

Министерство образования Российской Федерации
Нижегородский государственный университет имени Н.И. Лобачевского
Институт информационных технологий, математики и механики

Отчёт
по Лабораторной работе №10
Поиск остова графа максимального веса

Выполнили:
Студенты группы 381506-3
Панов А. А.
Сергеев А. П.
Проверил:
Ассистент
Грибанов Д. В.

Нижний Новгород
2017

Содержание

Введение	3
Постановка задачи	4
Описание алгоритмов	5
Алгоритм Краскала	5
Алгоритм Прима	9
Разделённые множества	11
Реализация на массиве	11
Реализация на дереве	12
Реализация на дереве с использованием оптимизации по рангу	12
Реализация на дереве с полной оптимизацией	13
Руководство пользователя	14
Руководство по запуску	14
Руководство по использованию	16
Руководство программиста	17
Структура проекта	17
Проект algs	18
Алгоритм Прима	19
Проект disjoint_sets	20
Проект graphs	21
Проект view	22
Проект test	23
Вывод	24
Результаты вычислительного эксперимента	24
Список литературы	29
Приложение	30
Код программы	30
alg_kraskal.h	30

Введение

Пусть $G = (V, E)$ - неориентированный случайный граф, для которого задана функция стоимости, отображающая ребра в целые(вещественные) числа. Поиск остовного дерева максимального веса – задача рассматриваемая в данной лабораторной работе.

Модели графов, в которых с каждым ребром связаны веса или стоимости, имеют обширную область применения:

- В картах авиалиний. Ребрами отмечаются авиарейсы и веса означают расстояния или стоимости билетов. Обычно требуется найти перелёт с минимальным расстоянием или минимальной стоимостью.
- В электронных схемах. Ребра представляют проводники, веса могут означать длину проводника, его стоимость или время прохода сигнала. Требуется спроектировать схему с минимальным временем прохода сигнала, стоимости или длины.
- Разработка сетей. Ребрами обозначается сеть между городами, а вес её длину или стоимость. Для этой области характерна задача: соединения n городов в единую телефонную сеть с минимальной суммарной стоимостью соединений.
- Наука, и в частности биология, используют многомерные данные для группировки объектов, растений, животных. Минимальное остовное дерево позволяет разбивать их на взаимосвязанные классы, четко отслеживая близкие по строению и характеристикам группы.

Существует несколько алгоритмов для нахождения минимального остовного дерева. Наиболее известными среди них являются:

- Алгоритм Прима
- Алгоритм Краскала
- Алгоритм Борувки

Именно их мы и рассмотрим в данной лабораторной работе.

Стоит отметить, что для нахождения максимального остова существует двойственная задача - задача нахождения минимального остова. При этом в рамках рассматриваемых алгоритмов решения, переход от одной задачи к другой очень простой, достаточно умножить веса всех ребер на -1.

Постановка задачи

Цель данной лабораторной работы – написать программу, в которой будут сравниваться разные алгоритмы поиска остоного подграфа максимального/минимального веса для некоторого взвешенного графа с использованием алгоритмов Краскала, Борушки и Прима. Реализовать структуру данных разделенное множество на нескольких вариантах:

- 1) разделенное множество на массивах.
- 2) разделенное множество на древовидной структуре с использованием рангов узлов.
- 3) предыдущий вариант с использованием эвристики сжатия путей.

Описание алгоритмов

Алгоритм Краскала

Алгоритм Краскала изначально помещает каждую вершину в отдельное множество, а затем постепенно объединяет эти множества, объединяя на каждой итерации два некоторых непересекающихся множества некоторым ребром изначального графа G .

Перед началом выполнения алгоритма, все рёбра сортируются по весу в порядке возрастания(неубывания). Затем начинается процесс объединения: перебираются все рёбра от первого до последнего, и если у текущего ребра его концы принадлежат разным множествам, то эти множества объединяются, а ребро добавляется к ответу. По окончании перебора всех рёбер – все вершины будут принадлежать одному множеству.

Рёбра, которые получаются в ответе, составят остов подграфа с максимальным(минимальным) весом.

На следующих рисунках рис. 1 – рис. 5 представлена работа алгоритма.

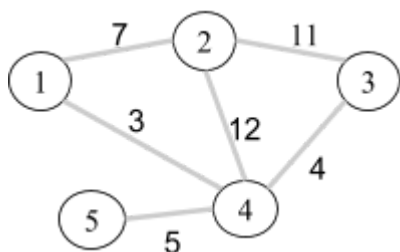


Рис.1 Изначальный граф.

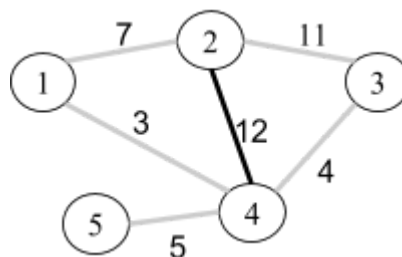


Рис.2 Добавление ребра (2,4) к ответу.

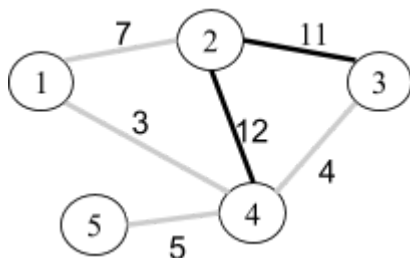


Рис. 3 Добавление ребра (2,3) .

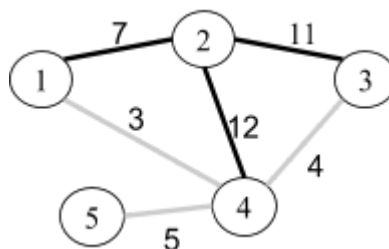


Рис. 4 Добавление ребра (1,2).

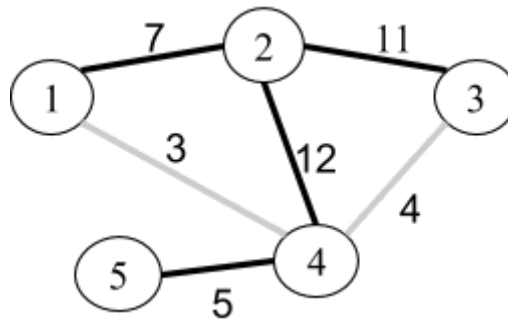


Рис. 5 Добавление ребра (4,5). Получившийся остовный подграф.

Сложность сортировки m ребер у графа из n вершин $O(m \ln(n))$, если для работы с множествами использовать “систему с непересекающимися множествами”, то можно получить сложность почти $O(m \ln(n))$.

Алгоритм Борувки

Работа алгоритма состоит из нескольких итераций, каждая из которых состоит в последовательном добавлении рёбер к подграфу \bar{G} , до тех пор, пока \bar{G} не будет иметь 1 компоненту связности.

В псевдокоде, алгоритм можно описать так:

1. Изначально, пусть \bar{G} — пустой граф подграф связного графа G с n вершинами и n компонентами связности.
2. Пока \bar{G} имеет больше 1 компоненты связности:
 - Для каждой компоненты связности в подграфе \bar{G} , найдём ребро максимального веса, связывающее эту компоненту с некоторой другой компонентой связности.
 - Добавим все найденные рёбра в подграф \bar{G} .
3. Полученный граф \bar{G} является остовным деревом максимального веса для входного графа G .

На рис. 6 – рис. 10 продемонстрирована работа алгоритма Борувки.

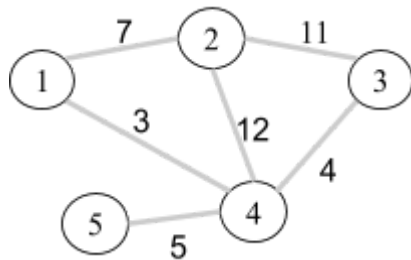


Рис. 6 Изначальный граф.

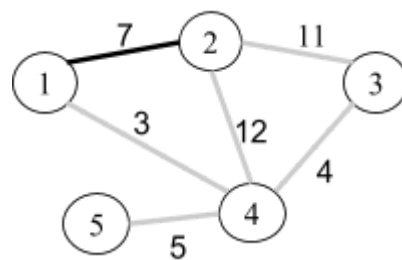


Рис. 7 Добавление ребра (1,2).

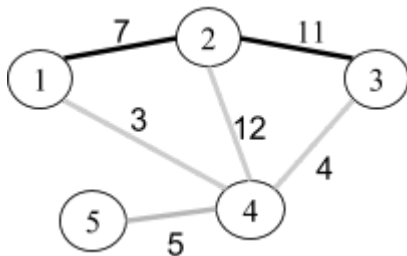


Рис. 8 Добавление ребра (2,3).

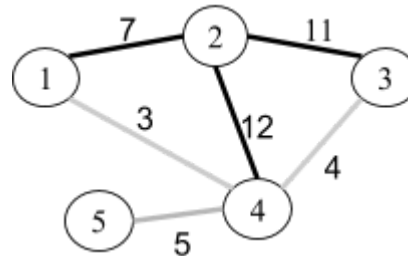


Рис. 9 Добавление ребра (2,4).

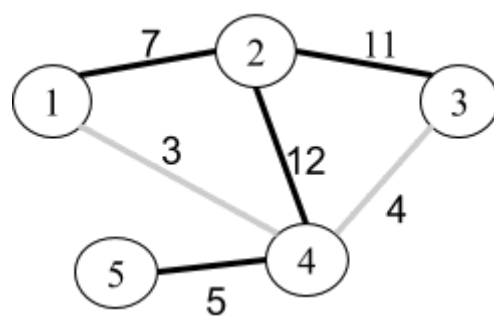


Рис. 10 Добавление ребра (4,5). Получившийся граф \overline{G} .

Алгоритм Прима

Сам алгоритм имеет очень простой вид. Искомый минимальный остов строится постепенно, добавлением в него рёбер по одному.

Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно). Затем выбирается ребро максимального веса, исходящее из этой вершины, и добавляется в остов максимального веса. После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро максимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется максимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины.

В итоге будет построен остов, являющийся максимальным. Если граф был изначально не связан, то остов найден не будет.

На следующих рисунках показан его алгоритм работы.

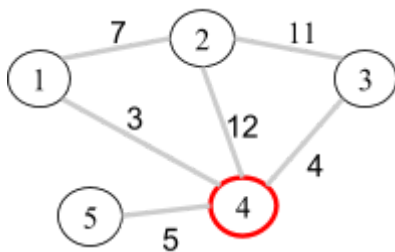


Рис. 11 Выбор ведущей вершины (4).

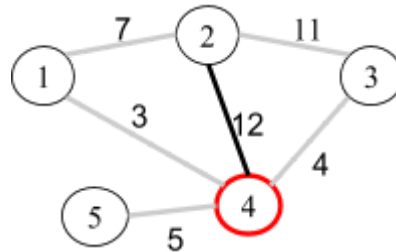


Рис. 12 Добавление ребра (2,4).

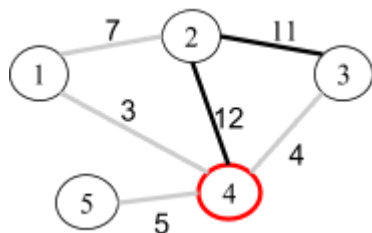


Рис. 13 Добавление ребра (2,3).

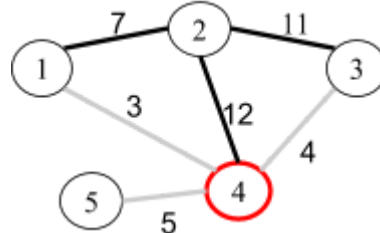


Рис. 14 Добавление ребра (1,2).

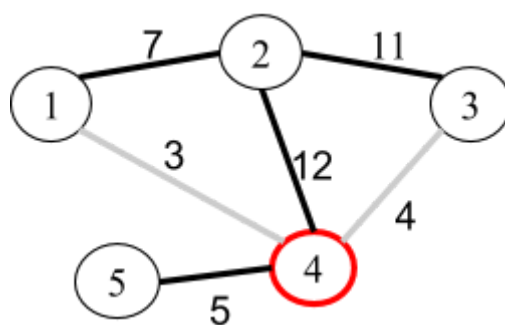


Рис. 15 Добавление ребра (4,5). Получившийся остов.

Разделённые множества

Разделенные множества – это абстрактный тип данных, предназначенный для представления коллекции, состоящего из некоторого числа k попарно непересекающихся подмножеств U_1, U_2, \dots, U_k заданного множества U . Далее в качестве U будем рассматривать множество $\{1, 2, \dots, n\}$.

Этот тип данных применяется в таких задачах, как отыскание минимального остовного дерева для заданного взвешенного неориентированного графа, построение компонент связности графа, и многих других, требующих динамического поддержания некоторого отношения эквивалентности.

Для разделённого множества реализуются 3 главные операции:

1. **make_set(x)** — добавляет новый элемент x , помещая его в новое множество, состоящее из него одного.
2. **union_sets(x, y)** — объединяет два указанных множества (множество, в котором находится элемент x , и множество, в котором находится элемент y).
3. **find_set(x)** — возвращает, в каком множестве находится указанный элемент x . На самом деле при этом возвращается один из элементов множества, называемый представителем. Этот представитель выбирается в каждом множестве самой структурой данных (и может меняться с течением времени, а именно, после вызовов `union_sets()`).

Реализация на массиве

Пусть $U = \{1, 2, \dots, n\}$ – множество, из элементов которого будет строиться коллекция разделенных подмножеств.

Одним из очевидных способов представления коллекции является представление ее с помощью массива. При таком способе для каждого элемента i в соответствующей i -й ячейке массива помещаем имя того подмножества, которому принадлежит элемент i .

Обозначим через f массив длины n , с помощью которого будет представлена коллекция.

Реализация операций с помощью массива:

1. **make_set**(x) — осуществляется записью элемента x в ячейку с номером x .
2. **union_sets**(x, y) — осуществляется следующим образом. Просматриваются элементы массива f , и в те ячейки, в которых было записано значение x , заносится новое значение — y .
3. **find_set**(x) — возвращает значение находящееся в массиве f на позиции x .

Реализация на дереве

В данной реализации множества элементы будут храниться в виде деревьев: одно дерево соответствует одному множеству. Корень дерева — это представитель множества.

При реализации это означает, что создаётся массив *parent* [], в котором для каждого элемента хранится ссылка на его предка в дереве. Для корней деревьев будем считать, что их предок — они сами.

Итак, вся информация о множествах элементов хранится с помощью массива *parent* [].

Реализация операций:

1. **make_set**(x) — для создания нового элемента, просто создаётся дерево с корнем в вершине x , отмечая, что её предок — это она сама.
2. **union_sets**(x, y) — чтобы объединить два множества, сначала ищутся лидеры множества, в котором находится x , и множества, в котором находится y . Если лидеры совпали, то ничего не нужно делать — это значит, что множества и так уже были объединены. В противном случае указывается, что предок вершины y равен x — тем самым присоединив одно дерево к другому.
3. **find_set**(x) — поиска лидера в данной реализации сводится к одному действию: подняться по предкам от вершины x , пока не достигнем корня.

Реализация на дереве с использованием оптимизации по рангу

Данная оптимизация заключается в небольшом изменении работы `union_sets`: если в реализации на дерево, какое множество будет присоединено к какому, определено константно: присоединение к x , то теперь решение будет приниматься на основе рангов.

Есть два варианта ранговой эвристики: в одном варианте рангом дерева называется количество вершин в нём, в другом — глубина дерева .

В обоих вариантах суть оптимизации одна и та же: при выполнении `union_sets` будем присоединять дерево с меньшим рангом к дереву с большим рангом.

Реализация на дереве с полной оптимизацией

Полная оптимизация разделённого множества представляет из себя оптимизацию по рангу и оптимизацию сжатия пути объединённые вместе.

Так как оптимизацию по рангу уже была рассмотрена для данной реализации будет достаточно рассмотреть метод сжатия пути.

Эвристика сжатия пути предназначена для ускорения работы `find_set(x)`. Она заключается в том, что когда после вызова `find_set(x)` мы найдём искомого лидера P множества, то запомним, что у вершины x и всех пройденных по пути вершин — именно этот лидер P . Проще всего это сделать, перенаправив их `parent[]` на эту вершину P .

Таким образом, у массива предков `parent[]` смысл несколько меняется: теперь это сжатый массив предков, т.е. для каждой вершины там может храниться не непосредственный предок, а предок предка, предок предка предка, и т.д.

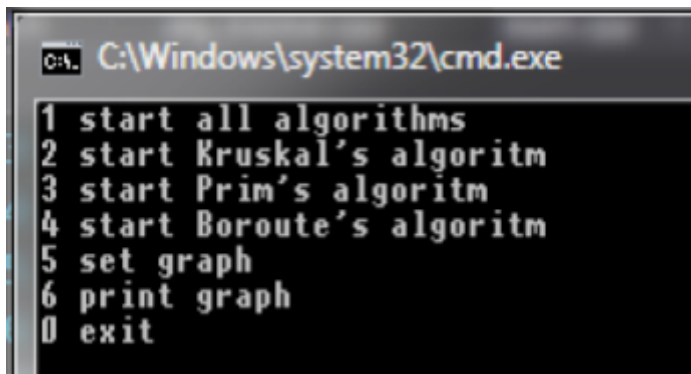
С другой стороны, понятно, что нельзя сделать, чтобы эти указатели `parent[]` всегда указывали на лидера: иначе при выполнении операции `union_sets()` пришлось бы обновлять лидеров у $O(n)$ элементов.

Таким образом, к массиву `parent[]` следует подходить именно как к массиву предков, возможно, частично сжатому.

Руководство пользователя

Руководство по запуску

При запуске программы появляется следующее меню:



```
C:\Windows\system32\cmd.exe
1 start all algorithms
2 start Kruskal's algorithm
3 start Prim's algorithm
4 start Boroute's algorithm
5 set graph
6 print graph
0 exit
```

рис. 16 Меню

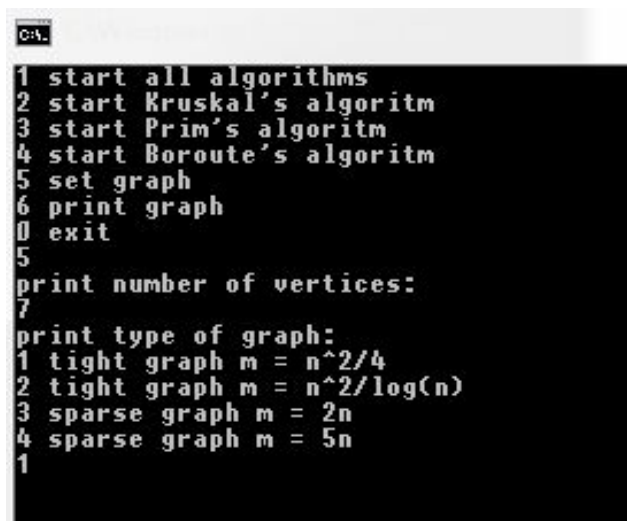
Для тестирования программы необходимо задать граф с помощью пункта меню 5. Затем можно запустить интересующие алгоритмы и получить информацию о времени их работы, а также о минимальном остоле. Для вывода исходного графа нужно воспользоваться 6 пунктом меню.

Вывод осуществляется в формате:

вершина V1, вершина V2, вес ребра (V1, V2)

Рассмотрим пример запуска:

1. Зададим случайный граф:



```
C:\Windows\system32\cmd.exe
1 start all algorithms
2 start Kruskal's algorithm
3 start Prim's algorithm
4 start Boroute's algorithm
5 set graph
6 print graph
0 exit
5
print number of vertices:
7
print type of graph:
1 tight graph m = n^2/4
2 tight graph m = n^2/log(n)
3 sparse graph m = 2n
4 sparse graph m = 5n
1
```

2. Выведем его:

```
number of vertices 7 number of edges 12
1 start all algorithms
2 start Kruskal's algoritm
3 start Prim's algoritm
4 start Boroute's algoritm
5 set graph
6 print graph
0 exit
6
0 1 394
0 3 519
0 5 291
1 2 916
1 3 857
1 4 368
1 6 129
3 4 396
3 6 438
5 4 162
5 2 551
2 6 510
```

3. Запустим алгоритм Прима:

```
number of vertices 7 number of edges 12
1 start all algorithms
2 start Kruskal's algoritm
3 start Prim's algoritm
4 start Boroute's algoritm
5 set graph
6 print graph
0 exit
3
number of connected components = 1
running Prim's algoritm
final Prim's algoritm, time = 0 , weight = 1856
print minimal skeleton?
1 yes
2 no
1
5 0 291
4 5 162
1 4 368
6 1 129
3 4 396
2 6 510
```

4. Запустим на всех алгоритмах и сравним результаты:

```
number of vertices 7 number of edges 12
1 start all algorithms
2 start Kruskal's algoritm
3 start Prim's algoritm
4 start Boroute's algoritm
5 set graph
6 print graph
0 exit
1
number of connected components = 1
running Boroute's algoritm
final Boroute's algoritm, time = 0 , weight = 1856
running Prim's algoritm
final Prim's algoritm, time = 0 , weight = 1856
running Kruskal's algoritm
final Kruskal's algoritm, time = 0 , weight = 1856
```

Руководство по использованию

Основные алгоритмы содержатся в различных статических библиотеках:

- Алгоритмы поиска min/max остова, библиотека `algs`
- Система непересекающихся множеств, библиотека `disjoint_sets`
- Алгоритмы работы с графами, библиотека `graphs`

Чтобы их использовать достаточно подключить необходимую библиотеку к своему проекту.

Руководство программиста

Структура проекта

Решение состоит из 6 проектов:

- Алгоритмы поиска min/max остова, библиотека `algs`
- Система непересекающихся множеств, библиотека `disjoint_sets`
- Алгоритмы работы с графами, библиотека `graphs`
- Демонстрационный проект `view`
- Проект с тестами, для проверки корректности алгоритмов и структур `test`
- Вспомогательный проект `gtest`

Проект algs

Перед применением какого либо алгоритма граф разбивается на компоненты связности, с помощью функции `getConnectedComponent()`

Входными данными является случайный граф, заданный списком ребер с весами, выходными данными является вектор списков ребер, каждый из которых принадлежит одной компоненте связности.

Алгоритм работы:

1. Из списка ребер создаем список смежности, функция `setAdjacencyList()`
2. Запускаем поиск в ширину и находим вершины принадлежащие одной компоненте
3. Добавляем в ответ все ребра принадлежащие этой компоненте

В проекте реализованы три алгоритма поиска min/max остова графа:

- Алгоритм Прима
- Алгоритм Краскала
- Алгоритм Борувки

Результатом работы каждого алгоритма будет вектор ребер (в формате вершина 1, вершина 2, вес ребра(вершина 1, вершина 2)) принадлежащих минимальному остову. Результат работы соответствует типу данных `std::vector<std::pair<int, std::pair<int, int>>>`, но для удобства объявлен синоним (через `typedef`) `EdgesWeight`

Ниже представлено описание конкретной реализации алгоритма Прима и алгоритма Борувки, все остальные алгоритмы соответствуют описанию на стр. 5

Алгоритм Прима

Представлен функцией `algPrima`, в качестве входных данных будем использовать матрицу смежности. Особенностью данной реализации является асимптотика $O(n^2)$.

Определим способ поиска наименьшего ребра: для каждой ещё не выбранной вершины будем хранить минимальное ребро, ведущее в уже выбранную вершину.

Тогда, чтобы на текущем шаге произвести выбор минимального ребра, надо просто просмотреть эти минимальные рёбра у каждой невыбранной ещё вершины — асимптотика составит $O(n)$.

Но теперь при добавлении в остов очередного ребра и вершины эти указатели надо пересчитывать. Эту фазу можно сделать также за $O(n)$.

Таким образом, получается вариант алгоритма Прима с асимптотикой $O(n^2)$.

Алгоритм Борувки

Изначально имеется n независимых множеств состоящих из одной вершины.

Пока число множеств не равно одному:

просматриваем все ребра (a,b) принадлежащие исходному графу

а. если a принадлежит множеству A , b принадлежит множеству B , A не равно B :

i. если вес (a,b) меньше веса $\text{minEdges}[A]$, то $\text{minEdges}[A] = (a,b)$

ii. если вес (a,b) меньше веса $\text{minEdges}[B]$, то $\text{minEdges}[B] = (a,b)$

2. просматриваем все ребра (a,b) принадлежащие minEdges

а. если a и b принадлежат разным множествам A и B

i. объединяем A и B

ii. $\text{minEdges}[A] = \text{maxEdge}$

iii. $\text{minEdges}[B] = \text{maxEdge}$

С каждым шагом внешнего цикла кол-во множеств будет уменьшаться в два раза. Обход всех ребер происходит за $O(m)$ шагов. Если реализовать поиск и объединение множеств за $O(1)$ сложность будет $O(m \lg n)$

Проект disjoint_sets

В проекте реализованы несколько вариантов системы непересекающихся множеств с использованием:

- Массива
- Деревя, с оптимизацией по рангу
- Деревя, с оптимизацией по рангу и сжатием пути

Их реализация совпадает с описанием на стр. 12.

Проект graphs

В проекте реализован метод создания случайного графа, методы преобразования графа из одного представления в другое, а также способы его вывода.

- `getRandomGraph(int size, int numEdges, int minWeight, int maxWeight)`
- `findWeight()`
- `getAdjacencyMatrixFromEdges()`
- `printEdges()`

Алгоритм формирования случайного графа:

1. Генерируется ребра полного графа со случайным весом от `minWeight` до `maxWeight`
2. Из них случайным образом выбирается ребро, удаляется из графа, а затем добавляется в ответ
3. Шаг 2 повторяется `numEdges` раз

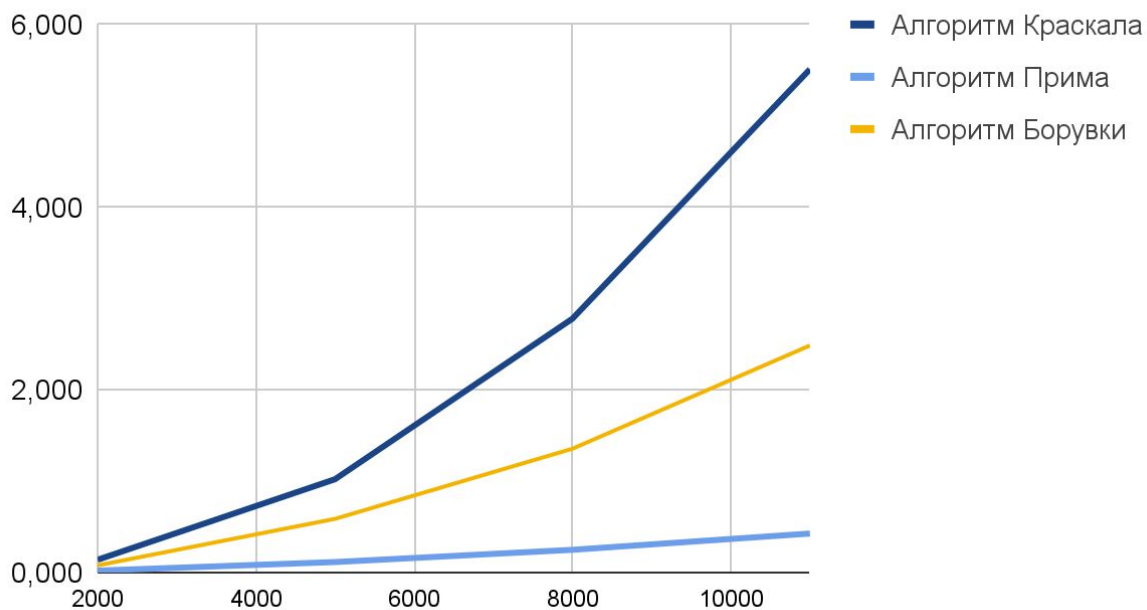
Вывод

Результаты вычислительного эксперимента

Для плотных графов (График 1), $m = \frac{n^2}{4}$:

Число вершин	Число ребер	Алгоритм Краскала, сек	Алгоритм Прима, сек	Алгоритм Борушки, сек
2000	1000000	0.14	0.021	0.078
5000	6250000	1.022	0.115	0.588
8000	16000000	2.78	0.249	1.356
11000	30250000	5.51	0.427	2.486

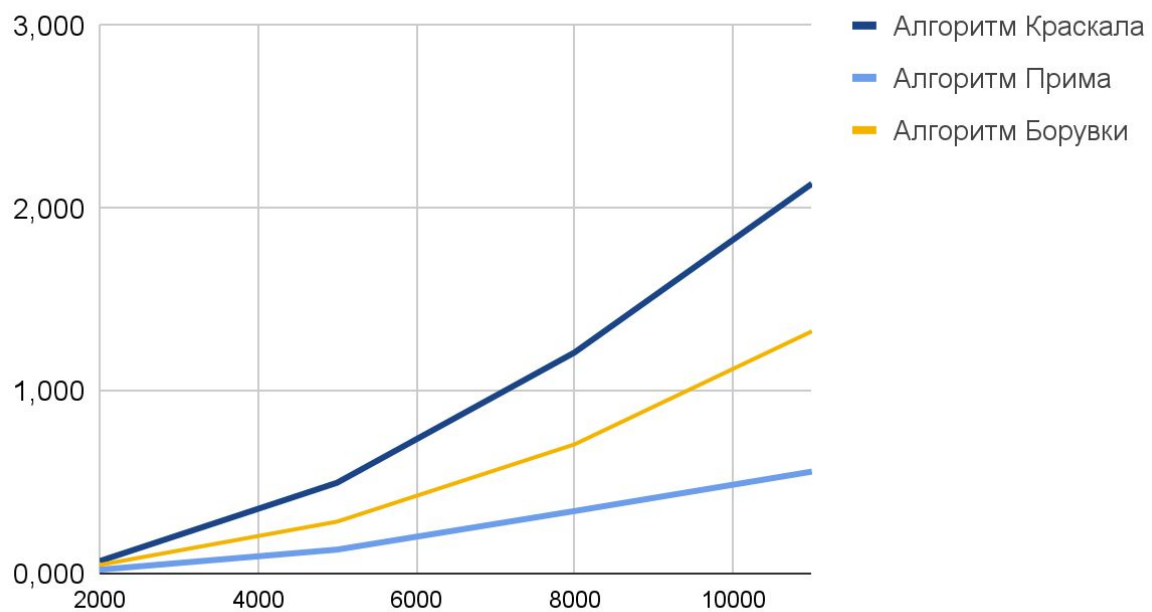
График 1



Для плотных графов (График 2), $m = \frac{n^2}{\ln(n)}$:

Число вершин	Число ребер	Алгоритм Краскала, сек	Алгоритм Прима, сек	Алгоритм Борувки, сек
2000	526253	0.068	0.021	0.047
5000	2935239	0.497	0.131	0.284
8000	7121241	1.21	0.342	0.707
11000	30250000	2.134	0.558	1.326

График 2

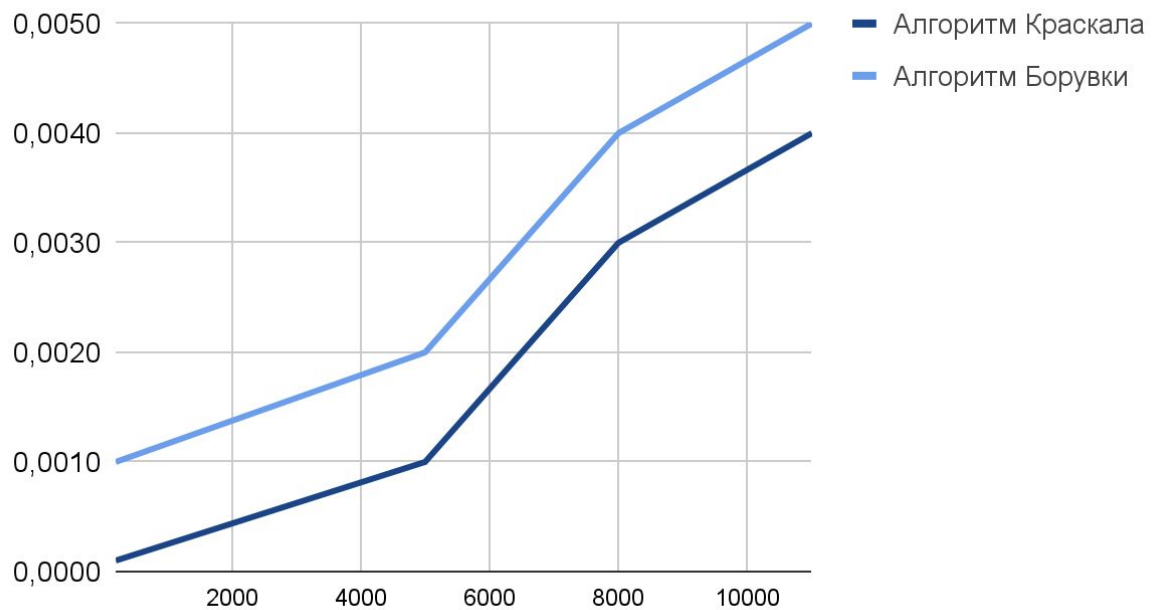


На плотных графах лучше всего себя показывает алгоритм Прима, так как имеет сложность $O(n^2)$, другие же алгоритмы имеют сложность $O(M \lg N)$, что в данном случае почти эквивалентно $O(N^2 \lg N)$.

Для разреженных графов (график 3), $m = 2n$:

Число вершин	Число ребер	Алгоритм Краскала, сек	Алгоритм Прима, сек	Алгоритм Борувки, сек
2000	4000	0.0001	0.021	0.001
5000	10000	0.001	0.151	0.002
8000	16000	0.003	0.334	0.004
11000	22000	0.004	0.626	0.005

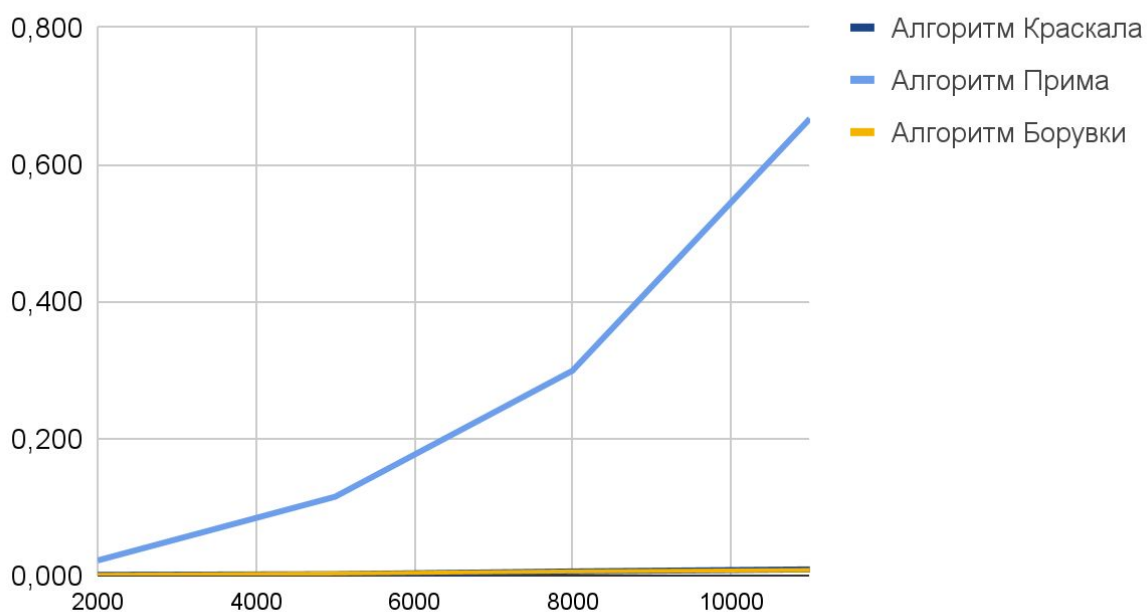
График 3



Для разреженных графов (график 4), $m = 5n$:

Число вершин	Число ребер	Алгоритм Краскала, сек	Алгоритм Прима, сек	Алгоритм Борувки, сек
2000	10000	0.001	0.022	0.001
5000	25000	0.002	0.115	0.003
8000	40000	0.006	0.299	0.006
11000	55000	0.009	0.667	0.008

График 4



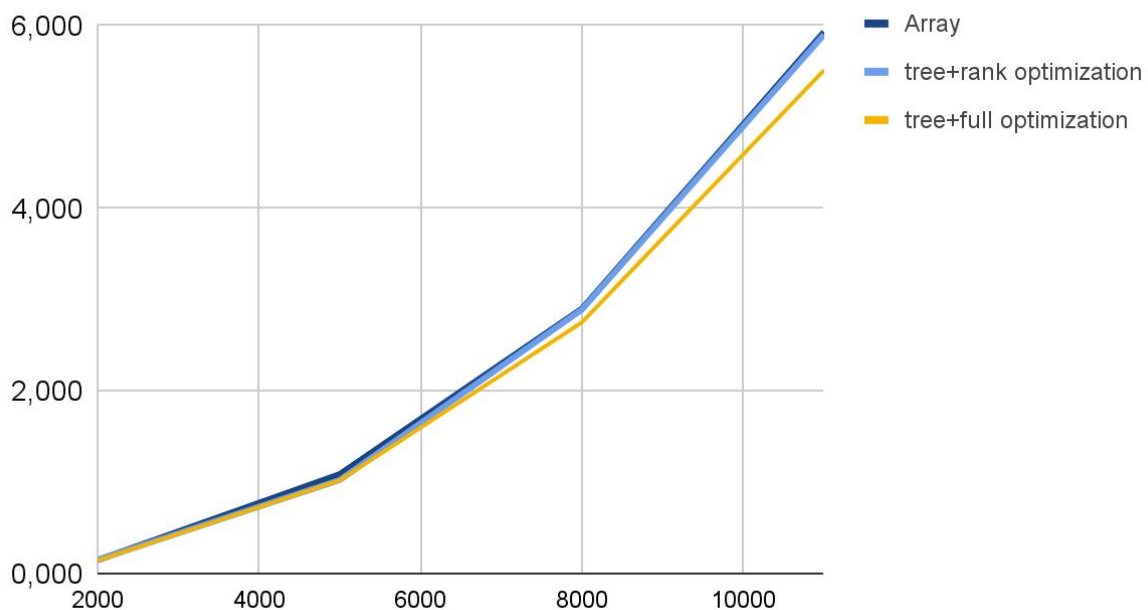
На разреженных графах лучше всего себя показывают алгоритм Краскала и алгоритм Борувки. Хотя константа у алгоритма Борувки хуже.

Время работы данной реализации алгоритма Прима почти не зависит от количества ребер в графе.

Алгоритм Краскала для плотных графов, с разными реализациями непересекающихся множеств(график 5):

Число вершин	Число ребер	Реализация разделенного множества		
		array, сек	tree + rank optimization, сек	tree + full optimization, сек
2000	1000000	0.147	0.150	0.143
5000	6250000	1.088	1.024	1.020
8000	16000000	2.9	2.89	2.75
11000	30250000	5.93	5.9	5.51

График 5



Сложность метода Краскала с непересекающимися множествами на основе массива $O(M \log N + N^2)$, поэтому при работе на плотных графах существенной разницы во времени нет. Но на разреженных графах оптимизация непересекающихся множеств очень важна.

Список литературы

1. Элементы теории графов: учеб. пособие / Л. Н. Домнин. – Пенза: Изд-во Пенз. гос. ун-та, 2007. – 144с.
2. Харари. Ф. Теория графов / Ф. Харари. – М. : Мир, 1973. – 300с.
3. Белов. В. В. Теория графов: Учеб. пособие для вузов / В. В. Белов, Е. М. Воробьев, В. Е. Шаталов. – М. : Высш. шк., 1976. – 392 с.
4. <http://e-maxx.ru>
5. https://ru.wikipedia.org/wiki/Алгоритм_Борувки

Приложение

Код программы

Весь код выложен на [GitHub](https://github.com/AleksandrPanov/minimal_graphs_skeleton/),
https://github.com/AleksandrPanov/minimal_graphs_skeleton/ здесь
представлена лишь малая часть

alg_kraskal.h

```
#pragma once
#include <vector>
enum TypeDisjointSet
{
    array, tree_rank, tree_full_optimization
};
typedef std::vector<std::pair<int, std::pair<int, int>>> EdgesWeight;
EdgesWeight algKraskal(EdgesWeight &edges, int n, TypeDisjointSet choose);
```

alg_kraskal.cpp

```
#include alg_kraskal.h"
#include "../disjoint_sets/disjoint_sets.h"
#include "../disjoint_sets/disjoint_sets_array.h"
#include "../disjoint_sets/disjoint_sets_tree.h"
#include <algorithm>

#define v1 second.first
#define v2 second.second
#define w first

using std::vector;
using std::pair;

typedef std::pair<int, int> Edge;
typedef std::pair<int, std::pair<int, int>> EdgeWeight;

EdgesWeight algKraskal(EdgesWeight &edges, int n, TypeDisjointSet choice)
{
    AbstractDisjointSet *set;
    if (choice == TypeDisjointSet::array)
    {
        set = new DisjointSetArray(n);
    }
    else if (choice == TypeDisjointSet::tree_full_optimization)
    {
        set = new DisjointSet(n);
    }
    else if (choice == TypeDisjointSet::tree_rank)
    {
        set = new DisjointSetRank(n);
    }
}
```

```

EdgesWeight result;
result.reserve(n - 1);
std::sort(edges.begin(), edges.end());
for (int i = 0; i < edges.size(); i++)
{
    auto edge = edges[i];
    int nameSet1 = set->find_set(edge.v1);
    int nameSet2 = set->find_set(edge.v2);
    if (nameSet1 != nameSet2)
    {
        set->union_sets(edge.v1, edge.v2);
        result.emplace_back(EdgeWeight(edge.w, Edge(edge.v1,
edge.v2)));
    }
}
return result;
}

```