Петрозаводский государственный университет
Институт математики и информационных технологий
Кафедра прикладной математики и кибернетики

Направление подготовки бакалавриата

09.03.04 - Программная инженерия

Отчет о практике по научно-исследовательской работе

Решение задачи о назначениях с помощью потоков минимальной стоимости. Сравнение времени работы с венгерским алгоритмом.

(промежуточный)

Выполнил:
студент 2 курса группы 22207

К. С. Кришталь _____
подпись

Место прохождения практики:
Кафедра прикладной математики и кибернетики

Период прохождения практики:
Период прохождения практики:
02.09.22-22.12.22

Руководители:
Первый руководитель Р. В. Сошкин, к.т.н.

_____
подпись

Второй руководитель Р. В. Алькин

Итоговая оценка

_____
оценка

Петрозаводск — 2022

Содержание

Введение **3**

1 Другие постановки задачи **3**

2 Методы решения **4**

3 Подходя к реализации **4**

4 Подробнее о каждом решении **5**

4.1 Полный перебор. 5

4.2 Сведение задачи к поиску потока фиксированной величны с минимальной стоимостью 5

4.3 Венгерский алгоритм 7

4.3.1 Лемма 1: 7

4.3.2 Лемма 2: 7

4.3.3 Первая версия алгоритма 7

4.3.4 Улучшение до $\mathcal{O}(n^3 \cdot m)$ 9

4.3.5 Улучшение до $\mathcal{O}(n^2 \cdot m)$ 10

4.3.6 Тонкости реализации венгерского алгоритма за $\mathcal{O}(n^2 \cdot m)$ 11

5 Тестирование **11**

6 Анализ времени работы и сравнение алгоритмов **12**

7 Заключение **19**

8 Список использованных источников **20**

Введение

Цель практики: Решить задачу о назначениях несколькими способами, и провести сравнительный анализ.

Задачи практики:

- Сформулировать математическую модель задачи
- Изучить алгоритмы, которые могут подойти для решения задачи
- Реализовать алгоритмы решения задачи на языке программирования
- Провести сравнительный анализ реализованных алгоритмов

Актуальность работы: Исследование способов решения задачи о назначениях с целью установления наиболее подходящих по различным критериям.

Можно сформулировать задачу о назначениях следующим образом:

Есть n рабочих и n заданий. Для каждого рабочего известно, сколько денег он запросит за выполнение того или иного задания. Каждый рабочий может взять себе только одно задание. Требуется распределить задания по рабочим так, чтобы минимизировать суммарные расходы.

В приведенной выше формулировке число задач равно числу рабочих. Но можно привести задачу в более общем случае, если число задач сделать m, (во всей работе принято считать (не умаляя общности) $m \geq n$). Тогда нужно будет взять какие то n задач, а остальные не использовать. Такая задача сводится к вышеописанной, путем добавления $m - n$ фиктивных рабочих, требующих бесконечно большие зарплаты.

Так же задачу можно свести не к минимизации, а к максимизации суммарной искомой величины. Для этого достаточно умножить все коэфициенты на -1.

1 Другие постановки задачи

Двудольный граф (графовая):

Дан полный взвешенный двудольный граф с n вершин в левой доле и m вершин в правой доле. Необходимо найти минимальное по сумме весов (весу) полное паросочетание

(паросочетние размера n).

Матричная (табличная):

Дана матрица $n \times m$. Необходимо выбрать n элементов так, чтобы максимизировать сумму элементов и никакие 2 взятых элемента не совпадали по строке или столбцу.

Я часто буду ссылаться как на графовое, так и на матричное представление задачи. Например говоря о "левой доле "рассмотрим ребра и т.п. имеется ввиду графовая постановка, а говоря о "строках "столбцах и т.п. речь идет о матричном представлении.

2 Методы решения

Классическим решением для данной задачи является венгерский алгоритм, разработанный и опубликованный Гаральдом Куном (именем которого назван алгоритм поиска максимального паросочетания в невзвешенном графе). Название "венгерский"Кун дал алгоритму, так как он был в значительной степени основан на более ранних работах двух венгерских математиков: Денеша Кёнига и Эйгена Эгервари.

Первоначальный алгоритм Куна имел асимптотическую оценку времени работы $\mathcal{O}(n^3 \cdot m)$, и лишь позже Джек Эдмондс и Ричард Карп (и независимо от них Томидзава) показали, каким образом улучшить его до асимптотики $\mathcal{O}(n^2 \cdot m)$. В рамках данной курсовой работы будут рассмотрены версии с асимптотикой $\mathcal{O}(n^4 \cdot m)$, $\mathcal{O}(n^3 \cdot m)$ и $\mathcal{O}(n^2 \cdot m)$, а так же небольшие подверсии.

В качестве альтернативного решения для этой задачи можно использовать поиск потока минимальной стоимости. В рамках работы рассматривается поиск поиск потока минимальной стоимости методом дополнения вдоль путей минимальной стоимости в 3 вариациях (с использованием алгоритма Форд-Беллмана, Левита и Дейкстры с потенциалами Джонсона).

Для сравнения будет реализовано решение с перебором всевозможных полных паросочетаний.

3 Подходя к реализации

Все перечисленные в пункте "методы решения"алгоритмы будут написаны на языке C++. Все решения реализованы в виде классов, которые наследуют абстрактный класс

$Solve\_base$ (приложение A).

Все решения, Реализующие поиск потока минимальной стомости, наследуют класс $Min\_cost\_flow\_base$ (приложение C)

4 Подробнее о каждом решении

4.1 Полный перебор.

Декомпозируем задачу. Переберем все возможные паросочетания на полном графе, и выберем лучшее из них. Декомпозируем перебор паросочетаний. поскольку $m \geq n$, сначала переберем все подмножества правой доли размера n (все возможные сочетания из правой доли по n), после чего переберем все возможные варианты сопоставления элементов левой доли с элементами фиксированного подмножества (легко реализуется путем перебора всех перестановок длины n). Перебор сочетаний реализуется используя простую рекурсивную функцию. Итоговое время работы алгоритма - это время перебора всех паросочетаний и подсчет стоимости каждого из них - $\mathcal{O}(\frac{m!}{(m-n)!} \cdot n)$. Реализацию смотреть в приложении B.

4.2 Сведение задачи к поиску потока фиксированной величны с минимальной стоимостью

Рассмотрим графовую постановку задачи. Ориентируем ребра из левой доли в правую, присвоим им пропускную способность 1, а стоимость за еденицу потока сделаем равной весу изначального ребра. Добавим фиктивные истоковую (s) и стоковую (t) вершины (для поиска потока между парой вершин). Направим из истока ребро в каждую вершину левой доли, а из каждой вершины правой доли направим ребро в сток, всем этим ребрам присвоим пропускную способность 1, и стоимость 0. Очевидно, что в такой постановке задачи о поиске потока минимальной стоимости мы найдем паросочетание минимального веса.

Для поиска потока минимальной стоимости воспользуемся следующей схемой:

1. строим остаточную сеть
2. если в остаточной сети есть путь из s в t переходим к шагу 3, иначе найден оптимальный поток.
3. ищем путь минимальной стоимости из s в t.

4. дополняем поток вдоль этого пути и идем к шагу 2

Внешний цикл алгоритма выполнится n раз (с каждой итерацией мы увеличиваем поток на еденицу). Дополнение пути вдоль потока займет $\mathcal{O}(n)$ операций. Таким образом, можно выразить асимптотическую оценку времени работы решения как $\mathcal{O}(n \cdot (n + F(s,t))$, где $F(s,t)$ - поиск кратчайшего по сумме стоимостей пути из s в t. Поскольку время работы $F(s,t)$ в нормальных условиях не меньше $\mathcal{O}(n)$, то в конечном счете можно сказать, что асимптотика решения $\mathcal{O}(n \cdot F(s,t)$.

Я рассматриваю 3 возможных выбора реализации $F(s,t)$. Поскольку в остаточной сети могут быть ребра отрицательной стоимости, то необходимо выбрать алгоритм поиска именно на графах с отрицательными весами (отрицательные циклы отсутствуют).

Оценим число вершин и ребер в остаточной сети. Число вершин: $n + m + 2$ (O(m)). Число ребер: $n \cdot m + n + m$ ($\mathcal{O}(n \cdot m)$). Так же введем обозначения: $|V|$ - число вершин, $|E|$ - число ребер.

Самым простым способом будем взять в качестве $F(s,t)$ алгоритм Форд-Беллмана. Этот алгоритм до неприличия прост, и имеет более менее адекватное время работы $\mathcal{O}(|V| \cdot |E|)$. Реализация с таким алгоритмом будет работат за $\mathcal{O}(n^2 \cdot m^2)$ (см. Приложение D).

Вместо алгоритма Форд-Беллмана можно использовать алгоритм Левита. Данный алгоритм имеет более плохую асимптотическую оценку - $\mathcal{O}(|V|^2 \cdot |E|)$. Однако, на практике данный алгоритм работает обычно быстрее Форд-Белманна, и достагает такой оценки времени работы только в очень особенных графах. Более того, на практике этот алгоритм обычно почти не отличается от алгоритма Дейкстры. Итоговая оценка асимптотики - $\mathcal{O}(n^2 \cdot m^3)$ (см. Приложение E).

Алгоритм Дейкстры работает только на графах без отрицательных весов (поэтому его дефолтный вариант не удовлетворяет для использования его в нашей задаче). Но существует специальная модификация, с использованием потенциалов Джонсона. Перед началом поиска потока, Необходимо запустить 1 раз алгоритм корректно работающий на графах с отрицательными весами (будем использовать Форд-Беллмана), а далее использовать обычный алгорит Дейкстры, с поправкой на потенциалы. Алгоритм Дейкстры, реализованный в рамках задачи будет работать за $\mathcal{O}(|V|^2)$, поэтому итоговое время работы $\mathcal{O}(n \cdot m^2)$ - что так же является асимптотической оценкой времени работы алгоритма Форд-Беллмана и поиска потока с Дейкстрой без его учета (см Приложение F).

4.3 Венгерский алгоритм

Для начала рассмотрим и докажем несколько лемм.

4.3.1 Лемма 1:

Если к весам всех ребер, инцедентных какой либо вершине левой доли, прибавить одно и то же число X, в новом графе оптимальное паросочетание будет то же, что и в старом, а ответ изменится на X.

Доказательство:

Полное паросочетание для каждой вершины левой доли содержит ровно одно ребро, инцидентное этой вершине. Указанная операция изменит на одно и то же число вес любого паросочетания. Значит, ребро, которое принадлежало оптимальному паросочетанию в старом графе, в новом графе тоже будет ему принадлежать.

Следствия:

В квадратной формулировке задачи ($n = m$), можно проделывать ту же операцию для вершин правой доли. Так же, если какая то вершина правой доли точно будет входить в паросочетание, из таких же соображений можно проделывать и с ней эту операцию.

Рассматривая же прибавление константы ко всех ребрам инцедентным одной вершине, нужно понимать, что в матричной постановке мы прибавляем константу к строке (для левой доли) или к столбцу (для правой).

Из этой леммы можно легко перейти к задаче с неотрицательными весами (вычтем из каждой строки ее минимальный элемент).

4.3.2 Лемма 2:

Если веса всех ребер графа неотрицательны и некоторое полное паросочетание состоит из ребер нулевого веса, то оно является оптимальным.

Доказательство:

Любое паросочетание будет иметь неотрицательный вес (так как отстуствуют ребра отрицательного веса), из чего следует что паросочетания весом меньше чем 0 получить невозможно.

4.3.3 Первая версия алгоритма

Обозначим за G_0 - граф на ребрах веса 0 из исходного.

Будем пытаться строить полное в G_0. Как только сможем построить полное паросочетание с 0 весом - по лемме 2 мы нашли ответ.

Воспользуемся алгоритмом Куна (для поиска максимального паросочетания в невзвешенном графе). Кратко напомним, в чем заключется алгоритм:

- Все ребра которые не входят в паросочетание ориентированы из левой доли в правую, а ребра входяшие в паросочетание - из правой в левую.
- Из каждой ненасыщенной (не входящей в паросочетание) вершины левой доли запускается обход в глубину. Если была достигнута ненасыщенная вершина правой доли, то мы нашли дополняюший путь, и можем "прочередовать"ребра вдоль него (ребра входящие в паросочетание - исключить из него, а не входящие - включить)
- Если не существует дополняющего пути - то паросочетание максимально.

Но что делать если при построении паросочетания оно оказалось неполным? В таком случае добавим новые ребра в G_0, пользуясь леммой 1. Осталось установить каким именно образом это сделать.

Рассмотрим множество вершин левой доли, которые не входят в паросочетание. Теперь рассмотрим все вершины, которые были посещены алгоритмом Куна, при попытке добавить эти вершины в паросочетание. Обозначим за L^+ и R^+ - множество посещенных вершин левой и правой долей соответсвенно, а за L^- и R^- - непосещенных. Все вершины из R^+ входят в паросочетание (иначе возникло бы противоречие, из-за наличия доплняющего пути, который не попал в наше паросочетание).

Для поиска дополняющего пути нужно добавить ребро из множества посещенных вершин во множество непосещенных. Нельзя добавить ребро из R^+ в L^-, так как из любой вершины правой доли может исходить только 1 ребро, и оно уже исходит в вершину в L^+. Значит нам необходимо добавить ребро из L^+ в R^-. Все эти ребра в исходном графе имеют вес, больший 0 (иначе было бы ребро в G_0 между L^+ и R^- - противоречие).

Найдем минимальный вес среди всех этих ребер (обозначим его за mn) и вычтем его из всех ребер в исходном графе между L^+ и R^-. Для этого, пользуясь леммой 1, Вычтем mn из весов всех ребер, инцедентных с вершинами из L^+.

Но теперь мы могли создать ребра отрицательного веса между L^+ и R^+, что портит нам алгоритм. Поэтому Теперь добавим mn ко всем ребрам инцедентным с R^+, и это так же корректно по лемме 1 (поскольку вершины из R^+ уже входят в паросочетание).

Однако, теперь мы могли удалить некоторые ребра из G_0, поскольку ребра между R^+ и L^- увеличили свои веса. Но это никак не влияет на уже достижимые вершины, посколько ребро ведет во множество еще не посещенных вершин. Зато теперь мы добавили как минимум 1 ребро между L^+ и R^-.

Таким образом, мы увеличили множество достижимых вершин, и можем повторить обход куна на новом G_0.

Итак, краткая схема алгоритма:

1. Ищем максимальное паросочетание в G_0. если оно является полным - алгоритм завершается (по лемме 2). иначе делаем следующий шаг

2. Вычитаем mn из всех ребер инцедентных с L^+ И прибавляем mn ко всем ребрам из R^+. Все веса остались неотрицательными, а множество доступных вершин расширяется. идем к шагу 1.

Оценим время работы алгоритма. Для увеличения размера максимального паросочетания алгоритму нужно не более n итераций, поскольку каждая итерация добавляет по крайней мере 1 ненасыщенную вершину правой доли к рассмотрению. Всего паросочетание может увеличится не более n раз. Следовательно произойдет не более $\mathcal{O}(n^2)$ внешних итераций цикла. На каждой итерации цикла алгоритм тратит $\mathcal{O}(|V_L|*|E|) = \mathcal{O}(n^2 \cdot m)$ ($|V_L|$ - число вершие в левой доле) на алгоритм Куна и $\mathcal{O}(n \cdot m)$ на переразвесовку. Итоговое время работы - $\mathcal{O}(n^4 \cdot m)$.

Реализация в приложении G.

4.3.4 Улучшение до $\mathcal{O}(n^3 \cdot m)$

Не будем строить паросочетание заново на каждой итерации алгоритма. Вместо этого будем рассматривать не сразу все вершины, а добавлять их к рассмотрению по очерди, ожидая, пока не будет найдено полное паросочетание на уже рассматриваемых вершинах. А точнее, пока мы не включим новую вершину в паросочетание, мы не будем рассматривать следующую за новой вершину.

Таким образом, на текущей итерации цикла будем пытаться вставить новую вершину в паросочетание, запуская алгоритм Куна только от нее, и рассматривать множества L^+, R^+, L^- и R^- так же только относительно запуска обхода Куном из этой вершины. А поскольку в первичном алгоритме самым долгим местом был шаг 1 (поиск паросочетания), то теперь сохранив число итераций алгоритма $\mathcal{O}(n^2)$ мы ускорили шаг 1 до $\mathcal{O}(|E|) = \mathcal{O}(n \cdot m)$, имея шаг 2 с таким же временем работы. Получаем итоговое время работы $\mathcal{O}(n^3 \cdot m)$.

Реализация в приложении H.

4.3.5 Улучшение до $\mathcal{O}(n^2 \cdot m)$

Внимание, в этом разделе активно используется матричная постановка задачи. Обозначим за a_{ij} матрицу, соответсвующую весам ребер, где a_{ij} - вес ребра между вершиной i из левой доли и вершиной j из правой.

Рассмотрим оба шага нашего алгоритма, и попробуем их ускорить.

Начнем с шага 1 (Обход алгоритмом Куна). Заметим, что пока мы не включили новую вершину в паросочетание, каждый обход включает в себя предыдущий обход, и все то, что получилось достичь после добавления новых ребер. Значит можно писать не классический обход, а итеративный, запускаясь всякий раз только от нового ребра, а не от рассматриваемой вершины. Таким образом, Если реализовать эту идею, то суммарное время работы шага 1, за все итерации алгоритма будет $\mathcal{O}(n^2 \cdot m)$ (для каждой рассматриваемой вершины мы суммарно выполним ровно одну итерацию обхода куном).

Рассмотрим шаг 2. Для начала введем 2 массива $lp_i, i = 1 \dots n$ и $rp_j, j = 1 \dots m$ (совокупность которых будет именоваться "потенциалом"). lp_i будет соответсвовать сумме всех прибавленных к строке i констант. rp_j будет соответствовать сумме всех прибавленных к столбцу j констант. Таким образом, можно избавиться от явного вычитания констант из каждого ребра (элемента матрицы) и производить операции по лемме 1 вместо линейного времени за константное, но тогда всякий раз при обращении к a_{ij} необходимо будет вычитать из него lp_i и rp_j.

Шаг 2 алгоритма по сути состоит из 2 частей:

1. Поиск минимума среди всех ребер между L^+ и R^-.

2. применение леммы 1 к некоторым строкам и столбцам.

Вторая операция теперь занимает $\mathcal{O}(m)$ времени. Осталось оптимизировать первую, которая изначально подразумевалась как тривиальный перебор всех элементов за $\mathcal{O}(n \cdot m)$.

Заведем вспомогательный массив $mnv_j, j = 1 \dots n$. Пусть mnv_j - минимум в j-ом столбце среди всех строк, которые принадлежат L^+. Будем поддерживать этот массив для всех столбцов, даже если столбец сейчас не принадлежит R^-. Для поддержания массива достаточно всякий раз, когда строка i переходит из R^- в R^+, сделать $mnv_j = min(mnv_j, a_{ij} - lp_i - rp_j), j = 1 \dots m$. Таким образом, поиск минимума среди всех ребер между L^+ и R^- сводится к перебору элементов mnv_j.

Таким образом, второй шаг теперь выполняется за $\mathcal{O}(m)$ (если считать, что на каждой итерации добавляется только по 1 строке, иначе достаточно рассмотреть суммарное время выполнения). Весь алгоритм теперь выполняется за $\mathcal{O}(n^2 \cdot m)$.

Реализация в приложении I, а так же модифицированная реализация в приложении J.

4.3.6 Тонкости реализации венгерского алгоритма за $\mathcal{O}(n^2 \cdot m)$

Реализовать шаг 1 алгоритма представляется достаточно затруднительным, если рассматривать графовую постановку задачи. Поэтому продолжая уход в сторону матриц, сделаем это и на этом шаге.

Поймем, что нам достаточно изначально добавить строку, соответсвующую рассматриваемой вершине к рассмотрению, а далее можно добавлять к рассмотрению столбцы по одному. При этом есть 2 случая:

- Столбец ненасыщен
- Столбец насыщен

В первом случае мы нашли дополняющий путь, прочередуем все по нему, и будем добавлять в паросочетание уже следующую строку.

Во втором случае мы не нашли чередующий путь, но у этого столбца есть соответствующая ему в паросочетании строка, поэтому добавим ее к рассмотрении и перейдем к следующей итерации алгоритма.

5 Тестирование

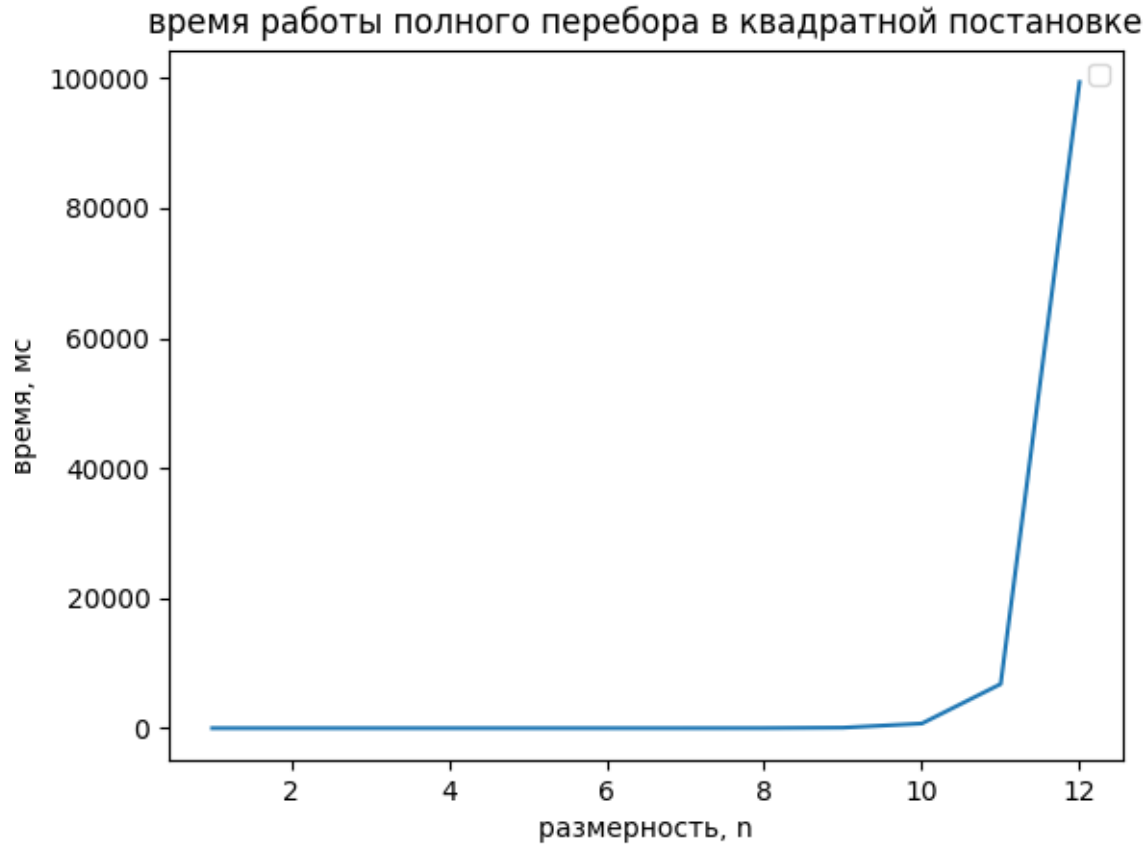
Для тестирования был реализован генератор, который принимает в качестве параметров минимальный и максимальный n, m и a_{ij} (вес в графе). Так же была создана дополнительная возможность с некоторой вероятностью, заданной в виде рациональнйо дроби, сделать ребро "бесконечного веса"(очень большого, значительно больше заданного диапазона).

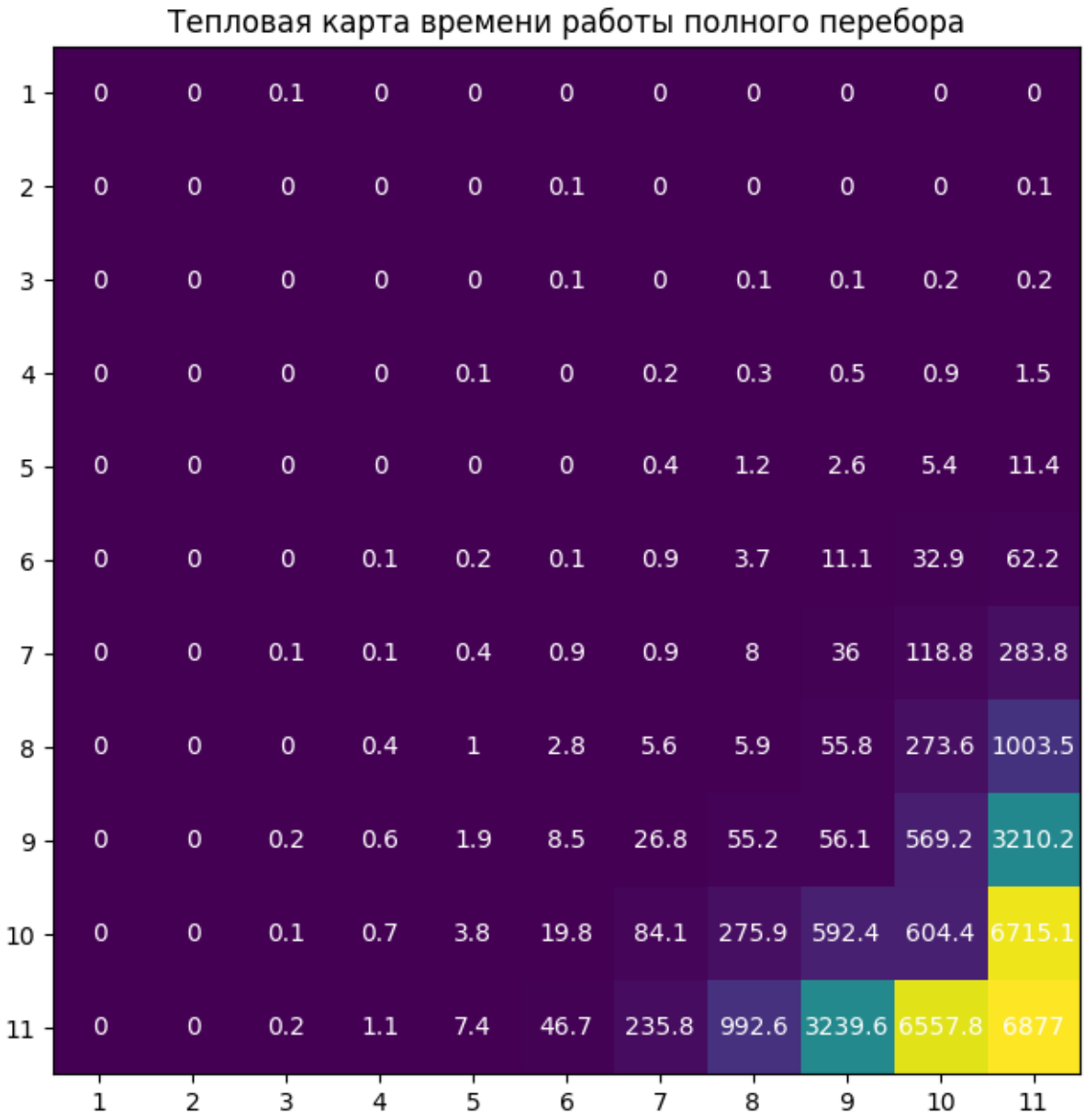
Все решения были совместно протестированы на огромном наборе сгенерированных данных, В процессе были устранены некоторые ошибки в нескольких решениях, отловленные при тестировании. Для всех нижеприведенных графиков, где встречается более 1 решения на графике, гененрировались одни и те же тесты для всех решений, после чего они по очереди на них запускались.

Реализация класса генерирующего тесты приведена в приложении K.

6 Анализ времени работы и сравнение алгоритмов

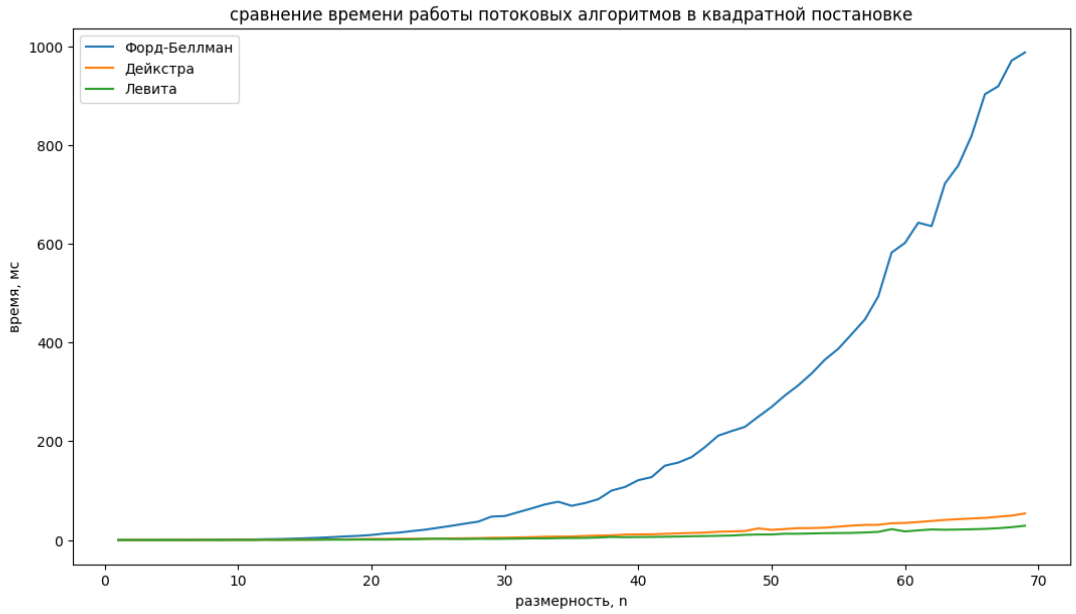
Решение использующее полный перебор особого интереса не представляет, и по скорости работы критически уступает всем остальным, но для полноты картины, вот некоторые данные о нем.





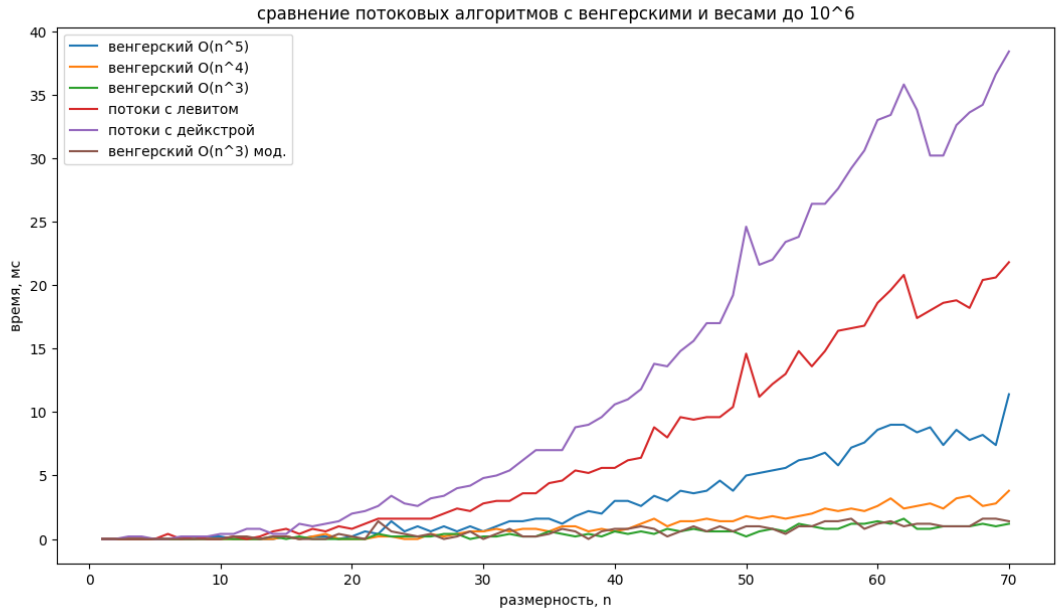
Вполне естественные графики для экспоненциального алгоритма.

Рассмотрим сравнение времени работы между всеми реализациями потокового алгоритма.



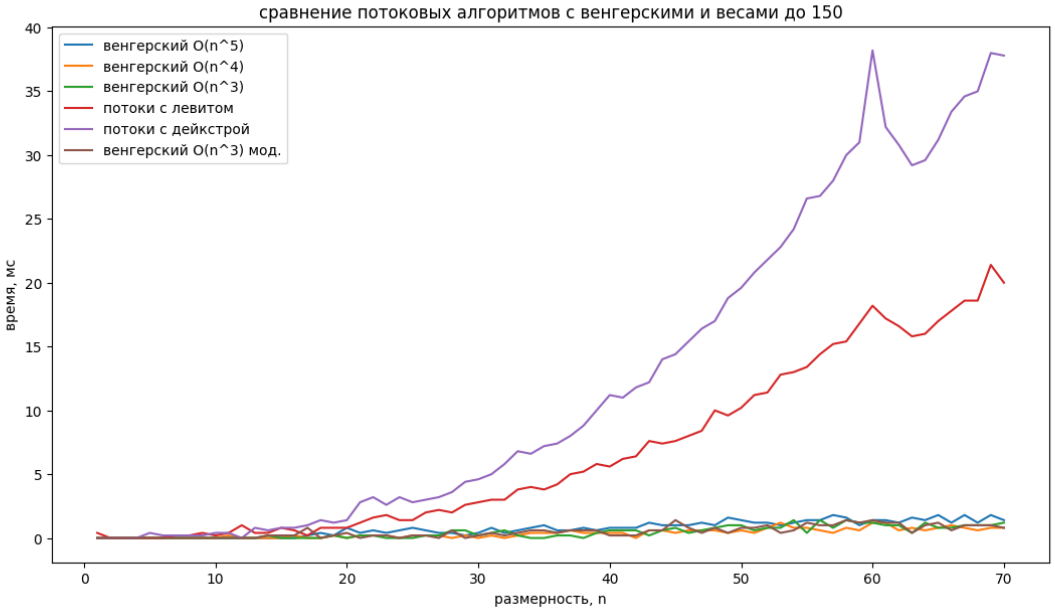
В сравнении реализации с использованием алгоритма Форд-Беллмана и Дейкстры видна очевидная закономерность (Беллман выполняется в n раз дольше). Однако, очень интересным является результат работы алгоритма Левита. Как оказалось, он даже немного быстрее алгоритма дейкстры (в среднем на константу, равную примерно 1.5-2).

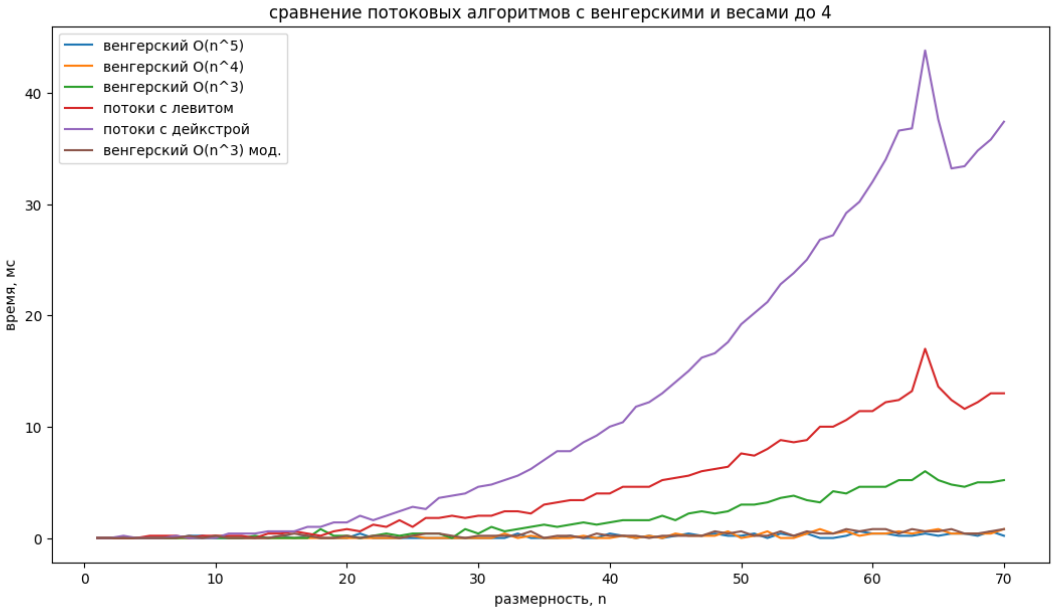
И тут крайне интресными оказались результаты сравнения Потоковых алгоритмов с венгерскими.



Оказалось, что даже венгерский алгоритм, работающий за $\mathcal{O}(n^4 \cdot m)$, работает зна-

чительно быстрее любого потокового алгоритма. Но после проверки времени работы на графах с разными развесовками (большой диапазон/маленький).





Из графиков видно (что можно легко показать и опираясь на код реализации), что потоки основанные на Дейкстре с потенциалами джонсона, не зависят от весов. В то же время видна зависимость у алгоритма Левита (при маленькой развесовке он работает чуть быстрее).

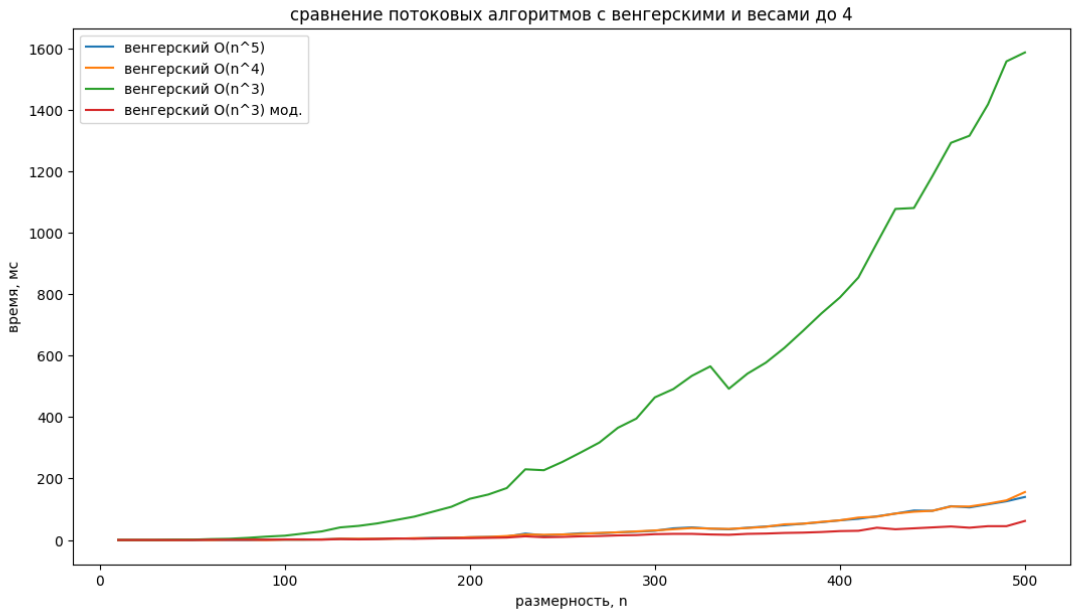
Но куда более интересными оказались резульаты сравнения только венгерских алгоритмов. Бросается в глаза тот факт, что $\mathcal{O}(n^3)$ без модификации очень сильно проигрывает

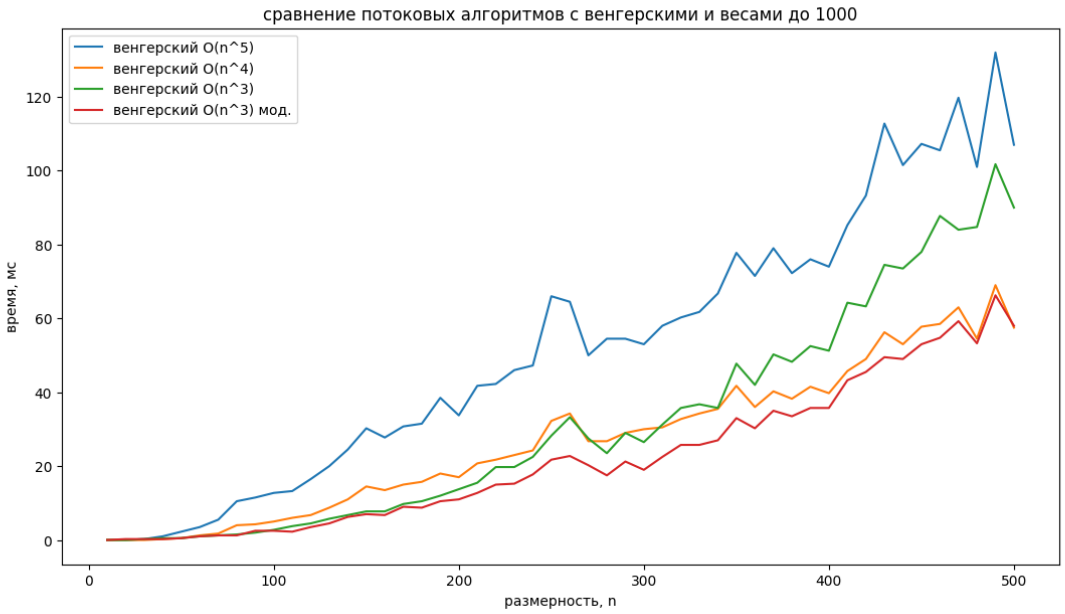
всем остальным реализациям. Собственно это есть одна из главных причин, добавления к нему модификации.

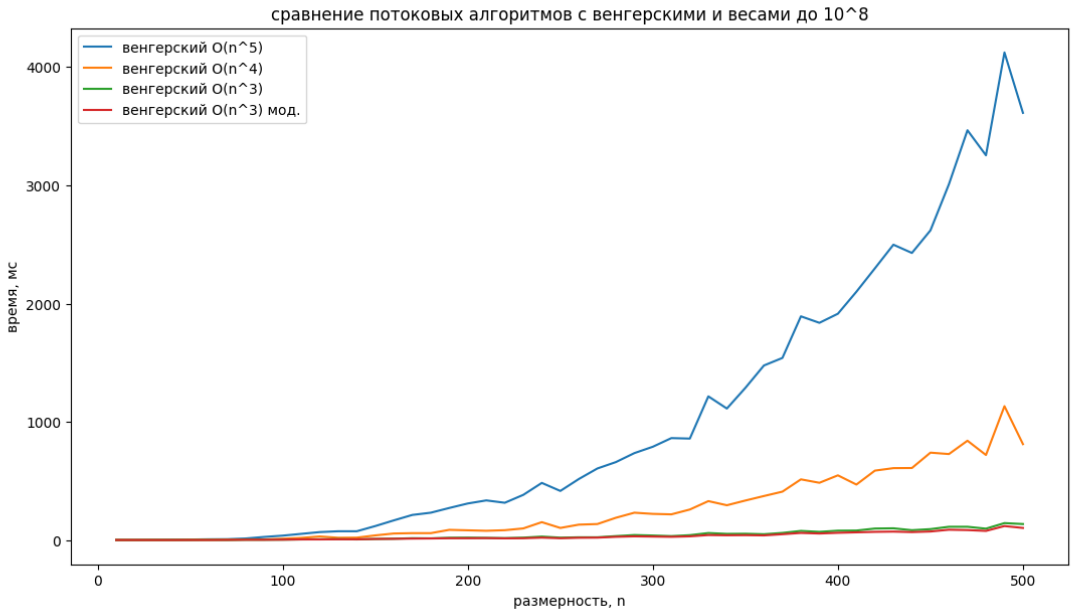
Дело в том, что асимтотическая оценка показывает время работы алгоритма в худшем случае, а не в среднем по случайным тестам. И вполне очевидным является тот факт, что главный цикл венгерского алгоритма (который есть в любой реализации, итерации которого состоят из двух рассмотренных ранее шагов) выполняет зачастую значительно меньше итераций, чем n^2. Отсюда и вытекают предпосылки для подобных расхождений.

Получается, что когда граф имеет маленький диапозон весов, алгоритмы имеющие более реальный обход Куна, вместо итеративного, с добавлением к рассмотрению не более 2 вершин, сильно выигравают, поскольку за несколько итераций добиваются результата (различных весов мало). Однако, это легко решается, добавлением в решение за $\mathcal{O}(n^3)$ выбора не произвольного столбца, а по возможности столбца не насыщенного паросочетанием. Ну и в качестве еще одной оптимизации, даже если мы не нашли ненасыщенного столбца, давайте добавим к рассмотрения не один столбец, а все которые получилось "занулить". Собственно это и есть краткое описание, отличия алгоритма с пометкой "мод."от обычного.

Ниже приведены графики для только потоковых алгоритмов.

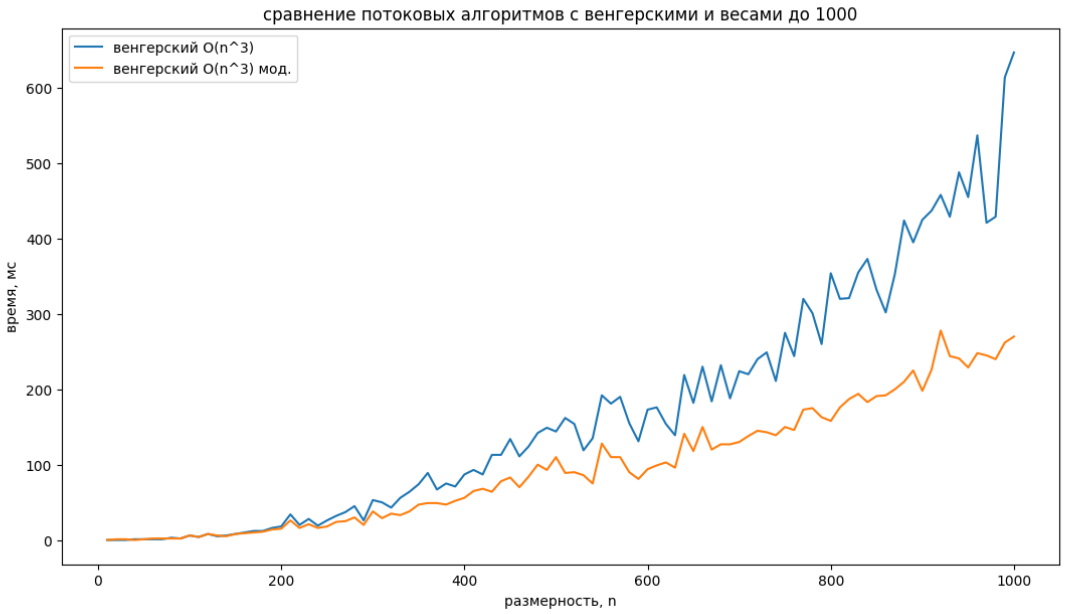


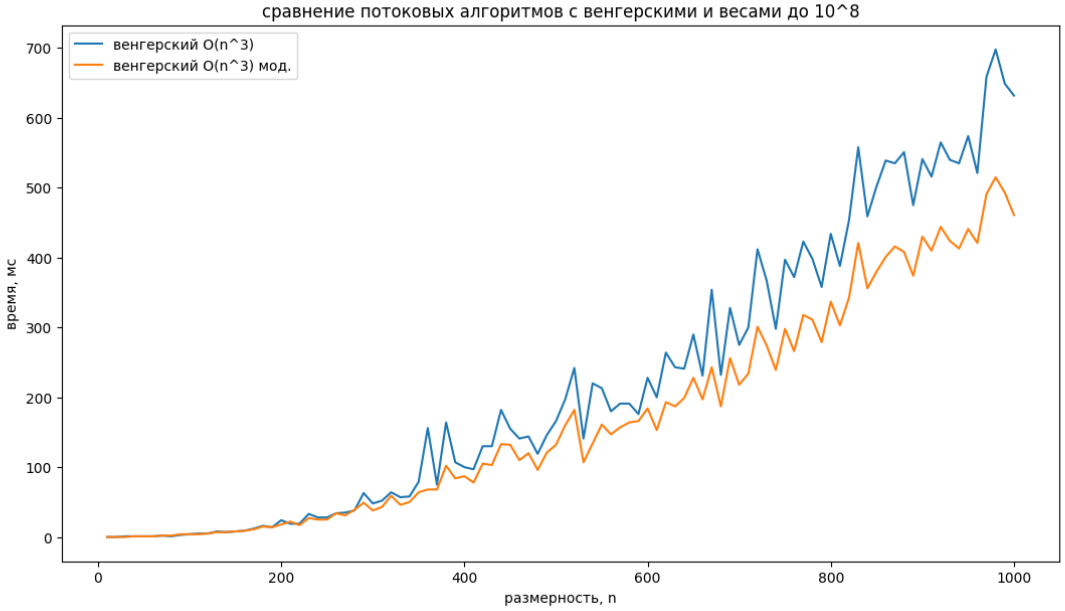




Видно, что при разреженной развесовке, алгоритмы более менее соответствуют заявленному асимптотическому времени работы.

Приведем так же сравнение только $\mathcal{O}(n^3)$





Было проведено так же тестрование для графов, где ребро с некоторой вероятностью имеет "бексонечный"вес, никаких серьезных закономерностей обнаружено не было.

7 Заключение

В рамках данной курсовой работы, я тщательно изучил Венгерский алгоритм, и решение задачи о назначениях с помощью поиска потока минимальной стоимости. Было выявлено, что потоки минимальной стоимости проигрывают любой приведенной здесь реализации венгерского алгоритма, даже имея асимптотически гораздо более долгое решение. Из чего следует очевидный вывод, о важности венгерского алгоритма.

Было установлено, что реализация поиска минимального потока, с помощью алгоритма Левита, является зачастую более предпочтительной, чем использование потенциалов Джонсона, хотя опять же, имеет более плохую асимптотическую оценку. А учитывая его относительную простоту, если все же нужно будет реализовать решение с помощью потоков минимальной стоимости, то скорее всего стоит отдать предпочтение алгоритму Левита.

Конечно же, важным аспектом является анализ времени работы относительно разреженности весов в графе, что позволило выявить изъян в реализации венгерского алгоритма за $\mathcal{O}(n^2 \cdot m)$. Изъян достаточно легко исправляется путем модифицированния алгоритма. Данным изъяном обладает так же например и реализация алгоритма приведенная на https://e-maxx.ru/algo/assignment$_h ungary$.

Еще одним важным выводом можно сделать то, что реализация за $\mathcal{O}(n^3 \cdot m)$ зачастую может работать не сильно хуже (при отнсительно не больших n и m).

8 Список использованных источников

1. https://algorithmica.org/ru/mincost-maxflow
2. https://neerc.ifmo.ru/wiki/index.php?title=Использование_потенциалов_Джонсона_при_
3. https://dic.academic.ru/dic.nsf/ruwiki/96142
4. https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Левита

Приложение А.

```
class Solve_base {
protected:
    const long long INF = 2e18;
    int n, m;
    vector<vector<long long>> a;

    long long answer = -1;
    long long delta = 0;
public:
    virtual void solve(vector<vector<long long>> b) = 0;

    void init (vector<vector<long long>>& b) {
        delta = 0;
        answer = -1;
        n = b.size();
        m = b[0].size();

        if (m < n){
            a = vector<vector<long long>> (m, vector<long long> (n));
            for (int i = 0; i < n; i++){
                for (int j = 0; j < m; j++){
                    a[j][i] = b[i][j];
                }
            }
            swap(n, m);
        }else{
            a = b;
        }

        for (int i = 0; i < n; i++){
            long long mn = INF;
            for (int j = 0; j < m; j++){
                mn = min(mn, a[i][j]);
            }
            delta += mn;
            for (int j = 0; j < m; j++){
                a[i][j] -= mn;
            }
        }
```

```
    }

    long long get_answer(){
        return answer + delta;
    }
};
```

Приложение B.

```
class Brute_force : public Solve_base {
protected:
    vector<int> id_right;

    void perm_right (vector<int> id_right) {
        do {
            long long value = 0;
            for (int i = 0; i < n; i++){
                value += a[i][id_right[i]];
            }
            answer = min(answer, value);
        } while (next_permutation(id_right.begin(), id_right.end()));
    }

    void gen_right (int i = 0) {
        if (m - i < n - id_right.size()){
            return;
        }

        if (id_right.size() == n){
            perm_right(id_right);
            return;
        }

        gen_right(i + 1);
        id_right.push_back(i);
        gen_right(i + 1);
        id_right.pop_back();
    }

public:
```

```
    void solve(vector<vector<long long>> b) override {
        init(b);
        id_right.clear();
        answer = INF;
        gen_right();
    }

    Brute_force () {

    }
};
```

Приложение C

```
class Min_cost_flow_base : public Solve_base {
protected:
    vector<vector<int>> g;
    vector<int> to;
    vector<int> cap;
    vector<long long> cost;

    int s, t;

    void add_edge (int u, int v, long long c) {
        g[u].push_back(to.size());
        to.push_back(v);
        cap.push_back(1);
        cost.push_back(c);

        g[v].push_back(to.size());
        to.push_back(u);
        cap.push_back(0);
        cost.push_back(-c);
    }

    void init(vector<vector<long long>> b) {
        Solve_base::init(b);
        g.clear();
        to.clear();
        cap.clear();
```

```
        cost.clear();

        answer = 0;
        s = n + m;
        t = s + 1;
        g.resize(t + 1);

        for (int i = 0; i < n; i++){
            add_edge(s, i, 0);
        }
        for (int i = 0; i < m; i++){
            add_edge(i + n, t, 0);
        }
        for (int i = 0; i < n; i++){
            for (int j = 0; j < m; j++){
                add_edge(i, j + n, a[i][j]);
            }
        }

    }
public:

};
```

Приложение D

```
class Min_cost_flow_bellman : public Min_cost_flow_base {
protected:

public:
    void solve(vector<vector<long long>> b) override {
        init(b);
        for (int _ = 0; _ < n; _++) {
            vector<long long> d(g.size(), INF);
            vector<int> r(g.size());
            d[s] = 0;

            for (int it = 1; it < g.size(); it++){
                for (int i = 0; i < to.size(); i++){
                    long long nd = d[to[i ^ 1]] + cost[i];
```

```
                    if (cap[i] && d[to[i]] > nd){
                        d[to[i]] = nd;
                        r[to[i]] = i;
                    }
                }
            }

            int cur = t;
            while (cur != s){
                int i = r[cur];
                answer += cost[i];
                cap[i]--;
                cap[i ^ 1]++;
                cur = to[i ^ 1];
            }
        }
    }

};
```

Приложение E

```
class Min_cost_flow_levit : public Min_cost_flow_base {
protected:

public:
    void solve(vector<vector<long long>> b) override {
        init(b);

        for (int _ = 0; _ < n; _++) {
            vector<long long> d(g.size(), INF);
            vector<int> r(g.size());

            vector<int> idv(g.size(), 2);
            queue<int> q2;
            queue<int> q1;

            idv[s] = 1;
            q1.push(s);
            d[s] = 0;
```

```
        while (q1.size() || q2.size()){
            int cur;
            if (q1.size()){
                cur = q1.front();
                q1.pop();
            }else{
                cur = q2.front();
                q2.pop();
            }

            for (int i : g[cur]){
                int next = to[i];
                long long nd = d[cur] + cost[i];
                if (!cap[i] || d[next] <= nd){
                    continue;
                }

                d[next] = nd;
                r[next] = i;

                if (idv[next] == 2){
                    q2.push(next);
                }else
                if (idv[next] == 0){
                    q1.push(next);
                }
                idv[next] = 1;
            }

            idv[cur] = 0;
        }

        int cur = t;
        while (cur != s){
            int i = r[cur];
            answer += cost[i];
            cap[i]--;
            cap[i ^ 1]++;
            cur = to[i ^ 1];
```

```
            }
        }
    }

};
```

Приложение F

```
class Min_cost_flow_dijkstra : public Min_cost_flow_base {
protected:

public:
    void solve(vector<vector<long long>> b) override {
        init(b);

        vector<long long> p(g.size(), INF);
        p[s] = 0;
        for (int it = 1; it < g.size(); it++){
            for (int i = 0; i < to.size(); i++){
                if (cap[i]){
                    p[to[i]] = min(p[to[i]], p[to[i ^ 1]] + cost[i]);
                }
            }
        }

        for (int _ = 0; _ < n; _++) {
            vector<long long> d(g.size(), INF);
            vector<int> r(g.size());
            vector<int> was(g.size());
            d[s] = 0;

            for (int _ = 0; _ < g.size(); _++){
                int cur = -1;
                for (int i = 0; i < g.size(); i++){
                    if (was[i]){
                        continue;
                    }
                    if (cur == -1 || d[cur] > d[i]){
                        cur = i;
                    }
```

```
                }
                was[cur] = 1;

                for (int i : g[cur]){
                    int next = to[i];
                    if (!cap[i] || was[next]){
                        continue;
                    }
                    long long nd = d[cur] + cost[i] + p[cur] - p[next];
                    if (d[next] > nd){
                        d[next] = nd;
                        r[next] = i;
                    }
                }
            }

            int cur = t;
            while (cur != s){
                int i = r[cur];
                answer += cost[i];
                cap[i]--;
                cap[i ^ 1]++;
                cur = to[i ^ 1];
            }

            for (int i = 0; i < g.size(); i++){
                p[i] = d[i] + p[i];
            }
        }
    }

};
```

Приложение G

```
class Veng_n_5 : public Solve_base {
protected:
    vector<int> lnei, rnei;
    vector<int> lwas, rwas;
    int cw;
```

```
bool kyn(int v){
    if (lwas[v] == cw){
        return false;
    }
    lwas[v] = cw;

    for (int u = 0; u < m; u++){
        if (a[v][u]){
            continue;
        }
        if (rwas[u] == cw){
            continue;
        }
        rwas[u] = cw;

        if (rnei[u] == -1 || kyn(rnei[u])){
            lnei[v] = u;
            rnei[u] = v;
            return true;
        }
    }

    return false;
}

void dfs(int v){
    if (lwas[v] == cw){
        return;
    }
    lwas[v] = cw;

    for (int u = 0; u < m; u++){
        if (a[v][u] || rwas[u] == cw || lnei[v] == u){
            continue;
        }

        rwas[u] = cw;
        if (rnei[u] != -1){
            dfs(rnei[u]);
```

```
            }
        }
    }

public:
    void solve (vector<vector<long long>> b) override {
        init(b);
        answer = 0;
        cw = 0;

        while (true) {
            lnei = vector<int> (n, -1);
            rnei = vector<int> (m, -1);
            lwas = vector<int> (n);
            rwas = vector<int> (m);

            int cnt = 0;
            for (int v = 0; v < n; v++){
                cw++;
                if (kyn(v)){
                    cnt++;
                }
            }

            if (cnt == n){
                break;
            }

            cw++;
            for (int i = 0; i < n; i++){
                if (lnei[i] == -1){
                    dfs(i);
                }
            }

            long long mn = INF;
            for (int i = 0; i < n; i++){
                if (lwas[i] == cw){
                    for (int j = 0; j < m; j++){
                        if (rwas[j] != cw){
```

```
                        mn = min(mn, a[i][j]);
                    }
                }
            }
        }

        for (int i = 0; i < n; i++){
            if (lwas[i] == cw){
                answer += mn;
                for (int j = 0; j < m; j++){
                    a[i][j] -= mn;
                }
            }
        }
        for (int j = 0; j < m; j++){
            if (rwas[j] == cw){
                answer -= mn;
                for (int i = 0; i < n; i++){
                    a[i][j] += mn;
                }
            }
        }
    }
};
```

Приложение H

```
class Veng_n_4 : public Solve_base {
protected:
    vector<int> lnei, rnei;
    vector<int> lwas, rwas;
    int cw;

    bool kyn(int v){
        if (lwas[v] == cw){
            return false;
        }
        lwas[v] = cw;
```

```
        for (int u = 0; u < m; u++){
            if (a[v][u]){
                continue;
            }
            if (rwas[u] == cw){
                continue;
            }
            rwas[u] = cw;

            if (rnei[u] == -1 || kyn(rnei[u])){
                lnei[v] = u;
                rnei[u] = v;
                return true;
            }
        }

        return false;
    }

public:
    void solve (vector<vector<long long>> b) override {
        init(b);
        answer = 0;
        cw = 0;

        lnei = vector<int> (n, -1);
        rnei = vector<int> (m, -1);
        lwas = vector<int> (n);
        rwas = vector<int> (m);

        for (int v = 0; v < n; v++){
            while (true) {
                if (lnei[v] != -1){
                    break;
                }

                cw++;
                if (kyn(v)){
                    break;
                }
```

```
                long long mn = INF;
                for (int i = 0; i < n; i++){
                    if (lwas[i] == cw){
                        for (int j = 0; j < m; j++){
                            if (rwas[j] != cw){
                                mn = min(mn, a[i][j]);
                            }
                        }
                    }
                }

                for (int i = 0; i < n; i++){
                    if (lwas[i] == cw){
                        answer += mn;
                        for (int j = 0; j < m; j++){
                            a[i][j] -= mn;
                        }
                    }
                }
                for (int j = 0; j < m; j++){
                    if (rwas[j] == cw){
                        answer -= mn;
                        for (int i = 0; i < n; i++){
                            a[i][j] += mn;
                        }
                    }
                }
            }
        }
    }
};
```

Приложение I

```
class Veng_n_3 : public Solve_base {
protected:
    vector<int> lnei, rnei;
    vector<int> lwas, rwas;
    vector<int> r_from;
```

```
    int cw = 0;
    vector<long long> lp, rp;
    vector<long long> mnv;

public:
    void solve (vector<vector<long long>> b) override {
        init(b);
        answer = 0;
        cw = 0;
        lnei = vector<int> (n, -1);
        rnei = vector<int> (m, -1);
        lwas = vector<int> (n);
        rwas = vector<int> (m);
        lp = vector<long long> (n);
        rp = vector<long long> (m);

        for (int v = 0; v < n; v++){
            if (lnei[v] != -1){
                continue;
            }

            r_from = vector<int> (m, -1);
            cw++;
            lwas[v] = cw;

            mnv = vector<long long> (m);
            for (int j = 0; j < m; j++){
                mnv[j] = a[v][j] - lp[v] - rp[j];
            }

            while (true){
                long long mn = INF;
                int imn;
                for (int j = 0; j < m; j++){
                    if (rwas[j] != cw){
                        if (mn > mnv[j]){
                            imn = j;
                            mn = mnv[j];
                        }
                    }
```

```
        }

        for (int i = 0; i < n; i++){
            if (lwas[i] == cw){
                answer += mn;
                lp[i] += mn;
            }
        }
        for (int j = 0; j < m; j++){
            if (rwas[j] == cw){
                answer -= mn;
                rp[j] -= mn;
            }
            mnv[j] -= mn;
        }

        int pos = imn;
        int link;
        for (int i = 0; i < n; i++){
            if (lwas[i] == cw && a[i][pos] - lp[i] - rp[pos] == 0){
                link = i;
                break;
            }
        }

        r_from[pos] = link;

        if (rnei[pos] == -1){
            while (true){
                int next = lnei[r_from[pos]];
                rnei[pos] = r_from[pos];
                lnei[r_from[pos]] = pos;
                if (r_from[pos] == v){
                    break;
                }
                pos = next;
            }

            break;
        }
```

```
                int sop = rnei[pos];
                rwas[pos] = cw;
                lwas[sop] = cw;

                for (int j = 0; j < m; j++){
                    mnv[j] = min(mnv[j], a[sop][j] - lp[sop] - rp[j]);
                }
            }
        }
    }
};
```

Приложение J

```
class Veng_opt : public Solve_base {
protected:
    vector<int> lnei, rnei;
    vector<int> lwas, rwas;
    vector<int> r_from;
    int cw = 0;
    vector<long long> lp, rp;
    vector<long long> mnv;

public:
    void solve (vector<vector<long long>> b) override {
        init(b);
        answer = 0;
        cw = 0;
        lnei = vector<int> (n, -1);
        rnei = vector<int> (m, -1);
        lwas = vector<int> (n);
        rwas = vector<int> (m);
        lp = vector<long long> (n);
        rp = vector<long long> (m);

        for (int v = 0; v < n; v++){
            if (lnei[v] != -1){
                continue;
            }
```

```
r_from = vector<int> (m, -1);
cw++;
lwas[v] = cw;

mnv = vector<long long> (m);
for (int j = 0; j < m; j++){
    mnv[j] = a[v][j] - lp[v] - rp[j];
}

while (true){
    long long mn = INF;
    vector<int> imn;
    for (int j = 0; j < m; j++){
        if (rwas[j] != cw){
            if (mn > mnv[j]){
                mn = mnv[j];
                imn.clear();
            }
            if (mn == mnv[j]){
                imn.push_back(j);
            }
        }
    }

    for (int i = 0; i < n; i++){
        if (lwas[i] == cw){
            answer += mn;
            lp[i] += mn;
        }
    }
    for (int j = 0; j < m; j++){
        if (rwas[j] == cw){
            answer -= mn;
            rp[j] -= mn;
        }
        mnv[j] -= mn;
    }

    int pos = imn.back();
```

```
                for (int j : imn){
                    if (rnei[j] == -1){
                        pos = j;
                    }
                    for (int i = 0; i < n; i++){
                        if (lwas[i] == cw && a[i][j] - lp[i] - rp[j] == 0){
                            r_from[j] = i;
                            break;
                        }
                    }
                }

                if (rnei[pos] == -1){
                    while (true){
                        int next = lnei[r_from[pos]];
                        rnei[pos] = r_from[pos];
                        lnei[r_from[pos]] = pos;
                        if (r_from[pos] == v){
                            break;
                        }
                        pos = next;
                    }

                    break;
                }

                for (int pos : imn){
                    int sop = rnei[pos];
                    rwas[pos] = cw;
                    lwas[sop] = cw;
                    for (int j = 0; j < m; j++){
                        mnv[j] = min(mnv[j], a[sop][j] - lp[sop] - rp[j]);
                    }
                }
            }
        }
    }
};
```

Приложение К.

```
class Gen_random_nofull : public Gen_random_base {
protected:
    long long ch, zn;
    long long big;
public:
    Gen_random_nofull (long long ch, long long zn, long long big,
        int sn, int bn, int sm, int bm, int sc, int bc) :
        Gen_random_base(sn, bn, sm, bm, sc, bc), ch(ch), zn(zn), big(big)
    {

    }

    Gen_random_nofull (long long ch, long long zn, long long big,
        int n, int m, int bc) :
        Gen_random_base(n, m, bc), ch(ch), zn(zn), big(big)
    {

    }

    Gen_random_nofull (long long ch, long long zn, long long big,
        int n, int bc) :
        Gen_random_base(n, bc), ch(ch), zn(zn), big(big)
    {

    }

    bool next (vector<vector<long long>>& a) override {
        int n = rand(sn, bn);
        int m = rand(sm, bm);
        a.resize(n, vector<long long> (m));

        for (int i = 0; i < n; i++){
            for (int j = 0; j < m; j++){
                if (rand(zn) <= ch){
                    a[i][j] = rand(sc, bc);
                }else{
                    a[i][j] = big;
                }
            }
```

```
        }

        return false;
    }

};
```