Санкт-Петербургский политехнический университет Петра Великого
Институт прикладной математики и информатики
Высшая школа прикладной математики и вычислительной физики

Лабораторная работа №3 по дисциплине

Дискретная математика

Тема: «Деревья»

Вариант 4 – Степенной ряд

Выполнил студент гр. 5030102/20202 Соколов А.Н.

Руководитель Новиков Ф. А.

«___» __________ 202__ г.

Санкт-Петербург

2024

1. Формулировка задачи и ее формализация

Формулировка задачи

1. Необходимо разработать статический класс, реализующий функции разложения графа в степенной ряд и наоборот, для дальнейшего использования пользователем.
2. Указать сложность алгоритма и доказать, что она именно такая.
3. Объяснить почему был выбран тот или иной способ представления графов в программе.
4. Определить, всегда ли граф или степенной ряд получается равен изначальному и объяснить почему.

2. Использованные технологии

Язык программирования

- C++ 23

Система сборки

- CMake 3.27
- Ninja 1.12.1

Исходные файлы программы:

https://github.com/azya0/dm2025/tree/master/lab3

3. Описание алгоритма

Все алгоритмы далее реализуются при помощи “**приоритетной очереди**”. Очередь приоритетов — это абстрактная структура данных, которая хранит элементы с определенными приоритетами (ключами). Чаще всего это используется для управления задачами, где некоторым задачам необходимо предоставить больший приоритет по сравнению с другими. Например, в системах планирования процессов, системах управления очередями, или в алгоритмах поиска.

Элементы в очереди приоритетов обычно представлены в виде пары «приоритет — элемент».

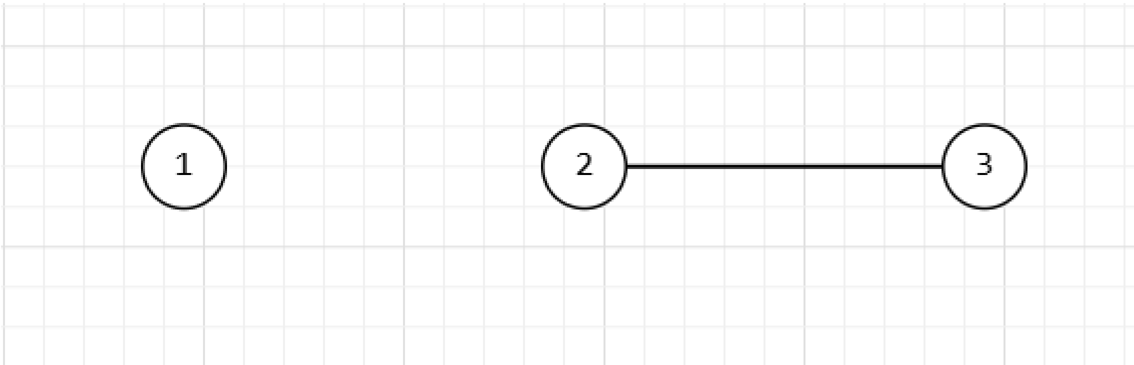
В прошлой лабораторной работе был обоснован выбор представления графа, как ***объектов класса “вершина”, ссылающихся друг на друга****. Аналогичное представление будет выбрано и в рамках данной лабораторной работы.*

Алгоритм разложения в ряд

На вход подается граф. Проходя по каждой его вершине, мы добавляем число ребер в конец массива. Сортируем массив. На выходе получаем неубывающий степенной ряд.

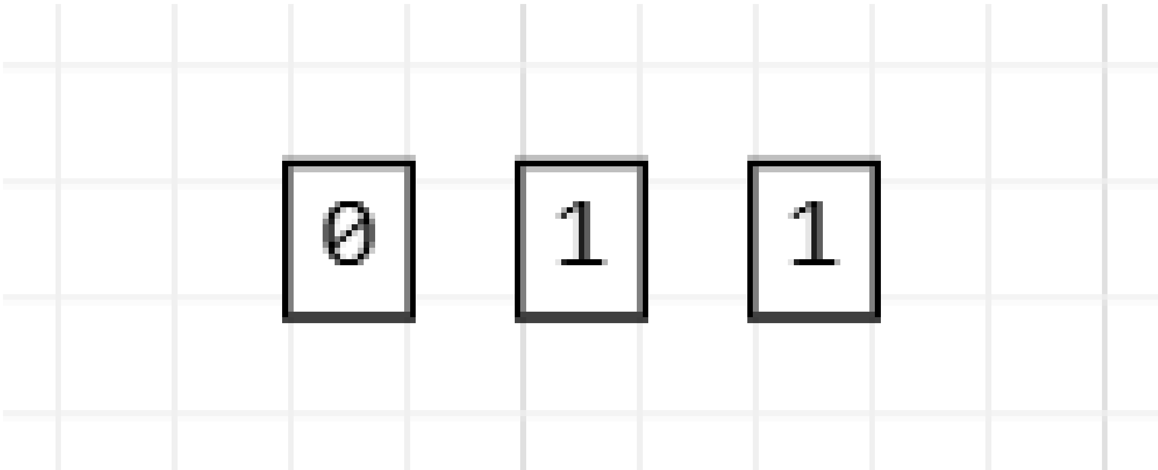
Построение графа по степенному ряду

На вход подается неубывающий степенной ряд. Проходя по каждому числу ребер в нем, записываем это число и индекс вершины в приоритетную очередь, где ключом будет количество ребер. В цикле, пока существуют пары в очереди, достаем максимальную по числу ребер (пусть будет E) и E раз достаем вершины по убыванию количества ребер, после чего устанавливаем между ними связь. На выходе получаем готовый граф или уведомление о невозможности построения такого.
Пример:
Пусть дан граф



1. **Берем вершину 1. У неё 0 рёбер. Записываем 0 в массив.**
2. **Берем вершину 2. У неё 1 ребро. Записываем 1 в массив.**
3. **Берем вершину 3. У неё 1 ребро. Записываем 1 в массив.**
4. **Больше вершин нет.**
5. **Сортируем массив.**
6. **Возвращаем полученный массив пользователю.**

Пусть дан степенной ряд



1. **Разобьем исходный массив на пары: (0, 0), (1, 1), (1, 2).**
2. **Добавим все элементы в приоритетную очередь.**
3. **Проверим, пуста ли очередь. Нет, не пуста.**
4. **Достанем элемент с максимальным количеством ребер из приоритетной очереди. (1, 1). Так получилось, потому что он был добавлен раньше, чем (1, 2). (1, 2) добавился в конец и при сравнении его с предком он не поднялся выше.**
5. **Проверяем, хранится ли в приоритетной очереди еще 1 элемент (количество ребер текущего). Да, хранится.**

6. Достаем из приоритетной очереди еще 1 элемент (количество ребер текущего). Этот элемент (1, 2).
7. Проверяем, есть ли у него возможность установить ещё 1 связь. Да, есть, т.к. количество его оставшихся ребер не 0.
8. В массиве смежностей устанавливаем связь между этими вершинами.
9. Снижаем количество ребер для (1, 2) -> (0, 2)
10. Проверяем, остались ли у него свободные ребра. Нет, не осталось, их теперь 0. Но если бы были, мы бы добавили его обратно в приоритетную очередь.
11. Проверим, пуста ли очередь. Нет, не пуста.
12. Достанем элемент с максимальным количеством ребер из приоритетной очереди. (0, 0).
13. Проверяем, есть ли в очереди еще 0 элементов. Да, есть.
14. Т.к. требуется 0 ребер, то мы не зайдем в цикл, а просто пропустим итераци.
15. Проверим, пуста ли очередь. Да, пуста.
16. Вернем пользователю граф, инициализированный из массива смежностей.

Сложность алгоритма

Рассмотрим алгоритм разложения в ряд

Для прохода по всем n вершинам, нам потребуется n итераций, т.е. $O(n)$

Операция записи количества рёбер вершины в конец массива константина по времени, т.е. $O(1)$

Сортировка массива происходит за $O(nlogn)$ по времени.

Таким образом сложность алгоритма: $O(n \ + \ n\, log\, n)$

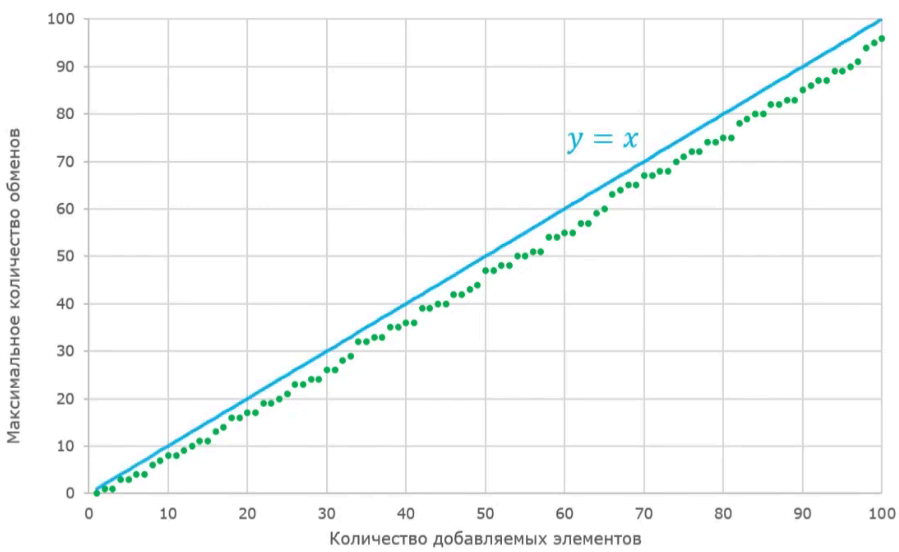
Рассмотрим алгоритм построения графа по степенному ряду

Оценивать будем его верхнюю временную границу.

Для создания пар из количества ребер и индекса нам потребуется n операций, т.е. $O(n)$

Для заполнения приоритетной очереди из не отсортированного массива можно использовать “алгоритм просеивания вниз”, который имеет сложность $O(n)$. Если бы мы просто сделали push для каждой пары, в худшем случае из-за просеивания каждого элемента вверх, это занимало бы $O(n\, log\, n)$, однако, для просеивания вниз время операций сокращается для $O(n)$. Для доказательства, рассмотрим суммы операций:

Двоичная куча: построение



Двоичная куча: построение

- Если мы будем добавлять в кучу 63 элемента и каждый просеивать вниз до листа, то общее число обменов будет равно $32 \cdot 0 + 16 \cdot 1 + 8 \cdot 2 + 4 \cdot 3 + 2 \cdot 4 + 1 \cdot 5 = 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 2 + 2^2 \cdot 3 + 2^1 \cdot 4 + 2^0 \cdot 5$.
- Чем больше расстояние до листа, тем меньше элементов должны его преодолеть.
- Для N элементов общее число обменов равно $2^{\lfloor \log_2 N \rfloor} \cdot 0 + 2^{\lfloor \log_2 N \rfloor - 1} \cdot 1 + 2^{\lfloor \log_2 N \rfloor - 2} \cdot 2 + \dots + 2^0 \cdot \lfloor \log_2 N \rfloor$.
  Можно показать, что с увеличением N эта сумма растёт пропорционально N.
- Следовательно, построение кучи из неупорядоченного массива при помощи просеивания вниз имеет сложность $O(N)$.

Пройдемся по всем элементам кучи. В самом худшем случае число таких проходов будет $n - 1$, т.к. предпоследняя не изолированная вершина установит связь с последний, а значит последнюю рассматривать не придется, т.е. $O(n)$

В каждой такой итерации нам надо получать вершину с максимальным количеством ребер (из оставшихся). Это будет занимать в худшем случае $O(log\, n)$

Для каждой выбранной вершины нужно пройтись по всем её ребрам, которых в худшем случае будет n.

Для установки связей в массиве будет и добавлений в контейнер будут использованы операции, выполняющиеся за константное время, т.е. за $O(1)$

Для возвращения в худшем случае n элементов из контейнера в приоритетную очередь потребуется $O(n\, log\, n)$

Стоит понимать, что выполнение всех худших случаев одновременно невозможно. Данная оценка времени алгоритма показывает лишь верхнюю границу, до которой точная сложность алгоритма не достанет. Однако данная граница находится довольно близко к точной.

Таким образом верхняя граница сложности алгоритма:

$$O(2n + n\,(log\, n + n\, log\, n + n)) =$$
$$= O(2n + n\, log\, n + n^2 log\, n + n^2) =$$
$$= O((n^2 + n)\, log\, n + n^2 + 2n)$$

Почему такая сложность?

Если бы мы использовали массив изначальный отсортированный массив, то мы бы столкнулись со следующей проблемой:

Откладывание от упорядоченной степенной последовательности не гарантирует упорядоченную степенную последовательность. Иными словами:

$d = (0\ 2\ 2\ 2\ 2\ 2\ 3\ 3)$

$d' = (0\ 2\ 2\ 2\ 1\ 1\ 2)$

В таком случае нам либо придется каждую итерацию сортировать массив, либо придется отказаться от возможности проверки через отложение вершины наибольшей степени, что дополнительно ещё сильнее усложняет процесс.

4. Практическая реализация

В программе граф представлен так же, как и в лабораторной работе №2.

Такое представление было выбрано, т.к. с ним удобно работать и оно подразумевает использование всей выделенной под него памяти. Например, если бы я представлял граф как матрицу, то часть значений были бы однотипной записью по типу “-1”, которое расходует память в пустую и требует $O(n^2)$ сложность вывода в консоль.

Необработанные вершины

Программа предполагает, что вместе с исполняемым файлом пользователь будет хранить файл произвольного расширения, описывающий граф в формате:

A 2 B 1 C 1
B 3 A 1 C 1 D 1
C 3 A 1 B 1 D 1
D 2 C 1 B 1
E 0

Приоритетная очередь реализована в “pqueue/pqueue.h” без “*.cpp файла” из-за желания использовать “дженерики”.

```
#pragma once

#include <memory>
#include <functional>
#include <vector>
#include <stdexcept>

template <typename T>
class PQueue {
private:
    using function = std::function<bool(T, T)>;

    std::shared_ptr<function> function_ptr;
    std::vector<T> list;

    void swap(int firstIndex, int secondIndex) noexcept {…

    using descendants = std::pair<std::shared_ptr<int>, std::shared_ptr<int>>;

    std::shared_ptr<int> getCorrectIndex(int index) noexcept {…

    descendants getDescendants(int index) noexcept {…

    int best(int firstIndex, int secondIndex) noexcept {…

    int chooseBest(int baseIndex, int anotherIndex) noexcept {…

    int getBestDescendant(int index) noexcept {…

    void up(int index) {…

    void down(int index) {…
public:
    PQueue(std::shared_ptr<function> _function_ptr) noexcept {…

    void fromVector(std::vector<T>& data) {…

    void push(T value) noexcept {…

    T pop() {…

    int size() noexcept {…

    bool empty() noexcept {…
};
```

Для реализации алгоритмов используется статический класс **“Builder”**

```
#include <algorithm>

#include "../pqueue/pqueue.h"
#include "../graph/graph.h"

class Builder {
public:
    static std::shared_ptr<std::vector<int>>
buildPowerSeries(std::shared_ptr<Graph> graph);

    using Pair = std::pair<int, int>;
    static std::shared_ptr<Graph>
buildGraph(std::shared_ptr<std::vector<int>> powerSeries);
};
```

статический метод `buildPowerSeries` строит неубывающий степенной ряд по графу, а статический метод `buildGraph` строит граф по не убывающему степенному ряду.

Вызов этих функций происходит в main файле.

5. Область применения

Файл с графом

- Неверный формат файла:
    - В файле содержится неправильный формат

Работа программы

- Невозможность построить граф по степенному ряду
    - Функция `buildGraph` вернет nullptr, а пользовательский интерфейс в файле main сообщит о невозможности построения пользователю

Во всех остальных случаях программа будет работать корректно

6. Формат входных и выходных данных

В качестве вводных данных пользователю нужно записать граф в верном формате в файле graph.txt

На выход пользователь получит:
1. неубывающий степенной ряд из введенного графа

2. новый граф, построенный из убывающего степенного ряда, построенного из введенного ряда

7. Вопрос совпадения с изначальными данными

Если мы соберем граф по степенному ряду, а потом разложим его в другой степенной ряд, то получим точно такой же, как исходный, потому что в рамках данной лабораторной работы я сортирую степенные ряды в порядке неубывания

Если мы разложим граф в степенной ряд, а потом соберем его в граф, то у нас может получиться как исходный, так и другой граф. Разница будет заключаться только в названии вершин, но не в структуре. Почему так? Потому что из степенного ряда величины n можно собрать до $n!$ графов, от размещения названий всех вершин.

8. Вывод

В рамках данной лабораторной работы был реализован алгоритмы разложения графа в степенной ряд и построения графа по степенному ряду.