

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

**ТЕОРИЯ ГРАФОВ**  
**ОТЧЕТ О ПРАКТИКЕ**

студента 3 курса 351 группы  
направления 09.03.04 — Программная инженерия  
факультета КНиИТ  
Янущика Ильи Андреевича

Проверил  
старший преподаватель

\_\_\_\_\_

М. С. Портенко

Саратов 2023

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	4
1 Минимальные требования для класса Граф .....	5
1.1 Условие задания .....	5
1.2 Примеры исходного кода .....	5
1.3 Примеры входных и выходных данных (формат JSON) .....	8
2 2. Список смежности Ia .....	12
2.1 Условие задания .....	12
2.2 Примеры исходного кода .....	12
2.3 Краткое описание алгоритма .....	12
2.4 Примеры входных и выходных данных .....	12
3 2. Список смежности Ia .....	14
3.1 Условие задания .....	14
3.2 Примеры исходного кода .....	14
3.3 Краткое описание алгоритма .....	14
3.4 Примеры входных и выходных данных .....	14
4 4. Список смежности Ib: несколько графов .....	16
4.1 Условие задания .....	16
4.2 Примеры исходного кода .....	16
4.3 Краткое описание алгоритма .....	16
4.4 Примеры входных и выходных данных .....	16
5 5. Обходы графа II .....	19
5.1 Условие задания .....	19
5.2 Примеры исходного кода .....	19
5.3 Краткое описание алгоритма .....	20
5.4 Примеры входных и выходных данных .....	21
6 2. Список смежности Ia .....	22
6.1 Условие задания .....	22
6.2 Примеры исходного кода .....	22
6.3 Краткое описание алгоритма .....	23
6.4 Примеры входных и выходных данных .....	23
7 7. Каркас III .....	25
7.1 Условие задания .....	25
7.2 Примеры исходного кода .....	25

	7.3	Примеры входных и выходных данных .....	26
8	8.	Веса IV а .....	28
	8.1	Условие задания .....	28
	8.2	Примеры исходного кода .....	28
	8.3	Примеры входных и выходных данных .....	30
9	9.	Веса IV б .....	31
	9.1	Условие задания .....	31
	9.2	Примеры исходного кода .....	31
	9.3	Краткое описание алгоритма .....	32
	9.4	Примеры входных и выходных данных .....	32
10	10.	Веса IV с .....	34
	10.1	Условие задания .....	34
	10.2	Примеры исходного кода .....	34
	10.3	Краткое описание алгоритма .....	36
	10.4	Примеры входных и выходных данных .....	36
11	11.	Максимальный поток V .....	38
	11.1	Условие задания .....	38
	11.2	Примеры исходного кода .....	38
	11.3	Примеры входных и выходных данных .....	39
12	12.	Творческая задача .....	41
	12.1	Условие задания .....	41
	12.2	Примеры исходного кода .....	41
	12.3	Работа приложения .....	42
	Приложение А	Консольный интерфейс пользователя .....	47
	Приложение Б	Творческое задание .....	52

## 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 задания выполнены на языке 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 рёбер на основе списка смежности)
2. Конструкторы (не менее 3-х):
  - а) добавляющие вершину
  - б) добавляющие ребро (дугу)
  - в) удаляющие вершину,
  - г) удаляющие ребро (дугу),
  - д) выводящие список смежности в файл (в том числе в пригодном для чтения конструктором формате).
3. Методы:
  - а) Конструктор по умолчанию, создающий пустой граф
  - б) Конструктор, заполняющий данные графа из файла
  - в) Конструктор-копию (аккуратно, не все сразу делают именно копию)
  - г) 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 рёбра (дуги) и просматривать текущий список смежности графа.

## 1.2 Примеры исходного кода

Код ниже описывает класс граф с конструктором класса и геттерами

```
1 export default class Graph {
2   private vertices: IVertex;
3   private isOrient: boolean;
4   public constructor(isOrient?: boolean, vertices?: IVertex) {
5     this.vertices = vertices ?? {};
6     this.isOrient = Boolean(isOrient);
7   }
```

```

8
9  public getAllVertices() {
10     return Object.keys(this.vertices);
11 }
12
13 public getAllEdges() {
14     return Object.entries(this.vertices)
15         .map(([key, value]) =>
16             value.map((v) => ({
17                 weight: v.weight,
18                 from: key,
19                 to: v.value,
20             })),
21         )
22         .flat() as Array<IEdge>;
23 }
24
25 public getIsOrient() {
26     return this.isOrient;
27 }
28 }

```

Методы класса реализующие добавление вершины и ребра:

```

1  public addVertex(vertex: KeyType) {
2      if (!this.vertices[vertex]) this.vertices[vertex] = [];
3  }
4
5  public addEdge(vertex1: KeyType, vertex2: KeyType, weight?: number) {
6      if (!(vertex1 in this.vertices) || !(vertex2 in this.vertices)) {
7          throw new Error('В графе нет таких вершин');
8      }
9
10     if (
11         (!this.vertices[vertex1].find((el) => el.value === vertex2) &&
12             !this.isOrient) ||
13         this.isOrient
14     ) {
15         this.vertices[vertex1].push({
16             value: vertex2,
17             weight: weight ?? 0,

```

```

18     });
19 }
20
21 if (
22     !this.vertices[vertex2].find((el) => el.value === vertex1) &&
23     !this.isOrient
24 ) {
25     this.vertices[vertex2].push({
26         value: vertex1,
27         weight: weight ?? 0,
28     });
29 }
30 }

```

Удаление вершины / ребра:

```

1  public deleteVertex(vertex: KeyType) {
2      const deletedVertices = this.vertices[vertex];
3      if (deletedVertices) {
4          deletedVertices.forEach((v) => {
5              const arr = this.vertices[v.value];
6              const deletedIndex = arr.findIndex((el) => el.value === vertex);
7              if (deletedIndex !== -1) arr.splice(deletedIndex, 1);
8          });
9      }
10     delete this.vertices[vertex];
11 }
12 public deleteEdge(vertex1: KeyType, vertex2: KeyType) {
13     if (!(vertex1 in this.vertices) || !(vertex2 in this.vertices)) {
14         throw new Error('В графе нет таких вершин');
15     }
16
17     if (this.vertices[vertex1].find((el) => el.value === vertex2)) {
18         const arr = this.vertices[vertex1];
19         const deletedIndex = arr.findIndex((el) => el.value === vertex2);
20         if (deletedIndex !== -1) this.vertices[vertex1].splice(deletedIndex,
21             ↪ 1);
22     }
23
24     if (
25         this.vertices[vertex2].find((el) => el.value === vertex1) &&

```

```

25     !this.isOrient
26   ) {
27     const arr = this.vertices[vertex2];
28     const deletedIndex = arr.findIndex((el) => el.value === vertex1);
29     if (deletedIndex !== -1) this.vertices[vertex2].splice(deletedIndex,
        ↪ 1);
30   }
31 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 1.3 Примеры входных и выходных данных (формат JSON)

Пример файла для загрузки в граф (ориентированный)

```

1 {
2   "isOrient": true,
3   "vertices": {
4     "A": [
5       {
6         "value": "B",
7         "weight": 0
8       },
9       {
10        "value": "C",
11        "weight": 0
12      },
13      {
14        "value": "F",
15        "weight": 0
16      }
17    ],
18    "B": [],
19    "C": [
20      {
21        "value": "D",
22        "weight": 0
23      },
24      {
25        "value": "E",

```



```

26         "weight": 0
27     }
28 ],
29     "D": [
30         {
31             "value": "C",
32             "weight": 0
33         }
34     ],
35     "E": [],
36     "F": [
37         {
38             "value": "G",
39             "weight": 0
40         }
41     ],
42     "G": [],
43     "H": []
44 }
45 }
46

```

Пример файла для загрузки в граф (неориентированный)

```

1 {
2     "isOrient": false,
3     "vertices": {
4         "A": [
5             {
6                 "value": "B",
7                 "weight": 0
8             },
9             {
10                "value": "C",
11                "weight": 0
12            },
13            {
14                "value": "F",
15                "weight": 0
16            }
17        ],

```

```

18     "B": [],
19     "C": [
20         {
21             "value": "D",
22             "weight": 0
23         },
24         {
25             "value": "E",
26             "weight": 0
27         }
28     ],
29     "D": [
30         {
31             "value": "C",
32             "weight": 0
33         }
34     ],
35     "E": [],
36     "F": [
37         {
38             "value": "G",
39             "weight": 0
40         }
41     ],
42     "G": [],
43     "H": []
44 }
45 }
46

```

### Пример выгрузки в файл

```

1 {
2     "isOrient": false,
3     "vertices": {
4         "A": [
5             {
6                 "value": "B",
7                 "weight": 0
8             },
9             {

```

```

10         "value": "C",
11         "weight": 0
12     },
13     {
14         "value": "F",
15         "weight": 0
16     }
17 ],
18 "B": [],
19 "C": [
20     {
21         "value": "D",
22         "weight": 0
23     },
24     {
25         "value": "E",
26         "weight": 0
27     }
28 ],
29 "D": [
30     {
31         "value": "C",
32         "weight": 0
33     }
34 ],
35 "E": [],
36 "F": [
37     {
38         "value": "G",
39         "weight": 0
40     }
41 ],
42 "G": [],
43 "H": []
44 }
45 }
46

```

## 2 2. Список смежности Ia

### 2.1 Условие задания

**Вариант 2:** Вывести полустепень исхода данной вершины орграфа.

### 2.2 Примеры исходного кода

Для нахождения полустепени исхода данной вершины, был описан метод класса - `degreeOfOutcome`.

```
1 public degreeOfOutcome(vertex: KeyType) {  
2     if (!this.isOrient) throw new Error('Граф неориентированный!');  
3     if (!(vertex in this.vertices)) throw new Error('Такой вершины нет!');  
4     else return this.vertices[vertex].length;  
5 }
```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 2.3 Краткое описание алгоритма

Алгоритм `degreeOfOutcome` принимает название вершины (`vertex`), проверяет, ориентированный ли граф и есть ли такая вершина в нём, затем выдает длину списка вершин, смежных с данной (это и есть полустепень исхода для данной вершины в орграфе)

### 2.4 Примеры входных и выходных данных

#### Граф ориентированный

```
1 const edges = [  
2     ['A', 'B', 7],  
3     ['A', 'C', 8],  
4     ['B', 'D', 2],  
5     ['C', 'B', 11],  
6     ['C', 'E', 9],  
7     ['C', 'D', 6],  
8     ['D', 'E', 11],  
9     ['D', 'F', 9],  
10    ['E', 'F', 10],  
11 ];  
12 vertices.forEach((v) => graphOrient.addVertex(v));
```

```

13 edges.forEach((e) => graphOrient.addEdge(e[0], e[1], e[2] as number));
14
15 console.log(graphOrient.degreeOfOutcome('A'));
16 // Выходные данные: 2

```

## Граф неориентированный

```

1  const edges = [
2    ['A', 'B', 7],
3    ['A', 'C', 8],
4    ['B', 'D', 2],
5    ['C', 'B', 11],
6    ['C', 'E', 9],
7    ['C', 'D', 6],
8    ['D', 'E', 11],
9    ['D', 'F', 9],
10   ['E', 'F', 10],
11 ];
12 vertices.forEach((v) => graph.addVertex(v));
13 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
14
15 console.log(graph.degreeOfOutcome('A'));
16 // Выводит предупреждение: 'Граф неориентированный!'

```

## 3 2. Список смежности Ia

### 3.1 Условие задания

**Вариант 12:** Вывести те вершины орграфа, в которых есть петли.

### 3.2 Примеры исходного кода

Для нахождения петель, был описан метод класса - printLoopsVertices.

```
1 public printLoopsVertices() {
2     if (!this.isOrient) throw new Error('Граф неориентированный!');
3     const loops = Object.entries(this.vertices).filter(([key, arr]) =>
4         arr.find((el) => el.value === key),
5     );
6     if (!loops.length) console.log('Петель нет');
7     else loops.forEach((l) => console.log(l[0] + ' '));
8 }
```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 3.3 Краткое описание алгоритма

Алгоритм printLoopsVertices проходит по всем вершинам в графе и проверяет, есть ли среди смежных вершин текущей вершины она же сама и заносит её в результирующий список.

### 3.4 Примеры входных и выходных данных

#### Граф неориентированный

```
1 const edges = [
2     ['A', 'B', 7],
3     ['A', 'A', 7],
4     ['A', 'C', 8],
5     ['B', 'D', 2],
6     ['C', 'B', 11],
7     ['C', 'E', 9],
8     ['C', 'D', 6],
9     ['D', 'E', 11],
10    ['D', 'F', 9],
11    ['E', 'F', 10],
12 ];
```

```

13 vertices.forEach((v) => graph.addVertex(v));
14 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
15
16 // Выводит предупреждение: 'Граф неориентированный!'

```

## Граф ориентированный

```

1  const edges = [
2    ['A', 'B', 7],
3    ['A', 'A', 7],
4    ['A', 'C', 8],
5    ['B', 'D', 2],
6    ['C', 'B', 11],
7    ['C', 'E', 9],
8    ['C', 'D', 6],
9    ['D', 'E', 11],
10   ['D', 'F', 9],
11   ['E', 'F', 10],
12 ];
13 vertices.forEach((v) => graphOrient.addVertex(v));
14 edges.forEach((e) => graphOrient.addEdge(e[0], e[1], e[2] as number));
15
16 // Выходные данные: 'A'

```

## 4 4. Список смежности Iб: несколько графов

### 4.1 Условие задания

**Вариант 6:** Построить граф, являющийся пересечением двух заданных.

### 4.2 Примеры исходного кода

Для получения нового графа из пересечением двух заданных, был описан метод класса - intersection.

```
1  public intersection(graph: Graph) {
2      const interGraph = new Graph();
3      Object.entries(this.vertices).forEach(([key, arr]) => {
4          if (key in graph.vertices) {
5              interGraph.addVertex(key);
6              const graphArr = graph.vertices[key];
7              arr.forEach((el) => {
8                  if (graphArr.find((grEl) => el.value === grEl.value)) {
9                      interGraph.addVertex(el.value);
10                     interGraph.addEdge(key, el.value);
11                 }
12             });
13         }
14     });
15     return interGraph;
16 }
```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 4.3 Краткое описание алгоритма

Алгоритм принимает на вход граф, с которым будет выполняться операция пересечения. Для начала создаём новый граф - interGraph, являющийся результатом пересечения. После чего добавляем в него только те вершины и рёбра, которые присутствуют в обоих графах

### 4.4 Примеры входных и выходных данных

**Граф 1:**



```

1  const newGr = new Graph();
2  newGr.addVertex('A');
3  newGr.addVertex('B');
4  newGr.addVertex('C');
5  newGr.addEdge('A', 'C');
6  newGr.addEdge('A', 'B');
7  newGr.print();
8  const newGr2 = new Graph();
9  newGr2.addVertex('A');
10 newGr2.addVertex('B');
11 newGr2.addVertex('C');
12 newGr2.addEdge('A', 'C');
13 newGr2.print();
14 const newGr3 = newGr.intersection(newGr2);
15 newGr3.print();
16
17 // Результат:
18 {
19   A: [ { value: 'C', weight: 0 } ],
20   C: [ { value: 'A', weight: 0 } ],
21   B: []
22 }

```

## Граф 2:

```

1  const newGr = new Graph();
2  newGr.addVertex('A');
3  newGr.addVertex('B');
4  newGr.addVertex('C');
5  newGr.addEdge('A', 'C');
6  newGr.addEdge('A', 'B');
7  newGr.print();
8  const newGr2 = new Graph();
9  newGr2.addVertex('A');
10 newGr2.addVertex('B');
11 newGr2.addVertex('C');
12 newGr2.addEdge('B', 'C');
13 newGr2.print();
14 const newGr3 = newGr.intersection(newGr2);
15 newGr3.print();
16

```

17 // Результат:

18 { A: [], B: [], C: [] }

## 5 5. Обходы графа II

### 5.1 Условие задания

**Вариант 8:** Найти путь, соединяющий вершины u1 и u2 и не проходящий через вершину v.

### 5.2 Примеры исходного кода

Для нахождения пути, были описаны два метода класса - dfs и findShortestPath.

```
1  dfs(startVertex: KeyType, badVertex: KeyType) {
2      let list = this.vertices; // список смежности
3      let stack = [startVertex]; // стек вершин для перебора
4      let visited = { [startVertex]: 1 }; // посещенные вершины
5
6      // кратчайшее расстояние от стартовой вершины
7      let distance = { [startVertex]: 0 };
8      // предыдущая вершина в цепочке
9      let previous = { [startVertex]: null };
10
11     function handleVertex(vertex: KeyType) {
12         // получаем список смежных вершин
13         let reversedNeighboursList = [...list[vertex]].reverse();
14
15         reversedNeighboursList.forEach((neighbour) => {
16             if (!visited[neighbour.value] && neighbour.value !== badVertex) {
17                 // отмечаем вершину как посещенную
18                 visited[neighbour.value] = 1;
19                 // добавляем в стек
20                 stack.push(neighbour.value);
21                 previous[neighbour.value] = vertex;
22                 // сохраняем расстояние
23                 distance[neighbour.value] = distance[vertex] + 1;
24             }
25         });
26     }
27
28     // перебираем вершины из стека, пока он не опустеет
29     while (stack.length) {
30         let activeVertex = stack.pop();
31         handleVertex(activeVertex);
```

```

32     }
33
34     return { distance, previous };
35 }
36
37 findShortestPath(
38     startVertex: KeyType,
39     finishVertex: KeyType,
40     badVertex: KeyType,
41 ) {
42     let result = this.dfs(startVertex, badVertex);
43
44     if (!(finishVertex in result.previous))
45         throw new Error(
46             `Нет пути из вершины ${startVertex} в вершину ${finishVertex}`,
47         );
48
49     let path = [];
50
51     let currentVertex = finishVertex;
52
53     while (currentVertex !== startVertex) {
54         path.unshift(currentVertex);
55         currentVertex = result.previous[currentVertex];
56     }
57
58     path.unshift(startVertex);
59
60     return path;
61 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 5.3 Краткое описание алгоритма

Метод `dfs` реализует алгоритм обхода графа в глубину, метод `findShortestPath` принимает начальную вершину, конечную и ту, которую нельзя посещать, и благодаря тому, что в ходе обхода мы сохраняли цепочку посещений, восстанавливает нужный путь.

## 5.4 Примеры входных и выходных данных

### Граф 1:

```
1  const edges = [  
2    ['A', 'B'],  
3    ['A', 'C'],  
4    ['C', 'D'],  
5    ['C', 'E'],  
6    ['A', 'F'],  
7    ['F', 'G'],  
8    ['B', 'C'],  
9    ['B', 'U'],  
10   ['U', 'E'],  
11   ['F', 'C'],  
12 ];  
13 vertices.forEach((v) => graph.addVertex(v));  
14 edges.forEach((e) => graph.addEdge(e[0], e[1]));  
15 resultDeepSearch = graph.findShortestPath('A', 'E', 'B');  
16  
17 // Результат: [ 'A', 'C', 'E' ]
```

### Граф 2:

```
1  const edges = [  
2    ['A', 'B'],  
3    ['A', 'F'],  
4    ['F', 'G'],  
5    ['B', 'C'],  
6    ['B', 'U'],  
7    ['U', 'E'],  
8    ['F', 'C'],  
9  ];  
10 vertices.forEach((v) => graph.addVertex(v));  
11 edges.forEach((e) => graph.addEdge(e[0], e[1]));  
12 resultDeepSearch = graph.findShortestPath('F', 'U', 'B');  
13  
14 // Результат: 'Нет пути из вершины F в вершину U'
```

## 6 2. Список смежности Ia

### 6.1 Условие задания

**Вариант 31:** Вывести длины кратчайших (по числу дуг) путей от всех вершин до u.

### 6.2 Примеры исходного кода

Для поиска длин кратчайших (по числу дуг) путей, был описан метод класса - bfs.

```
1  bfs(startVertex: KeyType, badVertex: KeyType) {
2      if (startVertex === badVertex) {
3          throw new Error(`Начальная вершина совпадает с нежелательной`);
4      }
5      let list = this.vertices;
6      let queue = [startVertex];
7      let visited = { [startVertex]: 1 };
8
9      // кратчайшее расстояние от стартовой вершины
10     let distance = { [startVertex]: 0 };
11     // предыдущая вершина в цепочке
12     let previous = { [startVertex]: null };
13
14     function handleVertex(vertex: KeyType) {
15         let neighboursList = list[vertex];
16
17         neighboursList.forEach((neighbour) => {
18             if (!visited[neighbour.value] && neighbour.value !== badVertex) {
19                 visited[neighbour.value] = 1;
20                 queue.push(neighbour.value);
21                 // сохраняем предыдущую вершину
22                 previous[neighbour.value] = vertex;
23                 // сохраняем расстояние
24                 distance[neighbour.value] = distance[vertex] + 1;
25             }
26         });
27     }
28
29     // перебираем вершины из очереди, пока она не опустеет
30     while (queue.length) {
31         let activeVertex = queue.shift();
```

```

32     handleVertex(activeVertex);
33 }
34
35 return { distance, previous };
36 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 6.3 Краткое описание алгоритма

Метод bfs реализует алгоритм обхода графа в ширину, формируя объект distance - кратчайшие расстояния от стартовой вершины до всех остальных.

### 6.4 Примеры входных и выходных данных

#### Граф 1:

```

1  const edges = [
2    ['A', 'B'],
3    ['A', 'C'],
4    ['C', 'D'],
5    ['C', 'E'],
6    ['A', 'F'],
7    ['F', 'G'],
8    ['B', 'C'],
9    ['B', 'U'],
10   ['U', 'E'],
11   ['F', 'C'],
12 ];
13 vertices.forEach((v) => graph.addVertex(v));
14 edges.forEach((e) => graph.addEdge(e[0], e[1]));
15 resultWidthSearch = graph.bfs('A', null).distance;
16
17 // Результат: { A: 0, B: 1, C: 1, F: 1, U: 2, D: 2, E: 2, G: 2 }

```

#### Граф 2:

```

1  const edges = [
2    ['A', 'B'],
3    ['A', 'F'],
4    ['F', 'G'],

```

```
5    ['B', 'C'],
6    ['B', 'U'],
7    ['U', 'E'],
8    ['F', 'C'],
9 ];
10 vertices.forEach((v) => graph.addVertex(v));
11 edges.forEach((e) => graph.addEdge(e[0], e[1]));
12 resultWidthSearch = graph.bfs('F', null).distance;
13
14 // Результат: { A: 0, B: 1, C: 1, F: 1, U: 2, D: 2, E: 2, G: 2 }
```



## 7 7. Каркас III

### 7.1 Условие задания

**Алгоритм Краскала:** Дан взвешенный неориентированный граф из N вершин и M ребер. Требуется найти в нем каркас минимального веса.

### 7.2 Примеры исходного кода

Для нахождения каркаса, был описан метод класса - `kruskal`.

```
1  public kruskal() {
2      if (this.getIsOrient()) {
3          throw new Error('Граф ориентированный!');
4      }
5      //Инициализируем новый граф, который будет содержать минимальное остовное
6      ↪ дерево исходного графа.
7      const minimumSpanningTree = new Graph();
8      const sortingCallbacks = {
9          compareCallback: (graphEdgeA: IEdge, graphEdgeB: IEdge) => {
10              if (graphEdgeA.weight === graphEdgeB.weight) return 1;
11              return graphEdgeA.weight <= graphEdgeB.weight ? -1 : 1;
12          },
13      };
14      const sortedEdges = this.getAllEdges().sort(
15          sortingCallbacks.compareCallback,
16      );
17      //Создаем непересекающиеся множества для всех вершин графа.
18      const keyCallback = (graphVertex: KeyType) => graphVertex;
19      const disjointSet = new DisjointSet(keyCallback);
20      this.getAllVertices().forEach((graphVertex) => {
21          disjointSet.makeSet(graphVertex);
22      });
23      // Пройдемся по всем ребрам, начиная с минимального, и попробуем их
24      ↪ добавить.
25      // к минимальному связующему дереву. Критерием добавления ребра будет то,
26      ↪ будет ли
27      // образует цикл или нет (если соединяет две вершины из одной
28      ↪ непересекающейся
29      // установлено или нет).
30      for (let edgeIndex = 0; edgeIndex < sortedEdges.length; edgeIndex += 1) {
31          const currentEdge = sortedEdges[edgeIndex];
32          if (!disjointSet.inSameSet(currentEdge.from, currentEdge.to)) {
```

```

29      // Объединяем два подмножества в одно.
30      disjointSet.union(currentEdge.from, currentEdge.to);
31      minimumSpanningTree.addVertex(currentEdge.from);
32      minimumSpanningTree.addVertex(currentEdge.to);
33      //Добавляем это ребро к связующему дереву.
34      minimumSpanningTree.addEdge(
35          currentEdge.from,
36          currentEdge.to,
37          currentEdge.weight,
38      );
39  }
40  }
41  return minimumSpanningTree;
42  }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

## 7.3 Примеры входных и выходных данных

### Граф 1:

```

1  const vertices = ['A', 'B', 'C', 'D', 'E', 'F'];
2  const edges = [
3      ['A', 'B', 7],
4      ['A', 'C', 8],
5      ['B', 'D', 2],
6      ['C', 'B', 11],
7      ['C', 'E', 9],
8      ['C', 'D', 6],
9      ['D', 'E', 11],
10     ['D', 'F', 9],
11     ['E', 'F', 10],
12 ];
13 vertices.forEach((v) => graph.addVertex(v));
14 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
15
16 export const tree = graph.kruskal();
17
18 // Результат:
19 {

```

```

20   B: [ { value: 'D', weight: 2 }, { value: 'A', weight: 7 } ],
21   D: [
22     { value: 'B', weight: 2 },
23     { value: 'C', weight: 6 },
24     { value: 'F', weight: 9 }
25   ],
26   C: [ { value: 'D', weight: 6 }, { value: 'E', weight: 9 } ],
27   A: [ { value: 'B', weight: 7 } ],
28   E: [ { value: 'C', weight: 9 } ],
29   F: [ { value: 'D', weight: 9 } ]
30 }

```

## Граф 2:

```

1  const vertices = ['A', 'B', 'C', 'D', 'E', 'F'];
2  const edges = [
3    ['A', 'B', 45],
4    ['A', 'C', 65],
5    ['C', 'B', 3],
6    ['C', 'D', 46],
7    ['D', 'E', 3],
8    ['D', 'F', 45],
9    ['E', 'F', 8],
10 ];
11 vertices.forEach((v) => graph.addVertex(v));
12 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
13
14 export const tree = graph.kruskal();
15
16 // Результат:
17 {
18   B: [ { value: 'C', weight: 3 }, { value: 'A', weight: 45 } ],
19   C: [ { value: 'B', weight: 3 }, { value: 'D', weight: 46 } ],
20   D: [ { value: 'E', weight: 3 }, { value: 'C', weight: 46 } ],
21   E: [ { value: 'D', weight: 3 }, { value: 'F', weight: 8 } ],
22   F: [ { value: 'E', weight: 8 } ],
23   A: [ { value: 'B', weight: 45 } ]
24 }

```

## 8 8. Веса IV а

### 8.1 Условие задания

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

### 8.2 Примеры исходного кода

Для нахождения пути, был описан метод класса - *dijkstra*.

```
1  public dijkstra(startVertex: KeyType) {
2      // Вспомогательные переменные инициализации, которые нам понадобятся для
   ↪ алгоритма Дейкстры.
3      const distances = {};
4      const visitedVertices = {};
5      const previousVertices = {};
6      const queue = new PriorityQueue();
7
8      // Инициализируем все расстояния с бесконечностью, предполагая, что
9      // в данный момент мы не можем достичь ни одной вершины, кроме начальной.
10     this.getAllVertices().forEach((vertex) => {
11         distances[vertex] = Infinity;
12         previousVertices[vertex] = null;
13     });
14
15     // Мы уже находимся в начальной вершине, поэтому расстояние до нее равно
   ↪ нулю.
16     distances[startVertex] = 0;
17
18     // Инициализация очереди вершин
19     queue.add(startVertex, distances[startVertex]);
20
21     // Перебирать приоритетную очередь вершин, пока она не станет пустой.
22     while (!queue.isEmpty()) {
23         // Получить следующую ближайшую вершину.
24         const currentVertex = queue.poll();
25
26         // Перебрать каждого непосещенного соседа текущей вершины.
27         this.vertices[currentVertex].forEach((neighbor) => {
28             // Не посещайте уже посещенные вершины.
29             if (!visitedVertices[neighbor.value]) {
30                 // Обновить расстояния до каждого соседа от текущей вершины.
```

```

31     const existingDistanceToNeighbor = distances[neighbor.value];
32     const distanceToNeighborFromCurrent =
33         distances[currentVertex] + neighbor.weight;
34
35     // Если мы нашли более короткий путь к соседу - обновите его.
36     if (distanceToNeighborFromCurrent < existingDistanceToNeighbor) {
37         distances[neighbor.value] = distanceToNeighborFromCurrent;
38
39         // Изменить приоритет соседа в очереди, так как он мог стать
40         ↪ ближе.
41         if (queue.hasValue(neighbor.value)) {
42             queue.changePriority(neighbor.value,
43                 ↪ distances[neighbor.value]);
44         }
45
46         // Запомните предыдущую ближайшую вершину.
47         previousVertices[neighbor.value] = currentVertex;
48     }
49
50     // Добавьте соседа в очередь для дальнейшего посещения.
51     if (!queue.hasValue(neighbor.value)) {
52         queue.add(neighbor.value, distances[neighbor.value]);
53     }
54 }
55
56 // Добавьте текущую вершину к посещенным, чтобы в дальнейшем не
57 ↪ посещать ее повторно.
58 visitedVertices[currentVertex] = currentVertex;
59 }
60
61 // Возвращает набор кратчайших расстояний до всех вершин и набор
62 // кратчайших путей ко всем вершинам графа.
63 return {
64     distances,
65     previousVertices,
66 };
67 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

## 8.3 Примеры входных и выходных данных

### Граф 1:

```
1  const edges = [  
2    ['A', 'B', 7],  
3    ['A', 'C', 8],  
4    ['B', 'D', 2],  
5    ['C', 'B', 11],  
6    ['C', 'E', 9],  
7    ['C', 'D', 6],  
8    ['D', 'E', 11],  
9    ['D', 'F', 9],  
10   ['E', 'F', 10],  
11  ];  
12  vertices.forEach((v) => graph.addVertex(v));  
13  edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));  
14  export const { previousVertices, distances } = graph.dijkstra('A');  
15  // Результат:  
16  { A: 0, B: 7, C: 8, D: 9, E: 17, F: 18 }
```

### Граф 2:

```
1  const edges = [  
2    ['A', 'B', 763],  
3    ['A', 'C', 358],  
4    ['B', 'D', 624],  
5    ['C', 'B', 656],  
6    ['C', 'E', 342],  
7    ['C', 'D', 356],  
8    ['D', 'E', 141],  
9    ['E', 'F', 160],  
10  ];  
11  vertices.forEach((v) => graph.addVertex(v));  
12  edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));  
13  export const { previousVertices, distances } = graph.dijkstra('A');  
14  // Результат:  
15  { A: 0, B: 7, C: 8, D: 9, E: 17, F: 18 }
```

## 9 9. Веса IV b

### 9.1 Условие задания

**Вариант 7:** N-периферией для вершины называется множество вершин, расстояние от которых до заданной вершины больше N. Определить N-периферию для заданной вершины графа.

### 9.2 Примеры исходного кода

Для нахождения пути, был описан метод класса - bellmanFord.

```
1 public bellmanFord(startVertex: KeyType, limit: number) {
2   const distances = {};
3   const previousVertices = {};
4
5   // Инициализируем все расстояния с бесконечностью, предполагая,
6   // что в данный момент мы не можем достичь ни одной вершины, кроме
7   // ↪ начальной.
8   distances[startVertex] = 0;
9   this.getAllVertices().forEach((vertex) => {
10     previousVertices[vertex] = null;
11     if (vertex !== startVertex) {
12       distances[vertex] = Infinity;
13     }
14   });
15
16   // Нам понадобится (|V| - 1) итераций.
17   for (
18     let iteration = 0;
19     iteration < this.getAllVertices().length - 1;
20     iteration += 1
21   ) {
22     // Во время каждой итерации проходят все вершины.
23     Object.keys(distances).forEach((vertex) => {
24       // Пройдите через все ребра вершин.
25       this.vertices[vertex].forEach((neighbor) => {
26         // Выясним, меньше ли расстояние до соседа в этой
27         // итерации, чем в предыдущей.
28         const distanceToVertex = distances[vertex];
29         const distanceToNeighbor = distanceToVertex + neighbor.weight;
30         if (distanceToNeighbor < distances[neighbor.value]) {
31           distances[neighbor.value] = distanceToNeighbor;
32         }
33       });
34     });
35   }
```

```

31         previousVertices[neighbor.value] = vertex;
32     }
33     });
34 });
35 }
36
37 const perefery = {};
38 Object.entries(distances).forEach(([key, dist]) => {
39     if (Number(dist) > limit) {
40         perefery[key] = dist;
41     }
42 });
43
44 return {
45     distances,
46     previousVertices,
47     perefery,
48 };
49 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 9.3 Краткое описание алгоритма

Алгоритм представляет собой классическую реализацию алгоритма Беллмана-Форда с одной модификацией: на вход подаётся параметр `limit` (для нахождения переферии), после того, как был составлен объект с расстояниями, выводим только те вершины, расстояния от которых до заданной больше `limit`.

### 9.4 Примеры входных и выходных данных

#### Граф 1:

```

1  const edges = [
2    ['A', 'B', 7],
3    ['A', 'C', 8],
4    ['B', 'D', 2],
5    ['C', 'B', 11],
6    ['C', 'E', 9],
7    ['C', 'D', 6],

```



```

8   ['D', 'E', 11],
9   ['D', 'F', 9],
10  ['E', 'F', 10],
11 ];
12 vertices.forEach((v) => graph.addVertex(v));
13 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
14 export const bellmanFord = graph.bellmanFord('A', 8);
15 // Результат:
16 { D: 9, E: 17, F: 18 }

```

## Граф 2:

```

1  const edges = [
2    ['A', 'B', 763],
3    ['A', 'C', 358],
4    ['B', 'D', 624],
5    ['C', 'B', 656],
6    ['C', 'E', 342],
7    ['C', 'D', 356],
8    ['D', 'E', 141],
9    ['E', 'F', 160],
10 ];
11 vertices.forEach((v) => graph.addVertex(v));
12 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
13 export const bellmanFord = graph.bellmanFord('A', 8);
14 // Результат:
15 { B: 763, C: 358, D: 714, E: 700, F: 860 }

```

## 10 10. Веса IV с

### 10.1 Условие задания

**Вариант 1:** Определить, существует ли путь длиной не более  $L$  между двумя заданными вершинами графа.

### 10.2 Примеры исходного кода

Для выполнения задания был описан метод класса - floydWarshall.

```
1 public floydWarshall(v1: KeyType, v2: KeyType, L: number) {
2     const vertices = this.getAllVertices();
3
4     // Инициализируем матрицу предыдущих вершин с нулями, что означает
5     // ↪ отсутствие
6     // предыдущие вершины, которые дадут нам кратчайший путь.
7     const nextVertices = Array(vertices.length)
8         .fill(null)
9         .map(() => {
10             return Array(vertices.length).fill(null);
11         });
12
13     // Начальная матрица расстояний с бесконечностью означает,
14     // что путей между вершинами пока не существует.
15
16     const distances = Array(vertices.length)
17         .fill(null)
18         .map(() => {
19             return Array(vertices.length).fill(Infinity);
20         });
21
22     //Инициализируем матрицу расстояний.
23     //А также инициализируем предыдущие вершины с ребер.
24     vertices.forEach((startVertex, startIndex) => {
25         vertices.forEach((endVertex, endIndex) => {
26             if (startVertex === endVertex) {
27                 distances[startIndex][endIndex] = 0;
28             } else {
29                 // Найдём ребро между начальной и конечной вершинами
30                 const edge = this.vertices[startVertex].find(
31                     (v) => v.value === endVertex,
```

```

32
33     if (edge) {
34         // Если существует ребро от вершины с startIndex до вершины с
           ↪ endIndex
35         // сохраним расстояние и предыдущую вершину.
36         distances[startIndex][endIndex] = edge.weight;
37         nextVertices[startIndex][endIndex] = startVertex;
38     } else {
39         distances[startIndex][endIndex] = Infinity;
40     }
41 }
42 });
43 });
44
45 // Теперь перейдем к сути алгоритма.
46 // Возьмем все пары вершин (от начала до конца) и попробуем проверить,
           ↪ если
47 // между ними существует более короткий путь через среднюю вершину.
           ↪ Средняя вершина также может
48 // быть одной из вершин графа. Как вы можете видеть, теперь у нас будет
           ↪ три
49 // циклических прохода по всем вершинам графа: для начальной, конечной и
           ↪ средней вершины.
50 vertices.forEach((middleVertex, middleIndex) => {
51     // Путь начинается с startVertex с помощью startIndex.
52     vertices.forEach((startVertex, startIndex) => {
53         // Путь заканчивается на endVertex с помощью endIndex.
54         vertices.forEach((endVertex, endIndex) => {
55             // Сравним существующее расстояние от startVertex до endVertex с
           ↪ расстоянием
56             // от startVertex до endVertex, но через middleVertex.
57             // Сохраняем кратчайшее расстояние и предыдущую вершину
58             const distViaMiddle =
59                 distances[startIndex][middleIndex] +
60                 distances[middleIndex][endIndex];
61
62             if (distances[startIndex][endIndex] > distViaMiddle) {
63                 // Мы нашли кратчайший проход через среднюю вершину.
64                 distances[startIndex][endIndex] = distViaMiddle;
65                 nextVertices[startIndex][endIndex] = middleVertex;
66             }

```

```

67     });
68     });
69     });
70
71     const indexV1 = this.getAllVertices().indexOf(String(v1));
72     const indexV2 = this.getAllVertices().indexOf(String(v2));
73     const dist = distances[indexV1][indexV2];
74     const isExist = dist <= L;
75     return { distances, nextVertices, dist, isExist };
76 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 10.3 Краткое описание алгоритма

Алгоритм представляет собой классическую реализацию алгоритма Флойда с одной модификацией: на вход подаётся параметр L и вершины v1, v2. После составления матрицы расстояний, находим в ней элемент, определяющий расстояние между v1 и v2, и определяем, не превышает ли оно L.

### 10.4 Примеры входных и выходных данных

#### Граф 1:

```

1  const edges = [
2    ['A', 'B', 7],
3    ['A', 'C', 8],
4    ['B', 'D', 2],
5    ['C', 'B', 11],
6    ['C', 'E', 9],
7    ['C', 'D', 6],
8    ['D', 'E', 11],
9    ['D', 'F', 9],
10   ['E', 'F', 10],
11 ];
12 vertices.forEach((v) => graphOrient.addVertex(v));
13 edges.forEach((e) => graphOrient.addEdge(e[0], e[1], e[2] as number));
14 export const floydWarshall = graphOrient.floydWarshall('F', 'D', 10);
15 // Результат: false

```

#### Граф 2:

```
1  const edges = [  
2    ['A', 'B', 7],  
3    ['A', 'C', 8],  
4    ['B', 'D', 2],  
5    ['C', 'B', 11],  
6    ['C', 'E', 9],  
7    ['C', 'D', 6],  
8    ['D', 'E', 11],  
9    ['D', 'F', 9],  
10   ['E', 'F', 10],  
11  ];  
12  vertices.forEach((v) => graphOrient.addVertex(v));  
13  edges.forEach((e) => graphOrient.addEdge(e[0], e[1], e[2] as number));  
14  export const floydWarshall = graphOrient.floydWarshall('A', 'F', 20);  
15  // Результат: true, путь: 18
```

## 11 11. Максимальный поток V

### 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

Основной метод, реализующий алгоритм Форда — Фалкерсона:

```
1 public maxStream(init: number, end: number) {
2   let matrixVertices = this.getVerticesMatrix();
3   const routeInit = [Infinity, -1, init]; // первая метка маршрута (a,
   ↪ from, vertex)
4   const routeStreams: Array<number> = []; // максимальные потоки найденных
   ↪ маршрутов
5   let j = init;
6   while (j !== -1) {
7     let startVertex = init; // стартовая вершина (нумерация с нуля)
8     const routes = [routeInit]; // метки маршрута
9     const visited = [init]; // множество просмотренных вершин
10    while (startVertex !== end) {
11      j = this.getMaxVertex(startVertex, matrixVertices, visited); //
   ↪ выбираем вершину с наибольшей пропускной способностью
12      // если следующих вершин нет
13      if (j === -1) {
14        if (startVertex == init) {
15          //и мы на истоке, то завершаем поиск маршрутов
16          break;
17        } else {
18          startVertex = routes.pop()[2];
19        }
20      } else {
21        let currentStream = // определяем текущий поток
22          matrixVertices[startVertex][j][2] == 1
23            ? matrixVertices[startVertex][j][0]
24            : matrixVertices[startVertex][j][1];
25
26        routes.push([currentStream, j, startVertex]); // добавляем метку
   ↪ маршрута
27        visited.push(j); // запоминаем вершину как просмотренную
28        // если дошли до стока
```

```

29         if (j === end) {
30             routeStreams.push(this.getMaxFlow(routes)); // находим
                 ↪ максимальную пропускную способность маршрута
31             matrixVertices = this.updateMatrix(
32                 matrixVertices,
33                 routes,
34                 routeStreams[routeStreams.length - 1],
35             ); // обновляем веса дуг
36             break;
37         }
38         startVertex = j;
39     }
40 }
41 }
42 return routeStreams.reduce((el, accum) => accum + el);
43 }

```

Использование всех методов в консольном интерфейсе пользователя представлено в приложении А.

### 11.3 Примеры входных и выходных данных

#### Граф 1:

```

1  const vertices = ['1', '2', '3', '4', '5'];
2  const edges = [
3      ['1', '2', 20],
4      ['1', '4', 10],
5      ['1', '3', 30],
6      ['2', '3', 40],
7      ['2', '5', 30],
8      ['3', '4', 10],
9      ['3', '5', 20],
10     ['4', '5', 20],
11 ];
12 vertices.forEach((v) => graph.addVertex(v));
13 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));
14 // Результат: 60

```

#### Граф 2:

```
1  const edges = [  
2    ['1', '2', 30],  
3    ['1', '4', 20],  
4    ['1', '3', 40],  
5    ['2', '3', 50],  
6    ['2', '5', 40],  
7    ['3', '4', 20],  
8    ['3', '5', 30],  
9    ['4', '5', 30],  
10 ];  
11 vertices.forEach((v) => graph.addVertex(v));  
12 edges.forEach((e) => graph.addEdge(e[0], e[1], e[2] as number));  
13 // Результат: 90
```



## 12 12. Творческая задача

### 12.1 Условие задания

Творческое задание, включающее визуализацию графов. В качестве творческого задания была выбрана визуализация алгоритмов обхода в ширину и в глубину на языках программирования TypeScript и CSS и языке гипертекстовой разметки HTML с использованием библиотеки react.

### 12.2 Примеры исходного кода

Фрагмент основной компоненты, реализующей визуализацию графа

```
1 export default function Graph() {
2   const [delay, setDelay] = useState('1000');
3   // eslint-disable-next-line @typescript-eslint/no-unused-vars
4   const [graph, setGraph] = useState(() => {
5     const newGraph = new GraphClass(false);
6     const vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'U'];
7     const edges = [
8       ['A', 'B'],
9       ['A', 'C'],
10      ['C', 'D'],
11      ['C', 'E'],
12      ['A', 'F'],
13      ['F', 'G'],
14      ['B', 'C'],
15      ['B', 'U'],
16      ['U', 'E'],
17      ['F', 'C'],
18    ];
19    vertices.forEach((v) => newGraph.addVertex(v));
20    edges.forEach((e) => newGraph.addEdge(e[0], e[1]));
21    return newGraph;
22  });
23  const [d3Data, setD3Data] = useState<GraphData<any, any> | null>(null);
24  const [stack, setStack] = useState<Array<string>>([]);
25  const [queue, setQueue] = useState<Array<string>>([]);
26  const [isProceed, setIsProceed] = useState(false);
27  const [isShowAlert, setIsShowAlert] = useState(false);
28  const [isStarting, setIsStarting] = useState(false);
29  const [visited, setVisited] = useState<Array<string>>([]);
30  useEffect(() => {
```

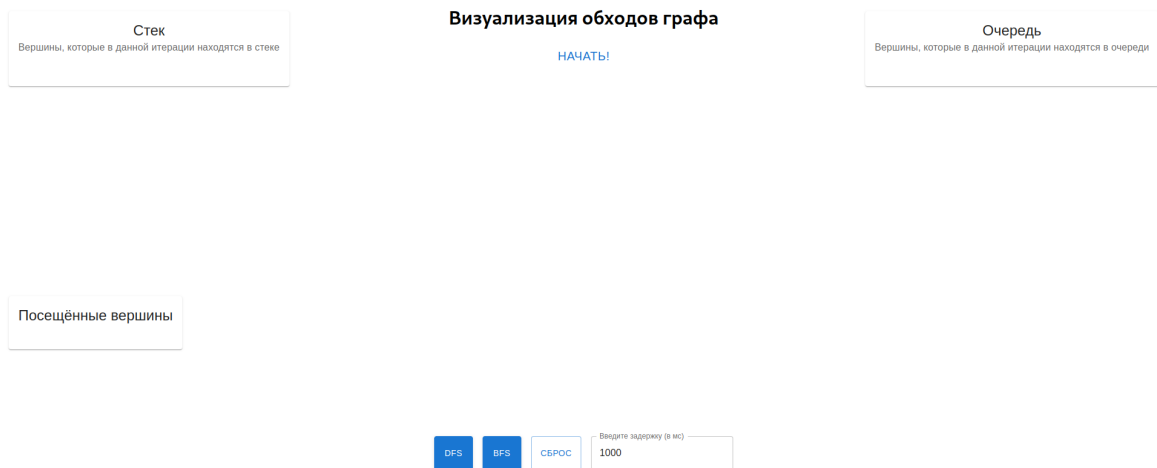
```

31     setD3Data(graph.getD3Data());
32 }, [graph]);
33 }

```

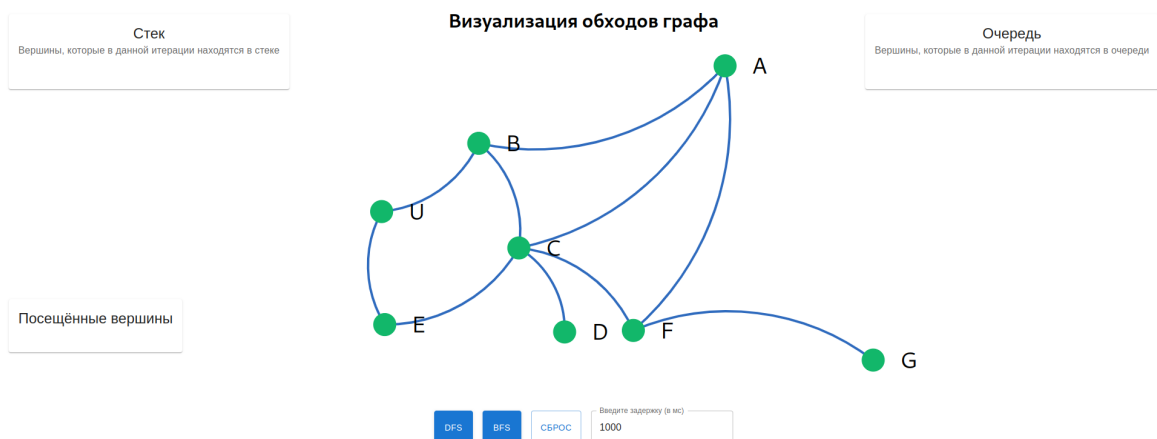
Программный код представлен в приложении Б.

## 12.3 Работа приложения



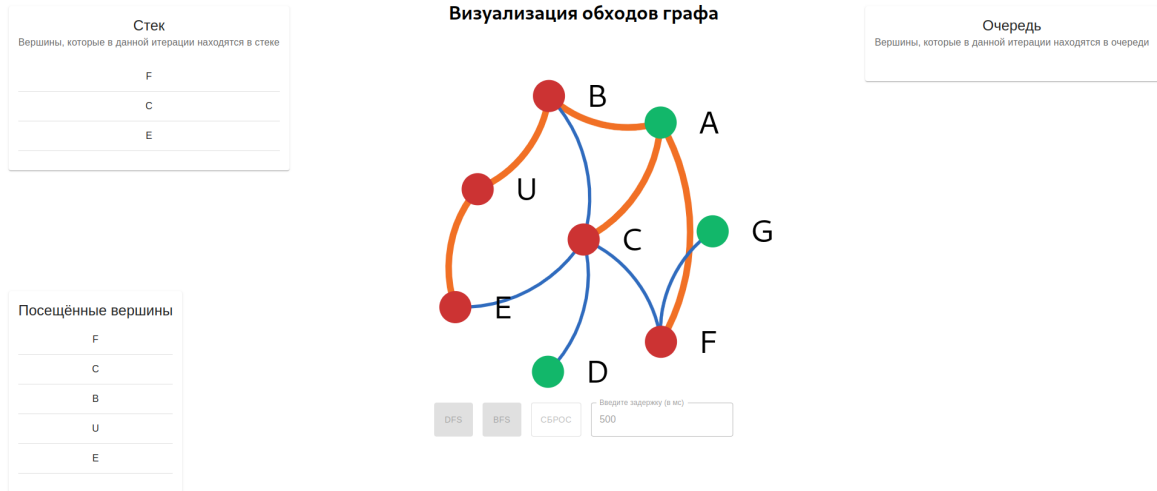
Стартовое окно

Нажмём на кнопку «НАЧАТЬ!», после чего увидим граф. Для удобства пользователя вершины можно перетаскивать. Граф можно приближать колёсиком мыши и двигать.



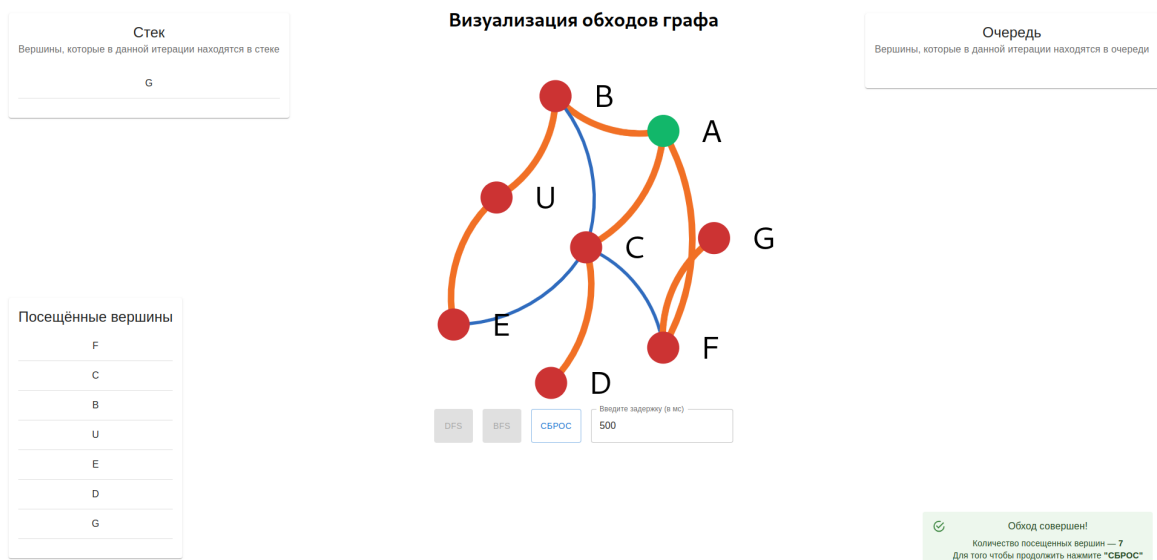
Отображение графа

Нажмём на кнопку «DFS», после чего увидим визуализацию обхода в глубину. Посещенные вершины заносятся в соответствующую таблицу, на экране можем видеть все вершины стека в текущий момент времени. Посещенные вершины и рёбра меняют цвет и толщину.



Обход в глубину

После завершения работы, увидим окно, сообщающее о результатах выполнения. Нажав на кнопку «СБРОС», вернем граф в исходный вид и очистим стек и массив посещенных вершин.



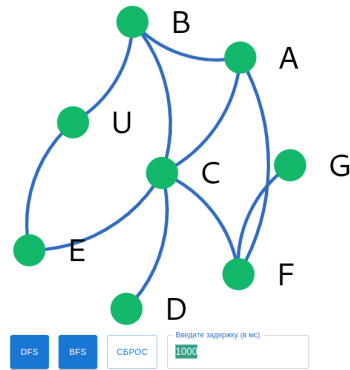
Окончание работы алгоритма

Для удобства пользователь может вводить время задержки работы алгоритма

Стек  
Вершины, которые в данной итерации находятся в стеке

Посещённые вершины

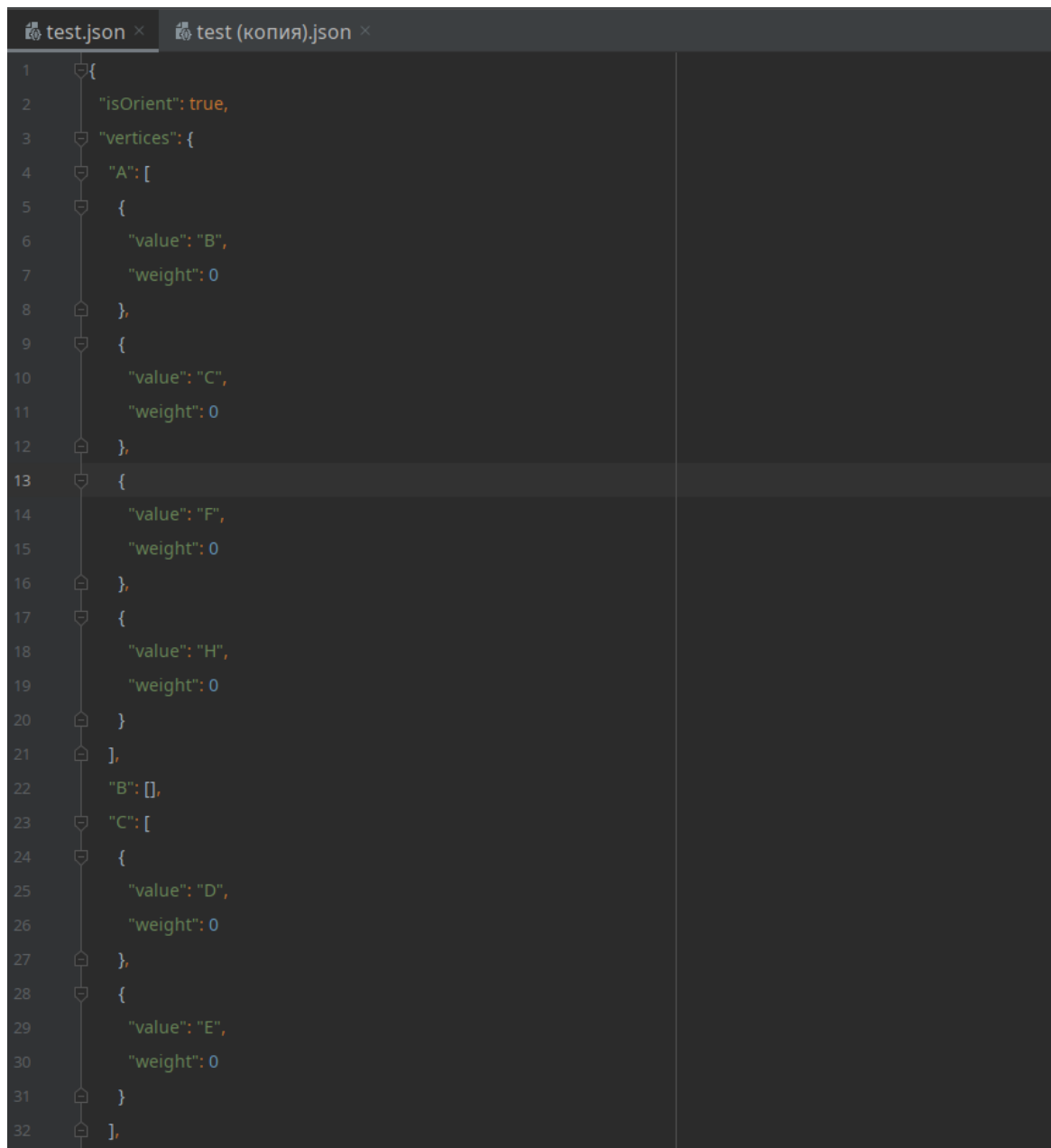
### Визуализация обходов графа



Очередь  
Вершины, которые в данной итерации находятся в очереди

Поле с задержкой

Помимо прочего можем загрузить свой файл в формате JSON с графом.



```
1  {
2    "isOrient": true,
3    "vertices": {
4      "A": [
5        {
6          "value": "B",
7          "weight": 0
8        },
9        {
10         "value": "C",
11         "weight": 0
12       },
13       {
14         "value": "F",
15         "weight": 0
16       },
17       {
18         "value": "H",
19         "weight": 0
20       }
21     ],
22     "B": [],
23     "C": [
24       {
25         "value": "D",
26         "weight": 0
27       },
28       {
29         "value": "E",
30         "weight": 0
31       }
32     ],
```

Пример загружаемого файла

После загрузки файла и нажатия кнопки «ПРОЧЕСТЬ ФАЙЛ» увидим, что заданный граф отобразился на экране.

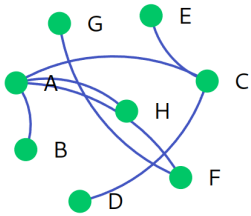
Стек

Вершины, которые в данной итерации находятся в стеке

Посещённые вершины

Визуализация обходов графа

НАЧАТЬ!



DFS

BFS

СБРОС

Введите задержку (в мс)  
1000

Выберите файл

test.json

ПРОЧЕСТЬ ФАЙЛ

Очередь

Вершины, которые в данной итерации находятся в очереди

Загруженный граф

## ПРИЛОЖЕНИЕ А

### Консольный интерфейс пользователя

```
1 import Graph from './graphs/graph';
2 import JSONGraph from './graph.json';
3 import JSONOrientGraph from './orientGraph.json';
4 import Traversals from './traversals';
5 import { tree } from './kruskal';
6 import { maxStream } from './stream';
7 import {
8   distances,
9   previousVertices,
10  bellmanFord,
11  floydWarshall,
12 } from './weight';
13
14 console.log(
15   '////////////////////Неориентированный
    ↪ граф////////////////////',
16 );
17 const graph = new Graph();
18 const vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
19 const edges = [
20   ['A', 'B'],
21   ['A', 'C'],
22   ['C', 'D'],
23   ['C', 'E'],
24   ['A', 'F'],
25   ['F', 'G'],
26 ];
27 vertices.forEach((v) => graph.addVertex(v));
28 edges.forEach((e) => graph.addEdge(e[0], e[1]));
29
30 graph.print();
31
32 graph.deleteVertex('A');
33
34 graph.print();
35
36 graph.deleteEdge('C', 'D');
```

```

38 graph.print();
39
40 console.log(
41     '////////////////////////////////Ориентированный граф////////////////////////////////',
42 );
43 const orientGraph = new Graph(true);
44
45 vertices.forEach((v) => orientGraph.addVertex(v));
46 edges.forEach((e) => orientGraph.addEdge(e[0], e[1]));
47
48 orientGraph.print();
49
50 orientGraph.deleteVertex('A');
51
52 orientGraph.print();
53
54 orientGraph.deleteEdge('C', 'D');
55
56 orientGraph.print();
57
58 console.log(
59     '////////////////////////////////Неориентированный с
        ↳ файла////////////////////////////////',
60 );
61
62 const graphFromFile = new Graph(JSONGraph.isOrient, JSONGraph.vertices);
63 graphFromFile.print();
64
65 graphFromFile.addEdge('A', 'H');
66
67 graphFromFile.print();
68
69 console.log(
70     '////////////////////////////////Ориентированный с
        ↳ файла////////////////////////////////',
71 );
72
73 const graphOrientFromFile = new Graph(
74     JSONOrientGraph.isOrient,
75     JSONOrientGraph.vertices,
76 );

```



```

77 graphOrientFromFile.print();
78
79 graphOrientFromFile.addEdge('A', 'H');
80 graphOrientFromFile.print();
81 graphOrientFromFile.write('test.json');
82 console.log(
83     '////////////////////////Пересечение графов////////////////////////',
84 );
85 const newGr = new Graph();
86 newGr.addVertex('A');
87 newGr.addVertex('B');
88 newGr.addVertex('C');
89 newGr.addEdge('A', 'C');
90 newGr.addEdge('A', 'B');
91 newGr.print();
92 const newGr2 = new Graph();
93 newGr2.addVertex('A');
94 newGr2.addVertex('B');
95 newGr2.addVertex('C');
96 newGr2.addEdge('B', 'C');
97 newGr2.print();
98 const newGr3 = newGr.intersection(newGr2);
99 newGr3.print();
100
101 let readline = require('readline');
102 let rl = readline.createInterface({
103     input: process.stdin,
104     output: process.stdout,
105     prompt: '>',
106 });
107
108 console.log('////////////////////////Обходы
    ↪ графов////////////////////////');
109 console.log(Traversals.resultDeepSearch);
110 console.log(Traversals.resultWidthSearch);
111
112 console.log('////////////////////////Краскал////////////////////////');
113 tree.print();
114
115 console.log('////////////////////////Дейкстра////////////////////////');
116 console.log(distances);

```

```

117 console.log(previousVertices);
118
119 console.log('//////////Беллман-фloyd//////////');
120 console.log(bellmanFord.distances);
121 console.log(bellmanFord.previousVertices);
122 console.log(bellmanFord.perefery);
123
124 console.log('//////////Флойд//////////');
125 console.log(floydWarshall.distances);
126 console.log(floydWarshall.dist);
127 console.log(floydWarshall.isExist);
128
129 console.log('//////////Поток//////////');
130 console.log(maxStream);
131
132 console.log(
133     'Создать граф: create, создать ориентированный граф createOrient, добавить
    ↪ вершину: vertex, добавить ребро: edge, ' +
134     'вывести на экран: print, write: вывести в файл, degree - вывести степень
    ↪ полуисходов, loops - вывести количество петель',
135 );
136
137 let isVertex = false;
138 let isEdge = false;
139 let isWrite = false;
140 let isDegree = false;
141 let CLIGraph: Graph;
142 rl.prompt();
143 rl.on('line', (line: string) => {
144     if (line === 'create') {
145         isVertex = false;
146         isEdge = false;
147         isDegree = false;
148         CLIGraph = new Graph();
149     } else if (line === 'createOrient') {
150         isVertex = false;
151         isEdge = false;
152         isDegree = false;
153         CLIGraph = new Graph(true);
154     } else if (line === 'vertex') {
155         isVertex = true;

```

```

156     isEdge = false;
157     isWrite = false;
158     isDegree = false;
159 } else if (line === 'edge') {
160     isVertex = false;
161     isWrite = false;
162     isEdge = true;
163     isDegree = false;
164 } else if (line === 'print') {
165     CLIGraph.print();
166 } else if (line === 'write') {
167     isVertex = false;
168     isEdge = false;
169     isWrite = true;
170     isDegree = false;
171 } else if (line === 'degree') {
172     isVertex = false;
173     isEdge = false;
174     isWrite = false;
175     isDegree = true;
176 } else if (line === 'loops') {
177     CLIGraph.printLoopsVertices();
178 } else if (isVertex) {
179     CLIGraph.addVertex(line);
180 } else if (isEdge) {
181     const arr = line.split(' ');
182     CLIGraph.addEdge(arr[0], arr[1]);
183 } else if (isWrite) {
184     CLIGraph.write(line);
185     isWrite = false;
186 } else if (isDegree) {
187     const deg = CLIGraph.degreeOfOutcome(line);
188     console.log(deg);
189     isDegree = false;
190 }
191 }).on('close', () => {
192     console.log('exit');
193     process.exit(0);
194 });

```

## ПРИЛОЖЕНИЕ Б

### Творческое задание

```
1 import { useEffect, useState } from 'react';
2 import GraphClass from 'graphs/graph';
3 import {
4   Graph as GraphD3,
5   GraphConfiguration,
6   GraphData,
7 } from 'react-d3-graph';
8 import {
9   Alert,
10  AlertTitle,
11  Box,
12  Button,
13  Card,
14  CardContent,
15  Divider,
16  List,
17  ListItem,
18  ListItemText,
19  TextField,
20  Typography,
21 } from '@mui/material';
22
23 const myConfig: Partial<
24   GraphConfiguration<
25     | { id: string; color: string; size: number }
26     | { id: string; color?: undefined; size?: undefined },
27     { source: string; target: string }
28   >
29 > = {
30   nodeHighlightBehavior: true,
31   width: 1500,
32   height: 600,
33   node: {
34     color: '#12b76a',
35     size: 300,
36     labelProperty: 'id',
37     highlightStrokeColor: 'blue',
38     fontSize: 19,
```

```

39   },
40   link: {
41     type: 'CURVE_SMOOTH',
42     highlightColor: 'lightblue',
43   },
44 };
45
46 export default function Graph() {
47   const [delay, setDelay] = useState('1000');
48   // eslint-disable-next-line @typescript-eslint/no-unused-vars
49   const [graph, setGraph] = useState(() => {
50     const newGraph = new GraphClass(false);
51     const vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'U'];
52     const edges = [
53       ['A', 'B'],
54       ['A', 'C'],
55       ['C', 'D'],
56       ['C', 'E'],
57       ['A', 'F'],
58       ['F', 'G'],
59       ['B', 'C'],
60       ['B', 'U'],
61       ['U', 'E'],
62       ['F', 'C'],
63     ];
64     vertices.forEach((v) => newGraph.addVertex(v));
65     edges.forEach((e) => newGraph.addEdge(e[0], e[1]));
66     return newGraph;
67   });
68   const [d3Data, setD3Data] = useState<GraphData<any, any> | null>(null);
69   const [stack, setStack] = useState<Array<string>>([]);
70   const [queue, setQueue] = useState<Array<string>>([]);
71   const [isProceed, setIsProceed] = useState(false);
72   const [isShowAlert, setIsShowAlert] = useState(false);
73   const [isStarting, setIsStarting] = useState(false);
74   const [visited, setVisited] = useState<Array<string>>([]);
75   useEffect(() => {
76     setD3Data(graph.getD3Data());
77   }, [graph]);
78
79   useEffect(() => {

```

```

80     if (
81         d3Data?.nodes.length === visited.length + 2 &&
82         queue.length <= 1 &&
83         stack.length <= 1
84     ) {
85         setIsProceed(false);
86         setIsShowAlert(true);
87     }
88 }, [d3Data?.nodes.length, queue.length, stack.length, visited.length]);
89
90 function dfsChange(vertex: string, parentVertex: string) {
91     console.log(vertex, parentVertex);
92     setD3Data((prev) => {
93         if (prev) {
94             return {
95                 links: prev.links.map((link) => ({
96                     ...link,
97                     strokeWidth:
98                         (link.source === parentVertex && link.target === vertex) ||
99                         (link.source === vertex && link.target === parentVertex)
100                         ? 4
101                         : link.strokeWidth,
102                     color:
103                         (link.source === parentVertex && link.target === vertex) ||
104                         (link.source === vertex && link.target === parentVertex)
105                         ? '#f17025'
106                         : link.color,
107                 })),
108                 nodes: prev.nodes.map((n) => ({
109                     ...n,
110                     color: vertex === n.id ? '#c33' : n.color,
111                 })),
112             };
113         }
114         return null;
115     });
116 }
117 function changeStack(newStack: Array<string>) {
118     setStack(newStack);
119 }
120

```

```

121 function changeQueue(newQueue: Array<string>) {
122     setQueue((prev) => [...newQueue]);
123 }
124
125 function changeVisited(v: string) {
126     setVisited((prev) => [...prev, v]);
127 }
128
129 const startDfs = () => {
130     setIsProceed(true);
131     graph.dfs('A', '_', Number(delay), dfsChange, changeStack,
132         ↪ changeVisited);
133 };
134
135 const startBfs = () => {
136     setIsProceed(true);
137     graph.bfs('A', '_', Number(delay), dfsChange, changeQueue,
138         ↪ changeVisited);
139 };
140
141 const revert = () => {
142     setIs showAlert(false);
143     setD3Data(graph.getD3Data());
144     setVisited([]);
145     setStack([]);
146     setQueue([]);
147 };
148
149 return (
150     <div>
151         {isShowAlert && (
152             <Alert
153                 severity="success"
154                 sx={{ position: 'absolute', zIndex: 1000, right: 20, bottom: 20 }}
155             >
156                 <AlertTitle>Обход совершен!</AlertTitle>
157                 Количество посещенных вершин - <strong>{visited.length}</strong>
158                 <div>
159                     Для того чтобы продолжить нажмите <strong>"СБРОС"</strong>
160                 </div>
161             </Alert>

```

```

160     })
161     {!isStarting && (
162         <Button
163             sx={{ fontSize: 20 }}
164             onClick={() => {
165                 setIsStarting(true);
166                 revert();
167             }}
168         >
169             Начать!
170         </Button>
171     )}
172
173     {d3Data && (
174         <GraphD3
175             id="graph-id" // id is mandatory, if no id is defined rd3g will
176                 ↳ throw an error
177             data={d3Data as any}
178             config={myConfig}
179         />
180     )}
181
182     <Box sx={{ display: 'flex', justifyContent: 'center', gap: 2 }}>
183         <Button
184             disabled={isProceed || visited.length !== 0}
185             variant={'contained'}
186             onClick={startDfs}
187         >
188             DFS
189         </Button>
190         <Button
191             disabled={isProceed || visited.length !== 0}
192             variant={'contained'}
193             onClick={startBfs}
194         >
195             BFS
196         </Button>
197         <Button disabled={isProceed} variant={'outlined'} onClick={revert}>
198             Сброс
199         </Button>
200         <TextField

```



```

200     disabled={isProceed}
201     variant="outlined"
202     value={delay}
203     onChange={(e) => setDelay(e.target.value)}
204     label={'Введите задержку (в мс)'}
205   </TextField>
206 </Box>
207
208 <Card sx={{ position: 'absolute', left: 10, top: 30 }}>
209   <CardContent>
210     <Typography variant="h5" component="div">
211       Стек
212     </Typography>
213     <Typography sx={{ mb: 1.5 }} color="text.secondary">
214       Вершины, которые в данной итерации находятся в стеке
215     </Typography>
216     <List component="nav">
217       {stack.map((el) => (
218         <>
219           <ListItem>
220             <ListItemText
221               sx={{ display: 'flex', justifyContent: 'center' }}
222               primary={el}
223             />
224           </ListItem>
225           <Divider />
226         </>
227       )}}
228     </List>
229   </CardContent>
230 </Card>
231 <Card sx={{ position: 'absolute', right: 10, top: 30 }}>
232   <CardContent>
233     <Typography variant="h5" component="div">
234       Очередь
235     </Typography>
236     <Typography sx={{ mb: 1.5 }} color="text.secondary">
237       Вершины, которые в данной итерации находятся в очереди
238     </Typography>
239     <List component="nav">
240       {queue.map((el) => (

```

```

241         <>
242         <ListItem>
243             <ListItemText
244                 sx={{ display: 'flex', justifyContent: 'center' }}
245                 primary={el}
246             />
247         </ListItem>
248         <Divider />
249     </>
250     )})
251 </List>
252 </CardContent>
253 </Card>
254 <Card sx={{ position: 'absolute', left: 10, top: 500 }}>
255     <CardContent>
256         <Typography variant="h5" component="div">
257             Посещённые вершины
258         </Typography>
259         <List component="nav">
260             {visited.map((el) => (
261                 <>
262                     <ListItem>
263                         <ListItemText
264                             sx={{ display: 'flex', justifyContent: 'center' }}
265                             primary={el}
266                         />
267                     </ListItem>
268                     <Divider />
269                 </>
270             )})
271         </List>
272     </CardContent>
273 </Card>
274 </div>
275 );
276 }

```