МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

УТВЕРЖДАЮ
Зав.кафедрой,
доцент, к. ф.-м. н.
_____ М. В. Огнева

ТЕОРИЯ ГРАФОВ

ОТЧЕТ О ПРАКТИКЕ

студента 3 курса 351 группы факультета КНиИТ
Мангасаряна Евгения Павловича

вид практики: учебная (рассредоточенная)
кафедра: информатики и программирования
курс: 3
семестр: 1

Проверено:
доцент _____ А. П. Грецова

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью практической работы является закрепление и углубление теоретических знаний по дисциплине «Теория графов» посредством реализации класса «Граф» на выбранном языке программирования. Для достижения данной цели были поставлены следующие задачи:

- создание класса «Граф»;
- работа со списками смежности;
- реализация обходов графа;
- построение минимального остовного дерева;
- работа со взвешенным графом;
- реализация потокового алгоритма;
- выполнение творческого задания.

Все задачи выполнены с использованием языка программирования TypeScript.

1 Минимальные требования для класса «Граф»

1.1 Условие задания

Для решения всех задач курса необходимо создать класс (или иерархию классов — на усмотрение разработчика), содержащий:

1. Структуру для хранения списка смежности графа (не работать с графом через матрицы смежности, если в некоторых алгоритмах удобнее использовать список ребер — реализовать метод, создающий список ребер на основе списка смежности).
2. Конструкторы (не менее 3-х):
   - добавляющие вершину;
   - добавляющие ребро (дугу);
   - удаляющие вершину;
   - удаляющие ребро (дугу);
   - выводящие список смежности в файл (в том числе в пригодном для чтения конструктором формате).
3. Методы:
   - конструктор по умолчанию, создающий пустой граф;
   - конструктор, заполняющий данные графа из файла;
   - конструктор-копию (аккуратно, не все сразу делают именно копию);
   - специфические конструкторы для удобства тестирования.
4. Должны поддерживаться как ориентированные, так и неориентированные графы.
5. Добавьте минималистичный консольный интерфейс пользователя, позволяющий добавлять и удалять вершины и ребра (дуги) и просматривать текущий список смежности графа.

1.2 Примеры исходного кода

Следующий код описывает класс `Graph`, соответствующий требованиям условия:

```
export class Graph {
  private adj: Map<string, Map<string, number>> = new Map()
  private weighted: boolean = false
  private oriented: boolean = false

  constructor(weighted: boolean, oriented: boolean)
```

```
  constructor(textRepr: string)
  constructor(other: Graph)
  constructor(arg1: boolean | string | Graph, arg2?: boolean) {
    if (typeof arg1 === 'boolean' && typeof arg2 === 'boolean') {
      this.weighted = arg1
      this.oriented = arg2
      this.adj = new Map()
    } else if (typeof arg1 === 'string' && arg2 == null) {
      this.loadFromFile(arg1)
      if (!this.oriented) {
        for (const [v, neighbors] of this.adj) {
          for (const [u, w] of neighbors) {
            this.adj.get(u)!.set(v, w)
          }
        }
      }
    } else if (arg1 instanceof Graph && arg2 == null) {
      this.weighted = arg1.weighted
      this.oriented = arg1.oriented
      this.adj = new Map(arg1.adj)
    } else {
      throw new Error('Invalid arguments')
    }
  }
  // другие методы ...
}
```

Методы для добавления вершины и ребра (дуги):

```
addNode(label: string) {
  if (this.adj.has(label)) {
    throw new NodeAlreadyExists(label)
  }
  this.adj.set(label, new Map())
}

connect(a: string, b: string, weight?: number) {
  if (!this.adj.has(a)) {
    throw new NodeNotExists(a)
  }
  if (!this.adj.has(b)) {
    throw new NodeNotExists(b)
```

```
  }
  if (this.adj.get(a)!.has(b)) {
    throw new ConnectionAlreadyExists(a, b)
  }

  if (this.weighted) {
    this.adj.get(a)!.set(b, weight!)
    if (!this.oriented) {
      this.adj.get(b)!.set(a, weight!)
    }
  } else {
    if (weight) {
      throw new WeightsInNonWeightedGraph()
    }
    this.adj.get(a)!.set(b, 0)
    if (!this.oriented) {
      this.adj.get(b)!.set(a, 0)
    }
  }
}
```

Удаление вершины и дуги (ребра):

```
removeNode(label: string) {
  if (!this.adj.has(label)) {
    throw new NodeNotExists(label)
  }

  this.adj.delete(label)
  for (let value of this.adj.values()) {
    value.delete(label)
  }
}

disconnect(a: string, b: string) {
  if (!this.adj.has(a)) {
    throw new NodeNotExists(a)
  }
  if (!this.adj.has(b)) {
    throw new NodeNotExists(b)
  }
  if (!this.adj.get(a)!.has(b)) {
```

```
20      throw new ConnectionNotExists(a, b)
21    }
22
23    this.adj.get(a)!.delete(b)
24    if (!this.oriented) {
25      this.adj.get(b)!.delete(a)
26    }
27  }
```

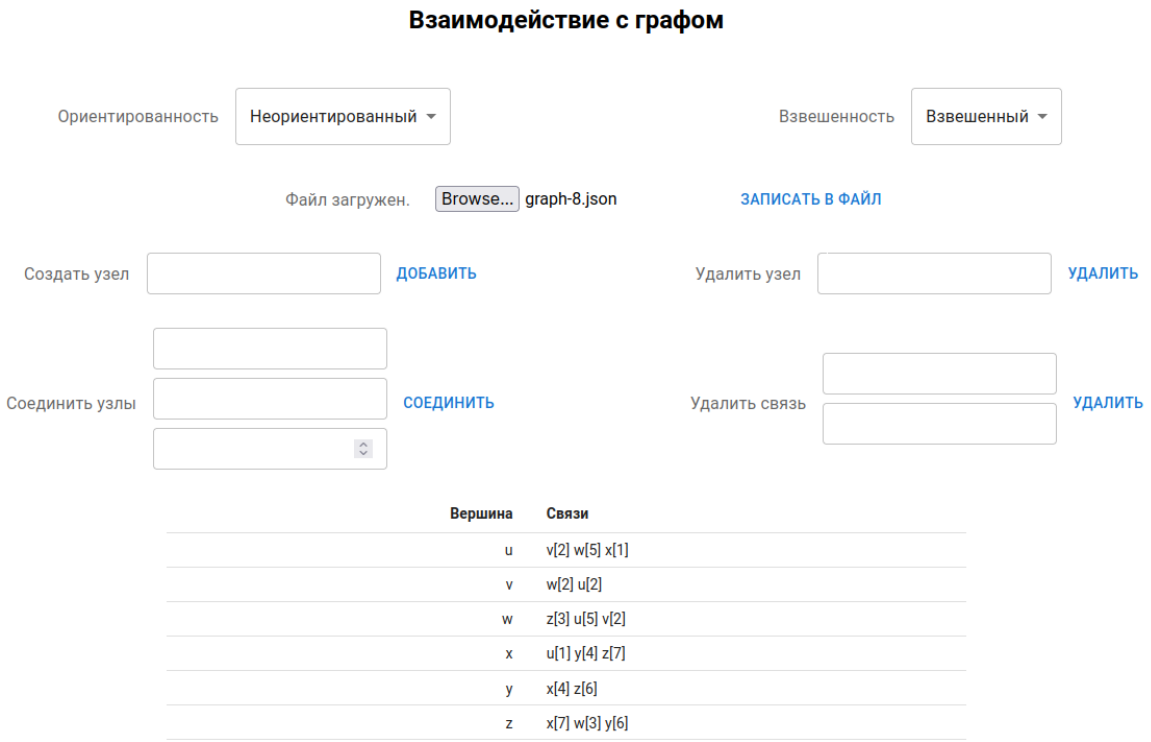


Рисунок 1.1 – Общий вид интерфейса

Вместо консольного был реализован веб-интерфейс (см. рис. 1.1) на React.js. Основной компонент интерфейса (`App`) представлен в приложении А.

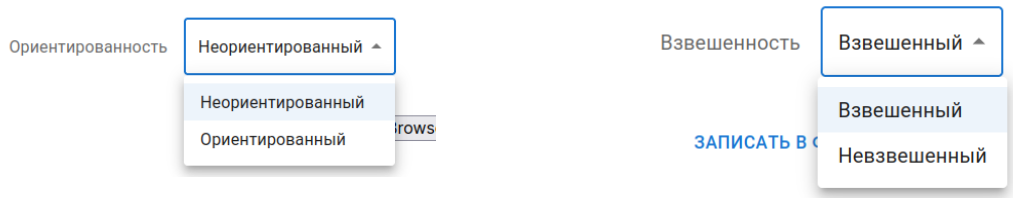


Рисунок 1.2 – Виды графов

Графы можно загружать из файлов и сохранять в файлы. В последнем случае на компьютер скачается файл с описанием графа в формате JSON.

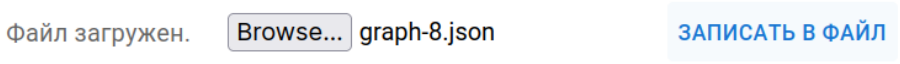


Рисунок 1.3 – Загрузка и выгрузка в файл

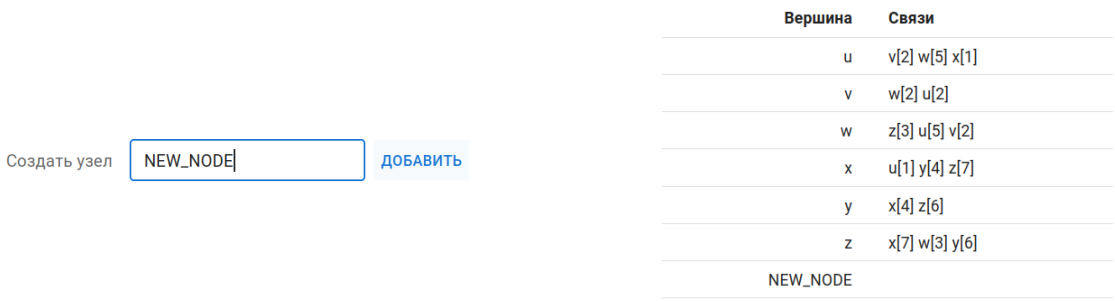


Рисунок 1.4 – Добавление вершины в граф

Интерфейс позволяет пользователю работать с различными видами графов: ориентированным/неориентированным, взвешенным/невзвешенным (см. рис. 1.2).

Текущее состояние графа отображается в виде списка смежности внизу интерфейса.

Доступные операции:

— добавление вершин;
— удаление вершин;
— создание связей;
— удаление связей.

Пример добавления вершины и результат действия представлен на рис. 1.4.

Пример создания связи между вершинами и результат представлены на рис. 1.5.

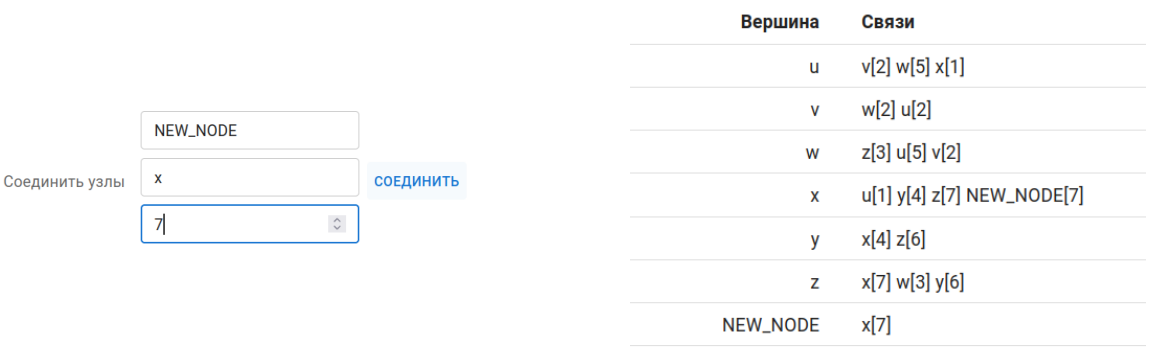


Рисунок 1.5 – Создание связи между вершинами

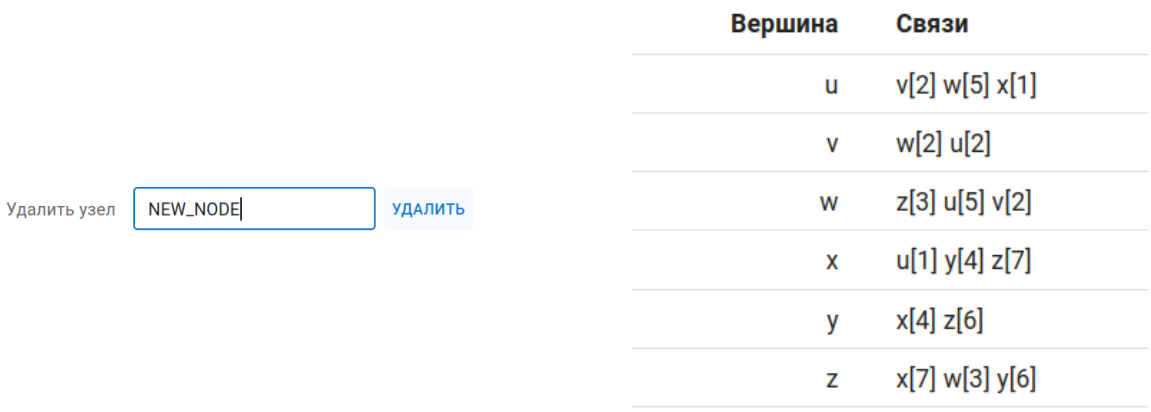


Рисунок 1.6 – Удаление вершины

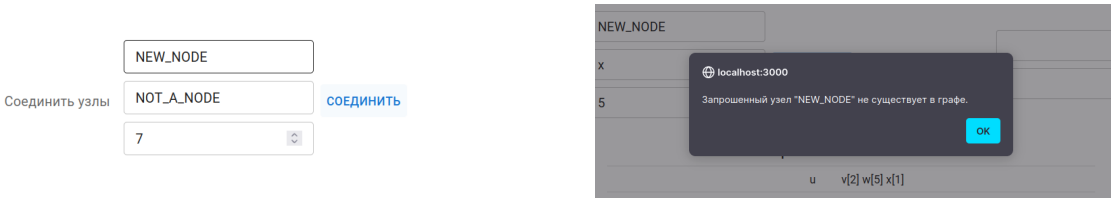


Рисунок 1.7 – Вывод сообщения об ошибке

Пример удаления вершины и результат представлены на рис. 1.6.

Если в какой-то операции пользователь пытается выполнить недопустимое действие (создать вершину с пустой меткой, сослаться на несуществующую вершину), ему выводится сообщение об ошибке (см. рис. 1.7).

1.3 Примеры входных и выходных данных

Взвешенный ориентированный граф:

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "a": {
      "b": 2,
      "c": 3
    },
    "b": {
      "d": 5
    },
    "c": {
      "e": 4,
      "f": 2
```

```
    },
    "d": {
      "g": 6
    },
    "e": {
      "h": 3
    },
    "f": {
      "h": 7
    },
    "g": {
      "h": 8
    },
    "h": {}
  }
}
```

Взвешенный неориентированный граф:

```
{
  "weighted": true,
  "oriented": false,
  "adj": {
    "a": {
      "b": 2,
      "c": 3,
      "d": 5
    },
    "b": {
      "c": 4,
      "e": 6
    },
    "c": {
      "d": 1,
      "f": 3
    },
    "d": {
      "g": 2
    },
    "e": {
      "f": 4,
      "h": 5
```

```
24      },
25      "f": {
26        "h": 2,
27        "g": 3
28      },
29      "g": {
30        "h": 1
31      },
32      "h": {}
33    }
34  }
```

Выходные файлы имеют такой же формат и полностью совместимы с входными.

2 Список смежности Ia

2.1 Условие задания

Вариант 5: Для каждой вершины орграфа вывести её степень.

2.2 Примеры исходного кода

Для нахождения и вывода степени каждой вершины был создан вспомогательный компонент `VertexPowers`, который находит степени вершин и выводит их в виде таблицы:

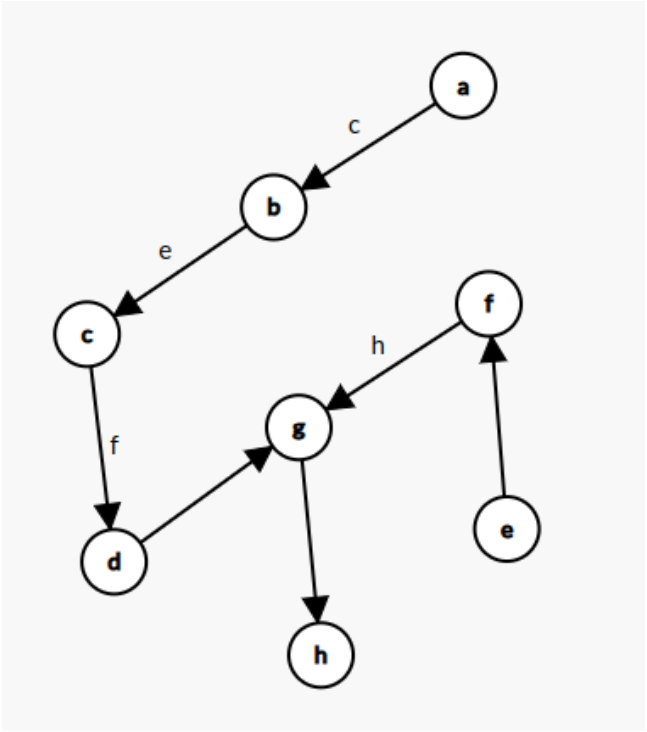
```
function VertexPowers() {
  const adjList = graph.current!.getAdjacencyList()
  const powers = new Map()

  for (const [k, v] of adjList) {
    powers.set(k, v.length)
    for (const [otherK, otherV] of adjList) {
      if (k === otherK) continue

      if (otherV.find(value => value[0] === k)) {
        powers.set(k, powers.get(k) + 1)
      }
    }
  }

  return (
    <TableBody>
      {graph.current!.getAdjacencyList().map(item => {
        return (
          <TableRow key={item[0]}>
            <TableCell align='right'>{item[0]}</TableCell>
            <TableCell align='left'>{powers.get(item[0])}</TableCell>
          </TableRow>
        )
      })
      }
    </TableBody>
  )
}
```

2.3 Краткое описание алгоритма

Для каждой вершины подсчитывается степень исхода и степень захода, а затем суммируются.

2.4 Примеры входных и выходных данных

2.4.1 Входные данные



Вершина	Связи
a	b c
b	c e
c	d f
d	g
e	f
f	g h
g	h
h	

Рисунок 2.1 – Ориентированный граф

```
{
  "weighted": false,
  "oriented": true,
  "adj": {
    "a": {
      "b": 0,
      "c": 0
    },
    "b": {
      "c": 0,
      "e": 0
    },
    "c": {
      "d": 0,
      "f": 0
```

```
    },
    "d": {
      "g": 0
    },
    "e": {
      "f": 0
    },
    "f": {
      "g": 0,
      "h": 0
    },
    "g": {
      "h": 0
    },
    "h": {}
  }
}
```

2.4.2 Выходные данные

Вершина	Связи	Вершина	Степень
a	b c	a	2
b	c e	b	3
c	d f	c	4
d	g	d	2
e	f	e	2
f	g h	f	4
g	h	g	3
h		h	2

Рисунок 2.2 – Результат работы

3 Список смежности Ia

3.1 Условие задания

Вариант 20: Вывести все вершины орграфа, не смежные с данной.

3.2 Примеры исходного кода

Для нахождения и вывода вершин орграфа, не смежных с данной, был описан метод `getAdjacent()`:

```
getAdjacent(label: string): Map<string, number> {
  if (!this.adj.has(label)) {
    throw new NodeNotExists(label)
  }
  return this.adj.get(label)!
}
```

Затем, при выводе результата, запрашиваются все метки вершин орграфа и из них отбрасываются те, которые смежны с данной:

```
let resList: string[] = []
for (const [node, _] of graph.current!.getAdjacencyList()) {
  if (nodeName !== node && !adj.has(node)) {
    resList.push(node.toString())
  }
}
setAnswer(resList.join(', '))
```

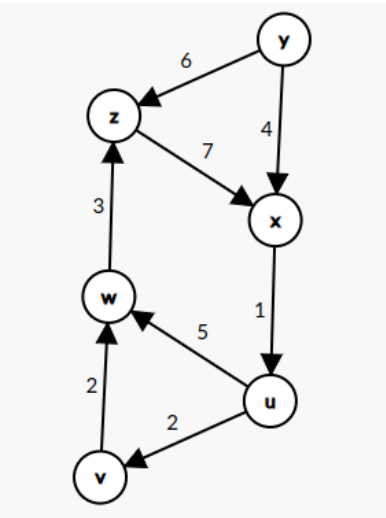
3.3 Краткое описание алгоритма

Поскольку данные о смежности хранятся в виде списка смежности, достаточно для заданной вершины запросить информацию о смежных с ней вершинах. А далее простая фильтрация списка строк.

Поскольку графы могут быть взвешенными, и как пересекать веса не оговаривается, то выбирается минимум из двух весов при пересечении.

3.4 Примеры входных и выходных данных

3.4.1 Входные данные



Вершина	Связи
u	v[2] w[5]
v	w[2]
w	z[3]
x	u[1]
y	x[4] z[6]
z	x[7]

Рисунок 3.1 – Ориентированный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "u": {
      "v": 2,
      "w": 5
    },
    "v": {
      "w": 2
    },
    "w": {
      "z": 3
    },
    "x": {
      "u": 1
    },
    "y": {
      "x": 4,
      "z": 6
    },
    "z": {
      "x": 7
    }
```

```
25      }
26  }
```

3.4.2 Выходные данные

Вершина	Связи
u	v[2] w[5]
v	w[2]
w	z[3]
x	u[1]
y	x[4] z[6]
z	x[7]

Ответ: u, v, w, y

Вершина z ВЫВЕСТИ

Рисунок 3.2 – Результат работы

4 Список смежности Iб: несколько графов

4.1 Условие задания

Вариант 8: Построить орграф, являющийся пересечением двух заданных.

4.2 Примеры исходного кода

Для нахождения пересечения двух орграфов был описан метод `intersect()`:

```
intersect(other: Graph): Graph {
  if (!this.oriented || !other.oriented) {
    throw new InvalidOperandTypes()
  }

  const res = new Graph(this.weighted || other.weighted, true)
  const intersection = new Map<string, Map<string, number>>()

  const commonNodes = Array.from(this.adj.keys()).filter(node =>
  ↪  other.adj.has(node))

  for (const node of commonNodes) {
    const neighborsA = this.adj.get(node) || new Map<string, number>()
    const neighborsB = other.adj.get(node) || new Map<string, number>()
    const commonNeighbors = new Map<string, number>()

    for (const [neighbor, weightA] of neighborsA) {
      if (neighborsB.has(neighbor)) {
        const weightB = neighborsB.get(neighbor) || 0
        commonNeighbors.set(neighbor, Math.min(weightA, weightB))
      }
    }

    intersection.set(node, commonNeighbors)
  }

  res.adj = intersection
  return res
}
```

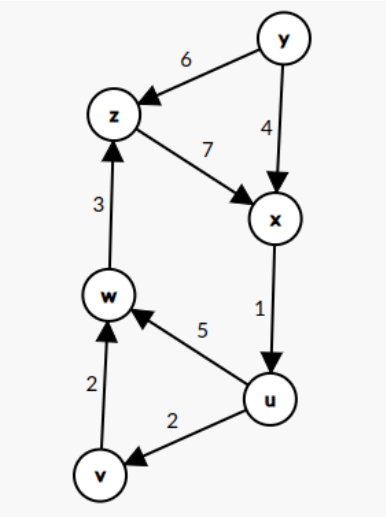
В интерфейсе орграф-пересечение выводится как обычно, в виде списка смежности.

4.3 Краткое описание алгоритма

Создается новый граф, в котором будет накапливаться результат. Затем по определению — добавляем в результат только нужные вершины и дуги.

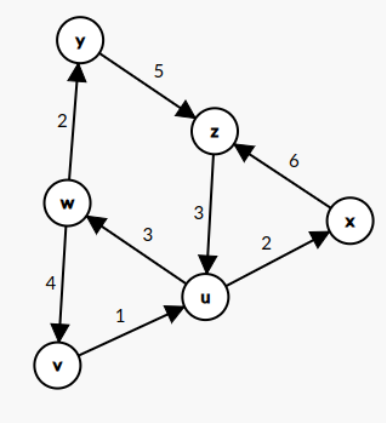
4.4 Примеры входных и выходных данных

4.4.1 Входные данные



Вершина	Связи
u	v[2] w[5]
v	w[2]
w	z[3]
x	u[1]
y	x[4] z[6]
z	x[7]

Рисунок 4.1 – Первый ориентированный граф



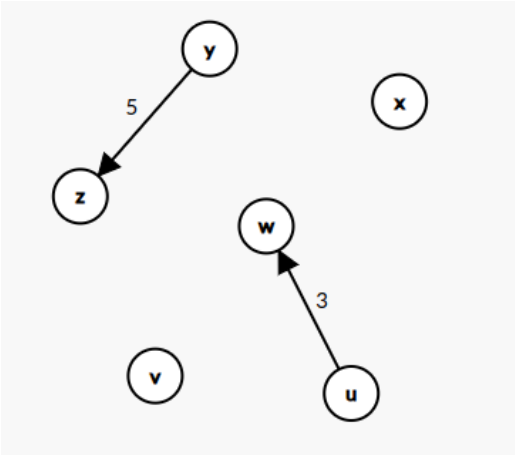
Вершина	Связи
u	w[3] x[2]
v	u[1]
w	v[4] y[2]
x	z[6]
y	z[5]
z	u[3]

Рисунок 4.2 – Второй ориентированный граф

```
// первый орграф
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "u": {
      "w": 3,
      "x": 2
    },
    "v": {
      "u": 1
    },
    "w": {
      "v": 4,
      "y": 2
    },
    "x": {
      "z": 6
    },
    "y": {
      "z": 5
    },
    "z": {
      "u": 3
    }
  }
}
```

```
// второй орграф
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "u": {
      "v": 2,
      "w": 5
    },
    "v": {
      "w": 2
    },
    "w": {
      "z": 3
    },
    "x": {
      "u": 1
    },
    "y": {
      "x": 4,
      "z": 6
    },
    "z": {
      "x": 7
    }
  }
}
```

4.4.2 Выходные данные



Результат

Вершина	Связи
u	w[3]
v	
w	
x	
y	z[5]
z	

Рисунок 4.3 – Результат работы

5 Обходы графа II

5.1 Условие задания

Вариант 5: Подсчитать количество связных компонент графа.

5.2 Примеры исходного кода

Для нахождения компонент связности графа был описан метод `connectedComponents()`:

```
connectedComponents(): number {
  const visited: Set<string> = new Set()

  let count = 0 // количество компонент
  for (const node of this.adj.keys()) {
    if (!visited.has(node)) { // если еще не посетили, выполнить DFS
      count++ // новая компонента
      this.dfs(node, visited) // DFS посещаем все узлы в этой компоненте
       ↪ связности
    }
  }

  return count
}
```

А также вспомогательный метод обхода в глубину:

```
private dfs(node: string, visited: Set<string>) {
  visited.add(node)

  const neighbors = this.adj.get(node)!
  for (const neighbor of neighbors.keys()) {
    if (!visited.has(neighbor)) {
      this.dfs(neighbor, visited) // рекурсивно обойти соседние узлы
    }
  }
}
```

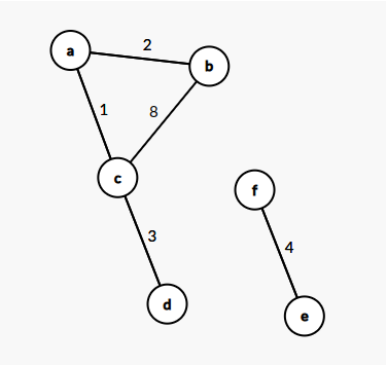
5.3 Краткое описание алгоритма

Вводится счетчик количества компонент связности. Также ведется множество уже посещенных вершин. Алгоритм заканчивает работу, когда все вершины посещены. На каждой итерации выбирается непосещенная вершина и с помощью обхода в глубину посещаются все связные с ней вершины (одна компонента

связности). Таким образом, мы сможем подсчитать, сколько всего компонент связности в графе.

5.4 Примеры входных и выходных данных

5.4.1 Входные данные



Вершина	Связи
a	b[2] c[1]
b	a[2]
c	b[8] d[3]
d	c[3]
e	f[4]
f	e[4]

Рисунок 5.1 – Неориентированный граф

```
{
  "weighted": true,
  "oriented": false,
  "adj": {
    "a": {
      "b": 2,
      "c": 1
    },
    "b": {
      "a": 2
    },
    "c": {
      "a": 1,
      "d": 3
    },
    "d": {
      "c": 3
    },
    "e": {
      "f": 4
    },
    "f": {
      "e": 4
    }
```

```
25      }
26  }
```

5.4.2 Выходные данные

Вершина	Связи
a	b[2] c[1]
b	a[2]
c	a[1] d[3]
d	c[3]
e	f[4]
f	e[4]

Ответ: 2

Рисунок 5.2 – Результат работы

6 Обходы графа II

6.1 Условие задания

Вариант 27: Найти длины кратчайших (по числу дуг) путей из вершины u во все остальные.

6.2 Примеры исходного кода

Для нахождения длины кратчайших путей из вершины u во все остальные был объявлен метод `shortestPathLengthsFrom()`, принимающий метку вершины u и возвращающий расстояния до всех вершин графа (по числу дуг):

```
shortestPathLengthsFrom(u: string): Map<string, number> {
  if (!this.adj.has(u)) {
    throw new NodeNotExists(u)
  }

  const shortestPaths: Map<string, number> = new Map()

  for (const node of this.adj.keys()) {
    shortestPaths.set(node, Infinity) // пока считаем, что расстояния до
    ↪ других узлов бесконечность
  }
  shortestPaths.set(u, 0) // расстояние до самого себя 0

  const queue = [u] // очередь обхода
  while (queue.length > 0) {
    const currentNode = queue.shift()!
    const neighbors = this.adj.get(currentNode)!

    for (const neighbor of neighbors.keys()) {
      if (shortestPaths.get(neighbor) === Infinity) { // если узел еще не был
      ↪ посещен
        // установить кратчайшее расстояние до него
        shortestPaths.set(neighbor, shortestPaths.get(currentNode)! + 1)
        queue.push(neighbor) // добавить соседний узел в очередь обхода
      }
    }
  }

  return shortestPaths
}
```

6.3 Краткое описание алгоритма

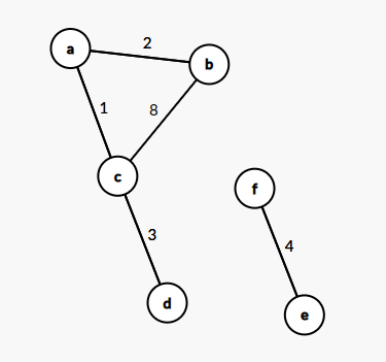
Этот алгоритм использует метод обхода в ширину (BFS) для нахождения кратчайших путей.

Создается `Map<string, number>`, где для каждой вершины устанавливается начальное расстояние. Расстояние от u до самой себя равно 0, а до остальных вершин — бесконечность.

Используется очередь для обхода графа. Начальная вершина u добавляется в очередь. Пока очередь не пуста, извлекается текущая вершина. Для каждого соседнего узла проверяется, был ли он уже посещен. Если не был, устанавливается кратчайшее расстояние и добавляется в очередь.

6.4 Примеры входных и выходных данных

6.4.1 Входные данные



Вершина	Связи
a	b[2] c[1]
b	a[2]
c	b[8] d[3]
d	c[3]
e	f[4]
f	e[4]

Рисунок 6.1 – Неориентированный граф

```
{
  "weighted": true,
  "oriented": false,
  "adj": {
    "a": {
      "b": 2,
      "c": 1
    },
    "b": {
      "a": 2
    },
    "c": {
      "a": 1,
      "d": 3
```

```
15        },
16        "d": {
17          "c": 3
18        },
19        "e": {
20          "f": 4
21        },
22        "f": {
23          "e": 4
24        }
25      }
26    }
```

6.4.2 Выходные данные

Вершина	Длина кратчайшего пути
a	1
b	2
c	0
d	1
e	Infinity
f	Infinity

Вершина c ВЫВЕСТИ

Рисунок 6.2 – Результат работы

7 Каркас III

7.1 Условие задания

Дан взвешенный неориентированный граф из N вершин и M ребер. Требуется найти в нем каркас минимального веса. (Алгоритм Прима)

7.2 Примеры исходного кода

Для нахождения каркаса минимального веса взвешенного неориентированного графа был создан метод `mst()` (*Minimal Spanning Tree*):

```
mst(): Graph {
  if (!this.weighted || this.oriented) {
    throw new GraphNotWeightedUnoriented()
  }

  const mst = new Graph(true, false) // минимальное остовное дерево
  const visited = new Set<string>() // множество посещенных вершин

  // взять любую вершину как начальную (здесь первая)
  const startVertex = this.adj.keys().next().value
  if (!startVertex) {
    throw new GraphIsEmpty()
  }
  visited.add(startVertex)
  mst.addNode(startVertex)

  while (visited.size < this.adj.size) { // пока не посетим все вершины
    // ребра, которые соединяют посещенные вершины с непосещенными
    const edges: Array<Edge> = []

    for (const vertex of visited) { // найти такие ребра
      for (const [neighbor, weight] of this.adj.get(vertex)!) {
        if (!visited.has(neighbor)) {
          edges.push({from: vertex, to: neighbor, weight});
        }
      }
    }

    if (edges.length === 0) {
      // Не все вершины еще посещены, но мы не смогли найти новые ребра.
      // Это значит, что граф несвязный.
      throw new GraphIsNotConnected()
```

```
33      }
34
35      // выбрать ребро с наименьшим весом
36      edges.sort((a, b) => a.weight - b.weight)
37      const {from, to, weight} = edges[0]
38
39      // добавить ребро в минимальное остовное дерево
40      mst.addNode(to)
41      mst.connect(from, to, weight)
42      visited.add(to)
43    }
44
45    return mst
46  }
```

7.3 Краткое описание алгоритма

Основная идея заключается в том, чтобы начать с одной вершины и пошагово добавлять ребра минимального веса, соединяющие посещенные вершины с непосещенными.

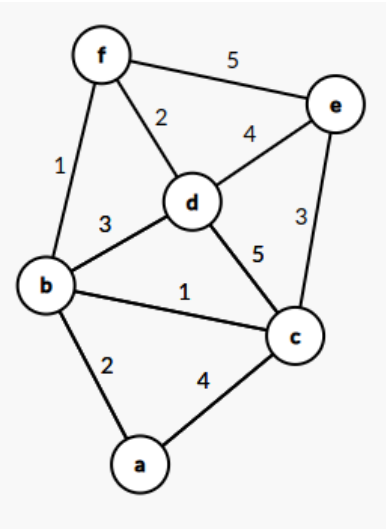
Создается новый граф `mst`, представляющий минимальное остовное дерево. Также создается множество `visited` для отслеживания посещенных вершин. Выбирается любая вершина в качестве начальной. В данной реализации используется первая вершина из списка вершин графа.

Пока не посещены все вершины графа, выполняется цикл:

1. Создается массив `edges`, содержащий ребра, соединяющие посещенные вершины с непосещенными.
2. Если массив `edges` пуст, это означает, что граф несвязный (ошибка).
3. Ребра в массиве сортируются по весу, и выбирается ребро с минимальным весом.
4. Выбранное ребро добавляется в остовное дерево, связывая вершины `from` и `to` с весом `weight`. Вершина `to` помечается как посещенная.

7.4 Примеры входных и выходных данных

7.4.1 Входные данные



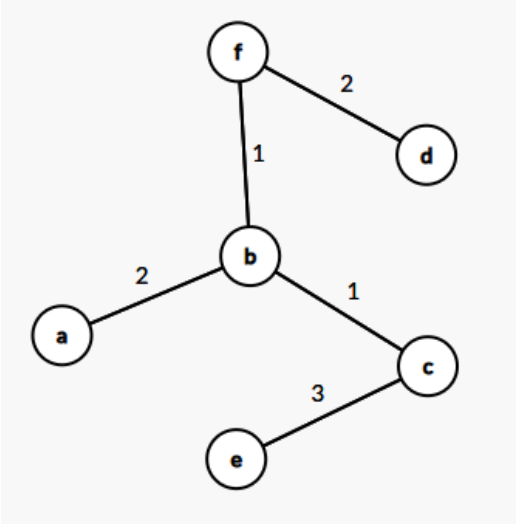
Вершина	Связи
a	b[2] c[4]
b	a[2] c[1] d[3] f[1]
c	a[4] b[1] d[5] e[3]
d	b[3] c[5] e[3] f[2]
e	c[3] d[3] f[5]
f	e[5] b[1] d[2]

Рисунок 7.1 – Неориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": false,
  "adj": {
    "a": {
      "b": 2,
      "c": 4
    },
    "b": {
      "a": 2,
      "c": 1,
      "d": 3,
      "f": 1
    },
    "c": {
      "a": 4,
      "b": 1,
      "d": 5,
      "e": 3
    },
    "d": {
      "b": 3,
      "c": 5,
      "e": 3,
```

```
25          "f": 2
26        },
27        "e": {
28          "c": 3,
29          "d": 4,
30          "f": 5
31        },
32        "f": {
33          "e": 5,
34          "b": 1,
35          "d": 2
36        }
37      }
38  }
```

7.4.2 Выходные данные



Вершина	Связи
a	b[2]
b	a[2] c[1] f[1]
c	b[1] e[3]
f	b[1] d[2]
d	f[2]
e	c[3]

Рисунок 7.2 – Результат работы

8 Веса IV а

8.1 Условие задания

В графе нет ребер отрицательного веса.

Вариант 2: Определить, есть ли в графе вершина, минимальные стоимости путей от которой до остальных в сумме не превосходят P.

8.2 Примеры исходного кода

Для этой задачи был создан метод `taskEight()`::

```
/**
 * Метод, определяющий, есть ли в графе вершина такая, что сумма
 * длин кратчайших путей от нее до всех остальных вершин
 * не превышает `P`. Построен на основе алгоритма Дейкстры.
 * В графе не может быть отрицательных весов.
 * @param P
 */
taskEight(P: number): boolean {
  /**
   * Вспомогательная функция для определения вершины, до которой
   * путь кратчайший. Возвращает `null`, если пути вообще нет.
   * @param dist расстояния до других вершин
   * @param visited множество посещенных
   */
  const minDistance = (dist: Map<string, number>, visited: Set<string>):
  ↪ string | null => {
    let min = Infinity  // минимальное расстояние
    let minVertex: string | null = null  // метка вершины, до которой
    ↪ расстояние минимальное

    for (const vertex of this.adj.keys()) {
      if (!visited.has(vertex) && dist.get(vertex)! <= min) {
        min = dist.get(vertex)!
        minVertex = vertex
      }
    }

    return minVertex
  }

  /**
   * Алгоритм Дейкстры нахождения кратчайших путей.
```

```
31      * @param source начальная вершина
32      */
33    const dijkstra = (source: string): Map<string, number> => {
34      const dist: Map<string, number> = new Map()  // расстояния до других
          ↪ вершин
35      const visited: Set<string> = new Set()  // для контроля уже посещенных
          ↪ вершин
36
37      // инициализируем расстояния бесконечностью
38      for (const vertex of this.adj.keys()) {
39        dist.set(vertex, Infinity)
40      }
41
42      dist.set(source, 0)  // расстояние до самой себя 0
43
44      for (let i = 0; i < this.adj.size - 1; i++) {
45        const u = minDistance(dist, visited)
46        if (u === null) {
47          break  // если нет путей в непосещенные вершины
48        }
49
50        visited.add(u)
51
52        for (const [v, weight] of this.adj.get(u)!.entries()) {
53          if (weight < 0) {
54            throw new GraphHasNegativeWeights()
55          }
56
57          const newDist = dist.get(u)! + weight
58          if (newDist < dist.get(v)!) {
59            dist.set(v, newDist)
60          }
61        }
62      }
63
64      return dist
65    }
66
67    // вычислить сумму длин кратчайших путей для каждой вершины
68    for (const vertex of this.adj.keys()) {
69      const dist = dijkstra(vertex)
```

```
70      const sum = Array.from(dist.values())
71        .reduce((acc, val) => acc + val, 0)
72
73      if (sum > P) {
74        return false
75      }
76    }
77
78    return true
79  }
```

8.3 Краткое описание алгоритма

Алгоритм основан на модификации алгоритма Дейкстры для нахождения кратчайших путей во взвешенных неориентированных графах.

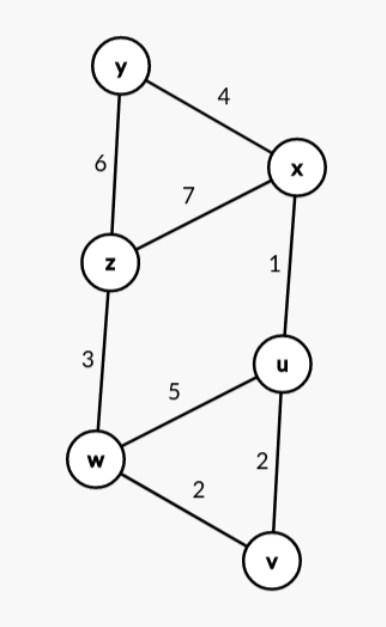
Вспомогательная функция `minDistance()` находит вершину, до которой кратчайшее расстояние и возвращает ее метку. В случае отсутствия путей в непосещенные вершины возвращает `null`.

Для каждой вершины графа выполняется алгоритм Дейкстры, который находит кратчайшие пути от этой вершины до всех остальных. Веса ребер проверяются на отрицательность, так как алгоритм Дейкстры не будет корректно работать в этом случае.

После нахождения кратчайших путей для каждой вершины вычисляется сумма длин этих путей. Если сумма превышает заданное значение `P`, алгоритм возвращает `false`, иначе `true`.

8.4 Примеры входных и выходных данных

8.4.1 Входные данные



Вершина	Связи
u	v[2] w[5] x[1]
v	w[2] u[2]
w	z[3] u[5] v[2]
x	u[1] y[4] z[7]
y	x[4] z[6]
z	x[7] w[3] y[6]

Рисунок 8.1 – Взвешенный граф

```
{
  "weighted": true,
  "oriented": false,
  "adj": {
    "u": {
      "v": 2,
      "w": 5
    },
    "v": {
      "w": 2
    },
    "w": {
      "z": 3
    },
    "x": {
      "u": 1
    },
    "y": {
      "x": 4,
      "z": 6
    },
    "z": {
```

```
23          "x": 7
24        }
25      }
26  }
```

8.4.2 Выходные данные

Рисунок 8.2 – Результат работы

9 Веса IV b

9.1 Условие задания

В графе нет циклов отрицательного веса.

Вариант 14: Вывести кратчайшие пути из вершины u во все остальные вершины.

9.2 Примеры исходного кода

Для выполнения задания был реализован метод `taskNine()`:

```
/**
 * Метод, возвращающий для данной вершины кратчайшие пути до других
 * вершин. При этом в графе могут присутствовать отрицательные веса,
 * но не может быть отрицательных циклов. В графе могут быть отрицательные
 * веса, но не может быть отрицательных циклов. Реализация построена
 * на основе алгоритма Беллмана-Форда.
 *
 * @param sourceVertex вершина, от которой искать кратчайшие пути
 */
taskNine(sourceVertex: string): Map<string, { distance: number, path: string[]
 ↪  }> {
  if (!this.exists(sourceVertex)) {
    throw new NodeNotExists(sourceVertex)
  }

  const paths: Map<string, { distance: number, path: string[] }> = new Map()

  // инициализировать расстояния до всех вершин бесконечностью, кроме
   ↪  начальной вершины
  for (const vertex of this.adj.keys()) {
    paths.set(vertex, {
      distance: Infinity,
      path: []
    })
  }
  paths.set(sourceVertex, {distance: 0, path: []})

  // релаксация ребер
  for (let i = 0; i < this.adj.size - 1; i++) {
    for (const [u, neighbors] of this.adj.entries()) {
```

```
29        for (const [v, weight] of neighbors.entries()) {
30          const uDist = paths.get(u)!.distance
31          const vDist = paths.get(v)!.distance
32
33          if (uDist + weight < vDist) {  // если d(u) + w < d(v)
34            paths.set(v, {
35              distance: uDist + weight,
36              path: [...paths.get(u)!.path, u]
37            })  // то d(v) <- d(u) + w
38          }
39        }
40      }
41    }
42
43    // проверка на отрицательные циклы
44    for (const [u, neighbors] of this.adj.entries()) {
45      for (const [v, weight] of neighbors.entries()) {
46        const uDist = paths.get(u)!.distance
47        const vDist = paths.get(v)!.distance
48
49        if (uDist + weight < vDist) {
50          throw new GraphHasNegativeLoops()
51        }
52      }
53    }
54
55    return paths
56  }
```

9.3 Краткое описание алгоритма

Этот метод реализует алгоритм Беллмана—Форда для нахождения кратчайших путей от заданной вершины `sourceVertex` до всех остальных вершин в графе. Алгоритм позволяет обрабатывать графы с отрицательными весами на ребрах, но при этом предполагается, что в графе отсутствуют отрицательные циклы.

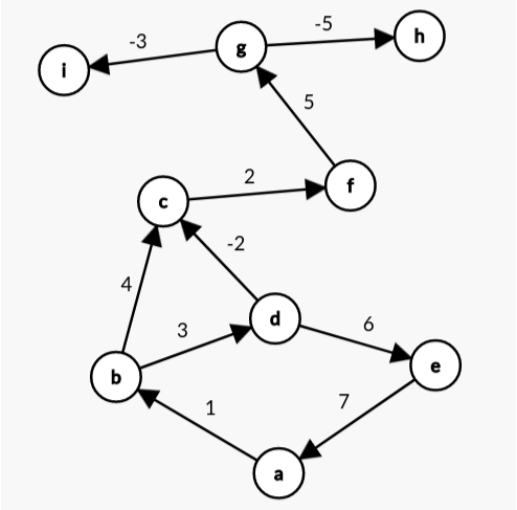
Создается отображение `paths`, где для каждой вершины хранится информация о кратчайшем пути: расстояние и список вершин, составляющих путь. Расстояния до всех вершин инициализируются бесконечностью, за исключением начальной вершины, для которой расстояние устанавливается в `0`.

Происходит итеративная релаксация ребер графа. Алгоритм повторяется $(n-1)$ раз. Для каждого ребра проверяется, можно ли уменьшить расстояние до его конечной вершины, используя текущий путь. Если такое уменьшение возможно, то обновляется информация о кратчайшем пути.

После завершения релаксации ребер происходит проверка наличия отрицательных циклов в графе. Это делается путем еще одного прохода по всем ребрам. Если находится ребро, для которого можно уменьшить расстояние до его конечной вершины, значит такой цикл есть.

9.4 Примеры входных и выходных данных

9.4.1 Входные данные



Вершина	Связи
a	b[1]
b	c[4] d[3]
c	f[2]
d	c[-2] e[6]
e	a[7]
f	g[5]
g	h[-5] i[-3]
h	
i	

Рисунок 9.1 – Неориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "a": {
      "b": 1
    },
    "b": {
      "c": 4,
      "d": 3
    },
    "c": {
```

```
        "f": 2
      },
      "d": {
        "c": -2,
        "e": 6
      },
      "e": {
        "a": 7
      },
      "f": {
        "g": 5
      },
      "g": {
        "h": -5,
        "i": -3
      },
      "h": {},
      "i": {}
    }
}
```

9.4.2 Выходные данные

Вершина	Длина	Кратчайший путь
a	Infinity	
b	Infinity	
c	0	
d	Infinity	
e	Infinity	
f	2	c
g	7	c,f
h	2	c,f,g
i	4	c,f,g

u c ВЫПОЛНИТЬ

Рисунок 9.2 – Результат работы

10 Веса IV с

10.1 Условие задания

В графе могут быть циклы отрицательного веса.

Вариант 10: Вывести кратчайший путь из вершины u до вершины v.

10.2 Примеры исходного кода

Для выполнения задания был реализован метод `taskTen()`:

```
/**
 * Метод, который находит кратчайший путь между двумя данными
 * вершинами `u` и `v`. В графе могут быть отрицательные циклы.
 * Реализация построена на алгоритме Флойда.
 * @param u начальная вершина
 * @param v конечная вершина
 */
taskTen(u: string, v: string): { distance: number; path: string[] } {
  if (!this.exists(u)) {
    throw new NodeNotExists(u)
  }
  if (!this.exists(v)) {
    throw new NodeNotExists(v)
  }

  // Map, связывающая строковую метку вершины с числом
  const indexOf = new Map(Array.from(this.adj.keys()).map((v, i) => [v, i]))
  const labelOf = new Map(Array.from(this.adj.keys()).map((v, i) => [i, v]))
  // Список смежности, только метки теперь числа
  const adj = new Map(Array.from(this.adj.entries()).map((v, i) =>
    [i, new Map(Array.from(v[1]).map(v => [indexOf.get(v[0])!, v[1]]))]
  ))
  const vertices = Array.from(adj.keys())

  // инициализировать матрицу расстояний и матрицу следующих вершин
  const dist: number[][] = []
  const next: (number | null)[][] = []

  for (const i of vertices) {
    dist[i] = []
    next[i] = []
    for (const j of vertices) {
```

```
33        dist[i][j] = i === j ? 0 : Infinity
34        next[i][j] = null
35      }
36    }
37
38    // заполнить матрицу расстояний на основе весов списка смежности
39    for (const [src, neighbors] of adj.entries()) {
40      for (const [dst, weight] of neighbors.entries()) {
41        dist[src][dst] = weight
42        next[src][dst] = dst // установить следующую вершину в соответствии с
          ↪ ребром
43      }
44    }
45
46    // алгоритм Флойда
47    for (const k of vertices) {
48      for (const i of vertices) {
49        for (const j of vertices) {
50          if (dist[i][k] + dist[k][j] < dist[i][j]) {
51            dist[i][j] = dist[i][k] + dist[k][j]
52            next[i][j] = next[i][k]
53          }
54        }
55      }
56    }
57
58    // построить кратчайший путь
59    if (dist[indexOf.get(u)!][indexOf.get(v)!] >= 0 &&
60        dist[indexOf.get(u)!][indexOf.get(v)!] !== Infinity)
61    {
62      // если в матрице неотрицательное число, значит, вершины
63      // не находятся на отрицательном цикле
64      const path: string[] = []
65      let current = indexOf.get(u) ?? null
66      let target = indexOf.get(v)!
67
68      while (current !== target) {
69        if (current === null) {
70          return {distance: Infinity, path: []}
71        }
72        path.push(labelOf.get(current)!)
```

```
73        current = next[current][target]
74      }
75      path.push(labelOf.get(target)!)
76
77      return {distance: dist[indexOf.get(u)!][indexOf.get(v)!], path}
78    }
79    else {
80      // иначе понятие"кратчайшее расстояние" не существует
81      return {distance: -Infinity, path: []}
82    }
83  }
```

10.3 Краткое описание алгоритма

Этот метод решает задачу нахождения кратчайшего пути между двумя вершинами u и v в графе, учитывая возможное наличие отрицательных циклов. Реализован на основе алгоритма Флойда.

Создаются отображения `indexOf` и `labelOf`, которые связывают строковые метки вершин с числовыми индексами и наоборот. Также создается матрица смежности `adj` с числовыми индексами вершин. Инициализируются матрицы расстояний `dist` и следующих вершин `next`.

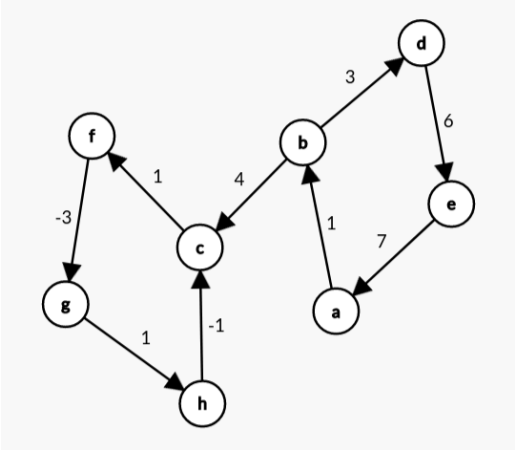
Заполняется матрица расстояний на основе весов ребер графа. Применяется алгоритм Флойда для нахождения кратчайших путей между всеми парами вершин в графе.

Строится кратчайший путь от вершины u до вершины v на основе матрицы следующих вершин. Результат включает в себя длину кратчайшего пути и список вершин, составляющих этот путь.

При отсутствии пути между вершинами u и v в графе возвращается объект с бесконечной длиной пути и пустым списком вершин.

10.4 Примеры входных и выходных данных

10.4.1 Входные данные



Вершина	Связи
a	b[1]
b	c[4] d[3]
c	f[1]
d	e[6]
e	a[7]
f	g[-3]
g	h[1]
h	c[-1]

Рисунок 10.1 – Ориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "a": {
      "b": 1
    },
    "b": {
      "c": 4,
      "d": 3
    },
    "c": {
      "f": 1
    },
    "d": {
      "e": 6
    },
    "e": {
      "a": 7
    },
    "f": {
      "g": -3
    },
    "g": {
      "h": 1
```

```
    },
    "h": {
      "c": -1
    }
  }
}
```

10.4.2 Выходные данные

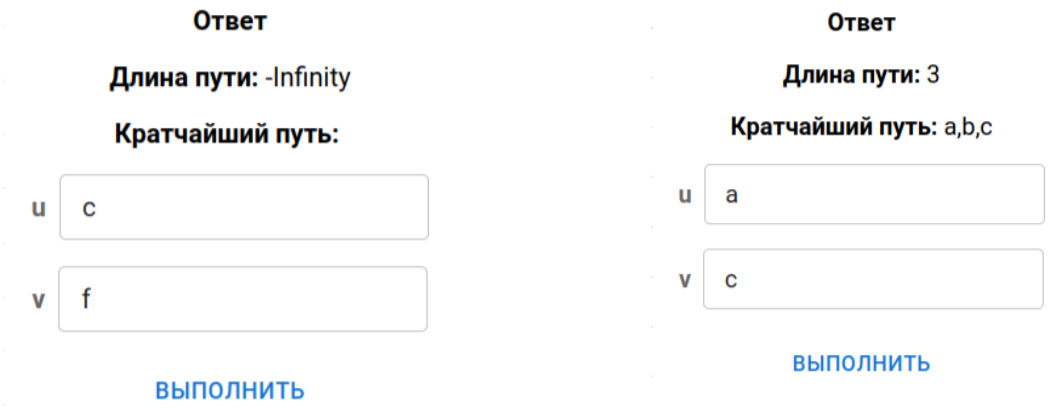


Рисунок 10.2 – Результат работы

11 Максимальный поток

11.1 Условие задания

Решить задачу на нахождение максимального потока любым алгоритмом. Подготовить примеры, демонстрирующие работу алгоритма в разных случаях.

11.2 Примеры исходного кода

Для выполнения задания был реализован метод `taskEleven()`, а также вспомогательный метод `findAugmentingPath()`. С целью экономии места, код `findAugmentingPath()` расположен в приложении Б.

```
/**
 * Метод, находящий максимальный поток а графе, используя алгоритм
 * Форда-Фалкерсона.
 * @param source источник
 * @param sink сток
 */
taskEleven(source: string, sink: string): number {
  if (!this.exists(source)) {
    throw new NodeNotExists(source)
  }
  if (!this.exists(sink)) {
    throw new NodeNotExists(sink)
  }

  // создать остаточный граф с теми же вершинами, что и в изначальном
  const resGraph: Map<string, Map<string, number>> = new Map()
  for (const [vertex, edges] of this.adj.entries()) {
    resGraph.set(vertex, new Map(edges))
  }

  let maxFlow = 0

  // расширять поток, пока есть расширяющий путь
  let path = this.findAugmentingPath(resGraph, source, sink)
  while (path.length > 0) {
    // найти минимальную пропускную способность
    const minCapacity = this.findMinCapacity(resGraph, path)

    // обновить остаточный граф вычитанием минимальной пропускной способности
    for (let i = 0; i < path.length - 1; i++) {
```

```
31        const u = path[i]
32        const v = path[i + 1]
33
34        resGraph.get(u)!.set(v, resGraph.get(u)!.get(v)! - minCapacity)
35
36        // добавить обратную дугу с отрицательным весом
37        if (!resGraph.has(v)) {
38          resGraph.set(v, new Map())
39        }
40
41        if (!resGraph.get(v)!.has(u)) {
42          resGraph.get(v)!.set(u, 0)
43        }
44
45        resGraph.get(v)!.set(u, resGraph.get(v)!.get(u)! + minCapacity)
46      }
47
48      // обновить значение максимального потока
49      maxFlow += minCapacity
50
51      // найти максимальный расширяющий путь
52      path = this.findAugmentingPath(resGraph, source, sink)
53    }
54
55    return maxFlow
56  }
```

11.3 Краткое описание алгоритма

Создается остаточный граф, который изначально совпадает с оригинальным графом, но все веса обратных ребер устанавливаются в 0.

Алгоритм производит поиск расширяющих путей в остаточном графе с помощью вспомогательного метода `findAugmentingPath()`. Этот метод использует обход в ширину (BFS) для нахождения пути от источника к стоку, учитывая только те ребра, у которых остаточная пропускная способность больше нуля.

Если найден расширяющий путь, вычисляется минимальная пропускная способность на этом пути. Затем обновляются значения остаточных пропускных способностей вдоль этого пути путем вычитания минимальной пропускной способности.

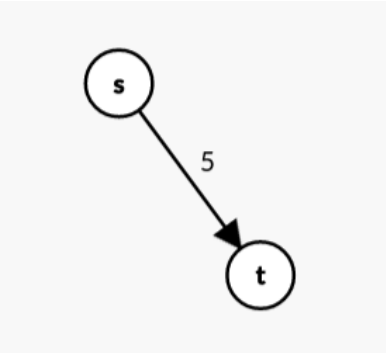
Процесс поиска расширяющих путей и их обновления в остаточном графе

повторяется до тех пор, пока не удастся найти больше расширяющих путей. В процессе выполнения алгоритма суммируется пропускная способность всех найденных путей, что и является максимальным потоком.

11.4 Примеры входных и выходных данных

11.4.1 Входные данные

Простой граф с двумя вершинами и одним направленным ребром.



Вершина	Связи
s	t[5]
t	

Рисунок 11.1 – Ориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "s": {
      "t": 5
    },
    "t": {}
  }
}
```

11.4.2 Выходные данные

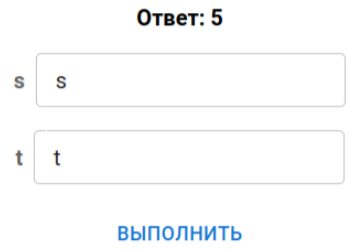
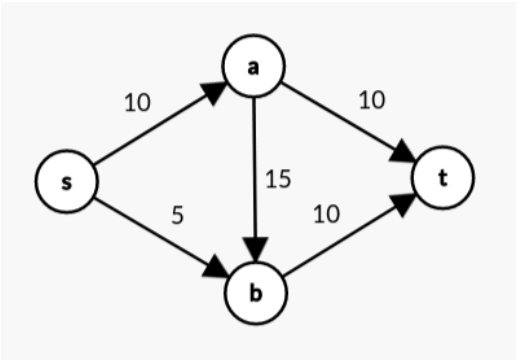


Рисунок 11.2 – Результат работы

11.4.3 Входные данные

Более сложный граф с четырьмя вершинами и несколькими ребрами.



Вершина	Связи
s	a[10] b[5]
a	b[15] t[10]
b	t[10]
t	

Рисунок 11.3 – Ориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "source": {
      "a": 10,
      "b": 5
    },
    "a": {
      "b": 15,
      "sink": 10
    },
    "b": {
      "sink": 10
    },
    "t": {}
  }
}
```

11.4.4 Выходные данные

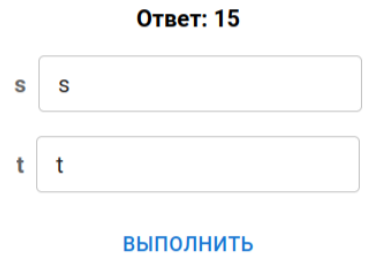
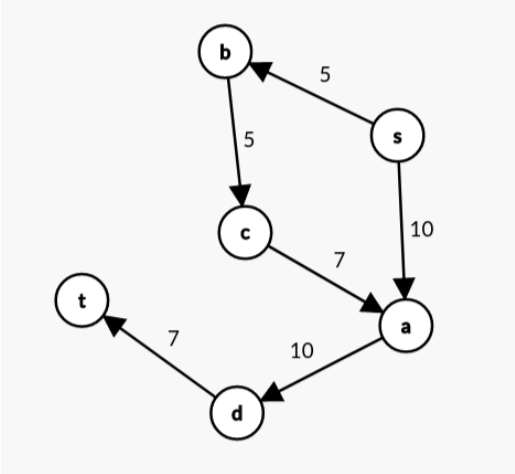


Рисунок 11.4 – Результат работы

11.4.5 Входные данные



Вершина	Связи
s	a[10] b[5]
c	a[7]
b	c[5]
a	d[10]
d	t[7]
t	

Рисунок 11.5 – Ориентированный взвешенный граф

```
{
  "weighted": true,
  "oriented": true,
  "adj": {
    "s": {
      "a": 10,
      "b": 5
    },
    "c": {
      "a": 7
    },
    "b": {
      "c": 5
    },
    "a": {
      "d": 10
```

```
    },
    "d": {
      "t": 7
    },
    "t": {}
  }
}
```

11.4.6 Выходные данные

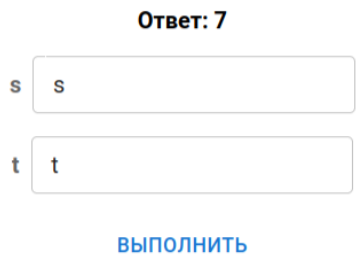


Рисунок 11.6 – Результат работы

ЗАКЛЮЧЕНИЕ

В ходе практики был выполнен ряд задач, которые поспособствовали закреплению и углублению теоретических знаний по дисциплине «Теория графов», посредством реализации класса «Граф» на языке TypeScript.

ПРИЛОЖЕНИЕ А

Компонент App веб-интерфейса

```
function App() {
  const graph = useRef(new Graph(false, false))
  const [newNode, setNewNode] = useState<string>('')
  const [deleteNode, setDeleteNode] = useState<string>('')
  const [connNodeA, setConnNodeA] = useState<string>('')
  const [connNodeB, setConnNodeB] = useState<string>('')
  const [connNodeWeight, setConnNodeWeight] = useState<number | null>(null)
  const [delConnA, setDelConnA] = useState<string>('')
  const [delConnB, setDelConnB] = useState<string>('')
  const [shouldRerender, setShouldRerender] = useState<boolean>(true)

  const orientedSelection = useRef('unoriented')
  const weightedSelection = useRef('non-weighted')

  useEffect(() => {
      setShouldRerender(false)
    }, [shouldRerender]
  )

  function onCreateNodeClick() {
    if (!newNode) {
      alert('Незаполненные поля!')
      return
    }

    try {
      graph.current.addNode(newNode)
    }
    catch (e) {
      if (e instanceof GraphError) {
        alert(e.message)
        return
      }
    }
    setNewNode('')
    setShouldRerender(true)
  }

  function onDeleteNodeClick() {
```

```
40      if (!deleteNode) {
41        alert('Незаполненные поля!')
42        return
43      }
44
45      try {
46        graph.current.removeNode(deleteNode)
47      }
48      catch (e) {
49        if (e instanceof GraphError) {
50          alert(e.message)
51          return
52        }
53      }
54      setDeleteNode('')
55      setShouldRerender(true)
56    }
57
58    function onConnectNodesClick() {
59      if (!connNodeA || !connNodeB || (graph.current.isWeighted() &&
        ↪  !connNodeWeight)) {
60        alert('Незаполненные поля!')
61        return
62      }
63
64      try {
65        graph.current.connect(connNodeA, connNodeB, connNodeWeight ?? undefined)
66      }
67      catch (e) {
68        if (e instanceof GraphError) {
69          alert(e.message)
70          return
71        }
72      }
73      setConnNodeA('')
74      setConnNodeB('')
75      setConnNodeWeight(null)
76      setShouldRerender(true)
77    }
78
79    function onDeleteConnectionClick() {
```

```
80      if (!delConnA || !delConnB) {
81        alert('Незаполненные поля!')
82        return
83      }
84
85      try {
86        graph.current.disconnect(delConnA, delConnB)
87      }
88      catch (e) {
89        if (e instanceof GraphError) {
90          alert(e.message)
91          return
92        }
93      }
94      setDelConnA('')
95      setDelConnB('')
96      setShouldRerender(true)
97    }
98
99    function onGraphLoaded(fileContent: string) {
100     try {
101       graph.current = new Graph(fileContent)
102     }
103     catch (e) {
104       alert('Неверный формат файла!')
105       return
106     }
107
108     if (graph.current.isOriented()) {
109       orientedSelection.current = 'oriented'
110     }
111     else {
112       orientedSelection.current = 'unoriented'
113     }
114
115     if (graph.current.isWeighted()) {
116       weightedSelection.current = 'weighted'
117     }
118     else {
119       weightedSelection.current = 'non-weighted'
120     }
```

```
121
122       setShouldRerender(true)
123     }
124
125     function onGraphOrientedChange(value: string) {
126       orientedSelection.current = value
127       graph.current.changeOriented(value === 'oriented')
128       setShouldRerender(true)
129     }
130
131     function onGraphWeightedChange(value: string) {
132       weightedSelection.current = value
133       graph.current.changeWeighted(value === 'weighted')
134       setShouldRerender(true)
135     }
136
137     return (
138       <div id='app'>
139         <header>
140           <h2>Взаимодействие с графом</h2>
141         </header>
142         <div className='controls'>
143           <div className='control' style={{ gridColumnStart: '2',
              ↪ gridColumnEnd: '4' }}>
144             <InputLabel>Ориентированность</InputLabel>
145             <Select value={orientedSelection.current}
146                     onChange={e => onGraphOrientedChange(e.target.value)}>
147               <MenuItem value='unoriented'>Неориентированный</MenuItem>
148               <MenuItem value='oriented'>Ориентированный</MenuItem>
149             </Select>
150           </div>
151           <div className='control' style={{ gridColumnStart: '4',
              ↪ gridColumnEnd: '6' }}>
152             <InputLabel>Взвешенность</InputLabel>
153             <Select value={weightedSelection.current}
154                     onChange={e => onGraphWeightedChange(e.target.value)}>
155               <MenuItem value='weighted'>Взвешенный</MenuItem>
156               <MenuItem value='non-weighted'>Невзвешенный</MenuItem>
157             </Select>
158           </div>
159           <div className='control' style={{gridTemplateColumns: '1fr 1fr 1fr',
              ↪ gridColumnStart: '2', gridColumnEnd: '6'}}>
```

```
160             <GraphLoader onGraphLoaded={onGraphLoaded} />
161             <GraphDumper graph={graph} />
162           </div>
163           <div className='control' style={{gridColumnStart: '2', gridColumnEnd:
              ↪ '4'}}>
164             <InputLabel>Создать узел</InputLabel>
165             <TextField value={newNode ?? ''}
166                        type='text'
167                        onChange={e => setNewNode(e.target.value)}
168                        size='small'>
169             </TextField>
170             <Button onClick={onCreateNodeClick}>Добавить</Button>
171           </div>
172           <div className='control' style={{gridColumnStart: '4', gridColumnEnd:
              ↪ '6'}}>
173             <InputLabel>Удалить узел</InputLabel>
174             <TextField value={deleteNode ?? ''}
175                        type='text'
176                        onChange={e => setDeleteNode(e.target.value)}
177                        size='small'>
178             </TextField>
179             <Button onClick={onDeleteNodeClick}>Удалить</Button>
180           </div>
181           <div className='control' style={{gridColumnStart: '2', gridColumnEnd:
              ↪ '4'}}>
182             <InputLabel>Соединить узлы</InputLabel>
183             <div style={{ display: 'grid', gridTemplateRows: '1fr 1fr 1fr'}}>
184               <TextField value={connNodeA ?? ''}
185                          type='text'
186                          onChange={e => setConnNodeA(e.target.value)}
187                          size='small'
188                          style={{paddingBottom: '0.5em'}}>
189               </TextField>
190               <TextField value={connNodeB ?? ''}
191                          type='text'
192                          onChange={e => setConnNodeB(e.target.value)}
193                          size='small'
194                          style={{paddingBottom: '0.5em'}}>
195               </TextField>
196               <TextField value={connNodeWeight ?? ''}
197                          type='number'
```

```
198                         onChange={e =>
                            ↪ setConnNodeWeight(Number(e.target.value))}
199                         size='small'>
200               </TextField>
201             </div>
202             <Button onClick={onConnectNodesClick}>Соединить</Button>
203           </div>
204           <div className='control' style={{gridColumnStart: '4', gridColumnEnd:
              ↪ '6'}}>
205             <InputLabel>Удалить связь</InputLabel>
206             <div style={{ display: 'grid', gridTemplateRows: '1fr 1fr'}}>
207               <TextField value={delConnA ?? ''}
208                         type='text'
209                         onChange={e => setDelConnA(e.target.value)}
210                         size='small'
211                         style={{paddingBottom: '0.5em'}}>
212               </TextField>
213               <TextField value={delConnB ?? ''}
214                         type='text'
215                         onChange={e => setDelConnB(e.target.value)}
216                         size='small'>
217               </TextField>
218             </div>
219             <Button onClick={onDeleteConnectionClick}>Удалить</Button>
220           </div>
221         </div>
222         <main style={{width: '100%', display: 'flex', justifyContent:
            ↪ 'center'}}>
223           <div id='connections'>
224             <GraphView graph={graph} />
225           </div>
226         </main>
227         <hr />
228         <div id='tasks'>
229           <TaskOne />
230           <hr />
231           <!-- Остальные компоненты задач -->
232         </div>
233       </div>
234     )
235   }
```

ПРИЛОЖЕНИЕ Б

Дополнительный код к заданию «Максимальный поток»

```
/**
 * Вспомогательный метод, находящий расширяющий путь в
 * остаточном графе, построенный на основе BFS.
 */
private findAugmentingPath(
  residualGraph: Map<string, Map<string, number>>,
  source: string,
  sink: string
): string[] | null {
  const visited: Set<string> = new Set()
  const queue: string[] = [source]
  const parent: Map<string, string | null> = new Map()
  parent.set(source, null)

  while (queue.length > 0) {
    const u = queue.shift()!
    for (const v of residualGraph.get(u)!.keys()) {
      if (!visited.has(v) && residualGraph.get(u)!.get(v)! > 0) {
        visited.add(v)
        parent.set(v, u)
        queue.push(v)

        if (v === sink) {
          // пересоздать расширяющий путь
          const path: string[] = []
          let current = v
          while (current !== null) {
            path.unshift(current)
            current = parent.get(current)!
          }
          return path
        }
      }
    }
  }

  return null // не нашлось расширяющего пути
}
```