	T_8	S_8	E_8
10^3	0,09	0,08	1,125	0,56	0,08	1,125	0,28	0,08	1,125	0,14
$3 * 10^3$	0,73	0,74	0,98	0,49	0,74	0,98	0,24	0,74	0,98	0,12
$5 * 10^3$	2,02	2,03	0,99	0,49	2,02	1,00	0,25	1,97	1,02	0,12
$7 * 10^3$	3,95	3,96	0,99	0,49	3,94	1,00	0,25	3,94	1,00	0,12
$9 * 10^3$	6,52	6,51	1,00	0,50	6,43	1,01	0,25	6,49	1,00	0,12
$11 * 10^3$	9,66	9,70	0,99	0,49	9,87	0,97	0,24	9,67	0,99	0,12
$13 * 10^3$	13,56	13,48	1,00	0,50	13,48	1,00	0,25	13,45	1,00	0,12
$15 * 10^3$	18,04	18,01	1,01	0,50	17,83	1,01	0,25	17,97	1,00	0,12
$17 * 10^3$	23,21	22,79	1,01	0,50	23,07	1,00	0,25	22,97	1,01	0,12
$19 * 10^3$	28,94	28,84	1,00	0,50	28,81	1,00	0,25	28,57	1,01	0,12

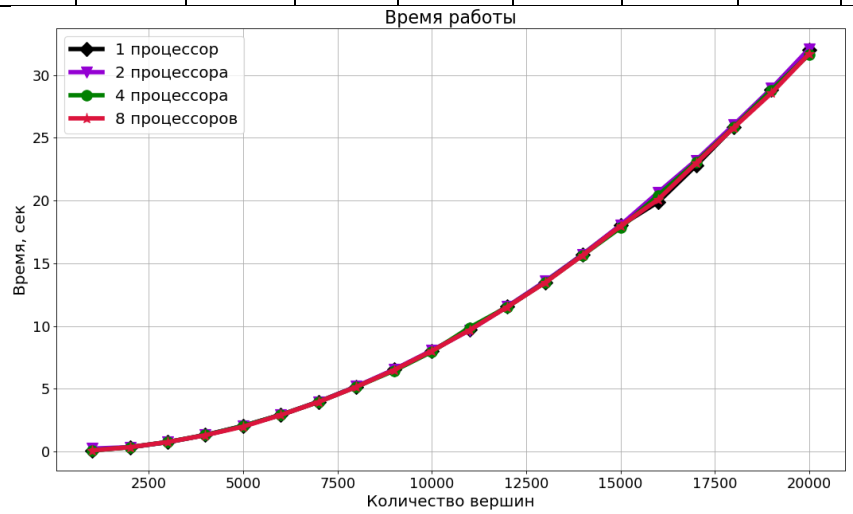

Рисунок 34 - Результаты тестирования параллельного алгоритма Прима

Параллельный алгоритм Прима аналогично, как и алгоритм Крускала не требует дополнительной памяти с ростом числа процессоров.

3.4 Сравнение параллельных алгоритмов поиска MST

По результатам исследования параллельных алгоритмов можно сделать вывод, что алгоритм Борувки имеет лучшее ускорение с ростом числа процессоров. Также у алгоритма остается возможность добавить параллельные вычисления в структуру данных DSU, используя её параллельную версию. Алгоритм Крускала ускоряется незначительно, но в отличии от алгоритма Борувки не требует дополнительной памяти. Алгоритм Прима хоть и имеет высокую долю параллельных вычислений, но из-за невозможности хранить достаточно большую матрицу смежности не эффективно порождать новые потоки для выполнения небольшого числа операций. На рисунке 35 представлен рисунок диаграммы зависимости числа используемых процессоров от ускорения алгоритмов.

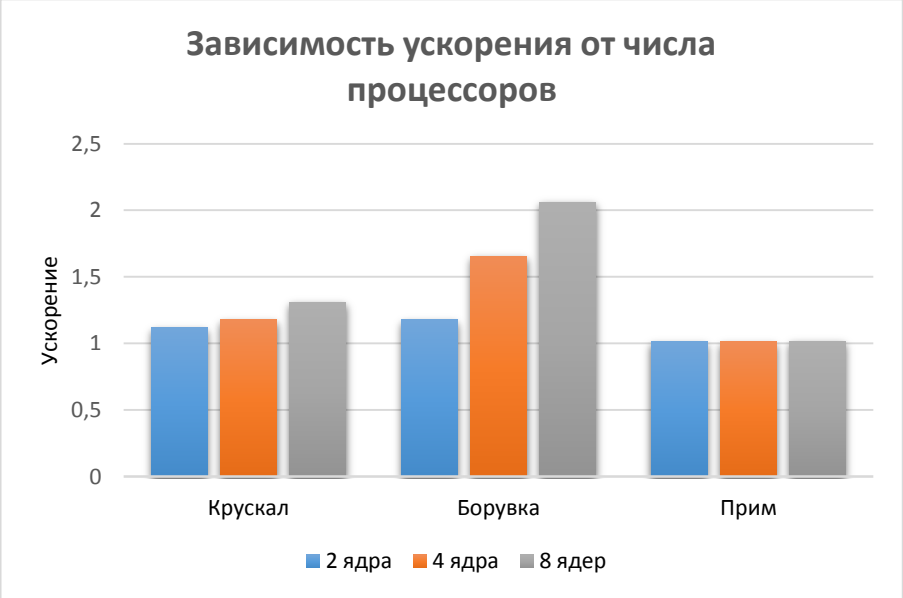

Рисунок 35 – Сравнение среднего ускорения параллельных алгоритмов поиска MST

По диаграмме заметно, что алгоритм Борувки лучше ускоряет с ростом числа процессоров чем алгоритм Крускала.

По результатам исследования параллельных алгоритмов можно сделать вывод, что если требуется организовать параллельный поиск MST, то для более эффективного использования процессоров следует выбирать алгоритм Борувки.

ЗАКЛЮЧЕНИЕ

В выпускной квалификационной работе произведен анализ основных алгоритмов поиска минимального остовного дерева.

Были проанализированы такие алгоритмы как алгоритм Крускала, алгоритм Прима и алгоритм Борувки. Разработаны их параллельные модели. Реализована тестирующая система для генерации графов, тестирования алгоритмов и сбора результатов. Проанализированы полученные результаты.

Каждый алгоритм имеет свои особенности и по-разному ведет себя на различных графах. Так было выяснено, что алгоритм Крускала по сравнению с другими алгоритмами потребляет наименьшее число оперативной памяти и показывает на разряженных графах лучший результат. Асимптотика работы алгоритма значительно зависит от используемой сортировки. Однако его параллельная версия обладает небольшим ускорением.

Алгоритм Прима отлично подходит для полных графов и выбирать его следует если граф хранится в виде списка смежности или матрицы смежности. Минусом алгоритма является высокое потребление памяти и отсутствием ускорением при использовании параллельных вычислений.

Алгоритм Борувки показывает хорошие результаты на более плотных графах и не высокое потребление памяти. Идеальным графом для алгоритма Борувки является граф-звезда. Также алгоритм Борувки обладает хорошим ускорением при использовании параллельных вычислений.

Полученные данные в результате выполнения выпускной квалификационной работы могут быть использованы для определения лучшего алгоритма для конкретной практической задачи.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Томас Х. Кормен, Чарльз И.Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы. Построение и анализ. М.: Вильямс, 2018. 1328 с.
2. Берцун В.Н. Математическое моделирование на графах. Часть 2. Томск: Изд-во Том. ун-та, 2014. 86 с.
3. Стивен Скиена. Алгоритмы. Руководство по разработке. СПБ.: БХВ-Петербург, 2015. 720 с.
4. Параллельная реализация алгоритмов поиска остовных деревьев с использованием центрального и графического процессоров / Колганов А.С. // Вестник ЮУр-ГУ. Серия: Вычислительная математика и информатика. 2016. Т. 5, №3. С. 5-19.
5. Седжвик Роберт. Фундаментальные алгоритмы на C++. Алгоритмы на графах. СПБ.: ООО «ДиаСофтЮП». 2014. 496 с.
6. Белоусов А.И., Ткачев С.Б. Дискретная математика. М.: МГТУ, 2016. 744с.
7. Vitaly Osipov, Peter Sanders, Johannes Singler. The Filter-Kruskal Minimum Spanning Tree Algorithm // Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009, New York, USA, January 3, 2009. 19 p.
8. Prim, R. C. Shortest connection networks And some generalizations, Bell System Technical Journal, 1957. 1389–1401 p.
9. Arun Nedunchezhian, Shankar Balachandran, Rohit Setia. A new parallel algorithm for minimum spanning tree problem. // Proc. International Conference on High Performance Computing, 2009.
10. Quinn, Michael J, Deo Narsingh. Parallel graph algorithms. // ACM Computing Surveys (CSUR), 1984. 319–348 p.
11. Loncar Vladimir, Skrbic Srdjan, Balaz Antun. Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures. // Transactions on Engineering Technologies. 2014. 543–554 p.

12. Sun Chung, A. Condon. Parallel implementation of Bouvka's minimum spanning tree algorithm // Proceedings of International Conference on Parallel Processing. 1996.

13. Мейерс Скотт. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14. М.: Вильямс, 2018. 304 с.

14. Энтони Уильямс. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. М.: ДМК Пресс, 2016. 672 с.

15. Parallelizing Boruvka's Algorithm [Электронный ресурс] URL: http://kfuh1.github.io/15418-project/# (дата обращения: 11.04.2019).

16. Minimum Spanning Trees [Электронный ресурс] URL: https://drive.google.com/file/d/0B4z2gzEmkDDCTUhKUWxJQnR0bEk/view (дата обращения: 22.05.2019).

17. A Lock-Free Implementation of Union-Find [Электронный ресурс] URL: https://is.muni.cz/th/kpdej/BakalarkaDigital.pdf (дата обращения: 02.03.2019).

18. Minimum Spanning Tree: Definition, Examples, Prim’s Algorithm [Электронный ресурс] URL: https://www.statisticshowto.datasciencecentral.com/minimum-spanning-tree/ (дата обращения: 10.03.2019).

19. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. М.: Изд-во МГУ, 2009. 77 с.

20. Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн. C++ Boost Graph Library. СПБ.: Питер, 2006. 304 с.

21. Брайан Уорд. Внутреннее устройство Linux. СПБ.: Питер, 2016. 384 с.