

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

**DEVOPS: ЕГО РОЛЬ В СОВРЕМЕННОЙ РАЗРАБОТКЕ
ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

ОТЧЁТ

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Устюшина Богдана Антоновича

Проверено:

доцент, к. п. н.

М. С. Портенко

Саратов 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Задание 1: создание класса графов	4
2 Задание 2: Список смежности Ia	8
3 Задание 3: Список смежности Ia	9
4 Задание 4: Список смежности Ib: несколько графов	10
5 Задание 5: Обходы графа II	13
6 Задание 6: Обходы графа II	14
7 Задание 7: Каркас III	16
8 Задание 8: Веса IVc	18
9 Задание 9: Веса IVc	20
10 Задание 10: Веса IVc	25
11 Задание 11: Максимальный поток	29
Приложение А Файлы класса графов	31
Приложение Б Файлы класса приложения	39
Приложение В Файлы Tasks (заданий)	48
Приложение Г Файлы алгоритмов	60
Приложение Д Файл Main.cpp	73
Приложение Е Используемые для тестирования файлы	77

ВВЕДЕНИЕ

Отчёт по теории графов. **Используемый язык** – C++.

Каждая функция в файле `Tasks.cpp` (приложение В) выполняет одно из заданий. Они используют алгоритмы, описанные в файле `Algos.cpp` (приложение Г).

В файле `Graph.cpp` (приложение А) представлена реализация графа, не пересекающаяся с консольным описанием интерфейса. Описание консольного интерфейса описано в файле `App.cpp` (приложение Б).

Все файлы взаимодействуют друг с другом посредством подключения `.h` (хедеров) соответствующих файлов.

Функции интерфейса используются в функции `main` (приложение Д) в главном файле `main.cpp`.

В отчёте прилагаются изображение графа и скриншоты результата работы программы с файлами из приложения Е, в которых хранятся графы.

1 Задание 1: Создание класса графов

В первом задании предлагалось создать свой собственный класс графов, в который входят:

1. Структуру хранения графа (список смежности или список рёбер)
2. Конструкторы (пустой граф, граф из файла, конструктор-копию)
3. Методы (добавления/удаления вершин, добавления/удаления рёбер или дуг, вывод списка смежности в файл, т.е. сохранения графа)
4. Должны поддерживаться как ориентированные/неориентированные графы
5. Должен быть создан минималистичный консольный интерфейс

Структура хранения выбрана как словарь словарей (`map<map<string, int>>`): то есть у каждой вершины есть `map`, который соответствует вершинам, с которыми она смежна (в случае неориентированного графа) или вершинам, в которые идут дуги (в случае ориентированного графа).

Представлены конструкторы на основе:

1. Bool-переменной `isOriented`, которая определяет, является ли граф ориентированным или нет (поддержка различия двух графов);
2. Файла, в котором граф представляется в json-формате с помощью библиотеки *nlohmann-json*;
3. Ссылки на константное значение класса графа – конструктор копирования.

Написать примеры создания графа

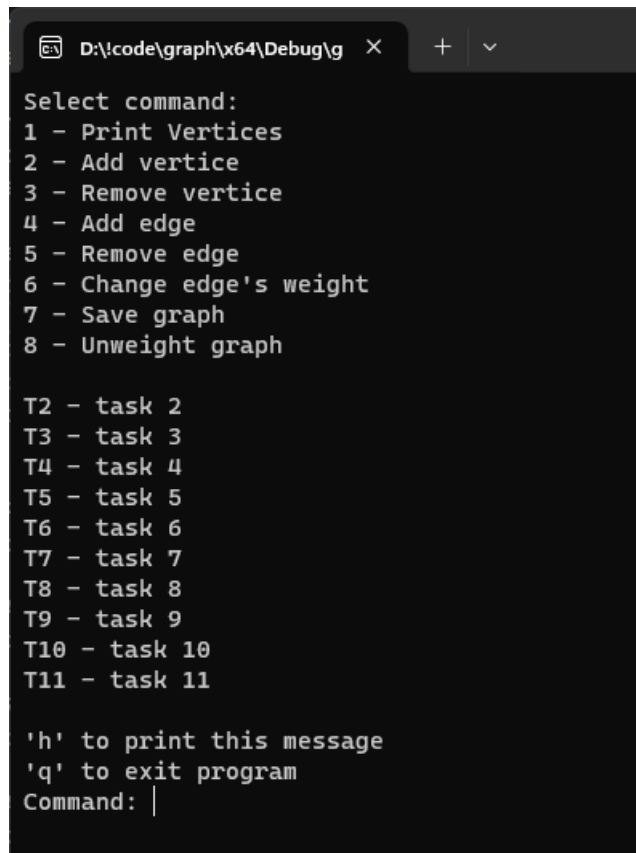
В нём созданы:

1. `enum string_code` – введённая пользователем строка на основе функции Hashing будет преобразовываться в номер для выполнения определённой команды.
2. `CommandMessage()` – выводит справку о поддерживаемых командах, появляется при старте программы и вызове функции `help (h)`
3. `CreateGraph` – интерфейс создания графа (вызывается при пустом файле, указанном в переменной `DATA_FILE_1` имени графа, используемой в главном файле `main.cpp`).
4. Функция `PrintVertices` – выводит все вершины графа в виде списка в консоли.
5. Функция `AddVertice` – добавляет вершину к текущему графу, в котором запущена программа.

6. Функция `RemoveVertice` – удаляет вершину из текущего графа.
7. Функция `AddEdge` – добавляет ребро/дугу к текущему графу.
8. Функция `RemoveEdge` – удаляет ребро/дугу у текущего графа.
9. Функция `ChangeWeight` – изменяет вес ребра в текущем графе.
10. Функция `Unweight` – изменяет вес всех рёбер на 1.

Все исключительные ситуации (ребра при изменении веса или удалении не существует, не существует одной из вершин при добавлении ребра, добавление уже существующей вершины и прочие) обрабатываются внутри класса ещё на этапе выполнения метода классом графа. Методы интерфейса `App` просто вызывают для переданного им параметра (графа) методы, реализованные внутри класса графа, что позволяет перенести всю логику обработки некорректных ситуаций в класс графа.

Также дополнительно в `App.cpp` существует функция `is_number`, проверяющий корректность введённого числа при вводе веса рёбер (они могут быть лишь целыми числами, положительными, отрицательным или нулём).



```

Select command:
1 - Print Vertices
2 - Add vertice
3 - Remove vertice
4 - Add edge
5 - Remove edge
6 - Change edge's weight
7 - Save graph
8 - Unweight graph

T2 - task 2
T3 - task 3
T4 - task 4
T5 - task 5
T6 - task 6
T7 - task 7
T8 - task 8
T9 - task 9
T10 - task 10
T11 - task 11

'h' to print this message
'q' to exit program
Command: |
```

Рисунок 1 – Стартовое сообщение

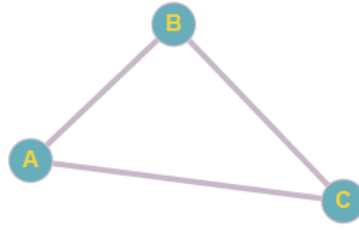


Рисунок 2 – Граф 1

```
q - to exit program
Command: 1
A: (B, 1) (C, 1)
B: (A, 1) (C, 1)
C: (A, 1) (B, 1)
Command: 2
Enter vertex to add: D
Vertex added succesfully
Command: 1
A: (B, 1) (C, 1)
B: (A, 1) (C, 1)
C: (A, 1) (B, 1)
D:
Command: 3
Enter vertex to remove: C
Vertex removed successfully
Command: 1
A: (B, 1)
B: (A, 1)
D:
Command: 4
Enter start vertex: A
Enter end vertex: B
Enter edge weight (default = 1):
Edge already exists between these 2 vertices
Command: |
```

Рисунок 3 – Пример использования функций 1

```
Command: 5
Enter start vertice: B
Enter end vertice: D
No such edge in graph
Command: 5
Enter start vertice: A
Enter end vertice: B
Edge removed successfully
Command: 1
A: (D, 1)
B:
D: (A, 1)
Command: 6
Enter start vertice: A
Enter end vertice: D
Enter edge weight (default = 1): 5
Weight changed successfully
Command: 1
A: (D, 5)
B:
D: (A, 5)
Command: |
```

Рисунок 4 – Пример использования функций 2

2 Задание 2: Список смежности Ia

Задание: Вывести все вершины орграфа, смежные с данной.

Обычный перебор используемого map для получения всех соседних вершин. Здесь понятие смежности я отождествляю с существованием дуги.

```
Command: 1
a: (e, 1) (m, 1)
b: (c, 1)
c: (b, 9) (c, 3)
d: (k, 1)
e: (a, 1) (d, 1) (k, 1)
g:
k:
l: (a, 1)
m: (n, 6)
n:
Command: T2
Enter vertex to print: a
a: (e, 1) (m, 1)
Command: T2
Enter vertex to print: g
No adjacent vertices!
Command: T2
Enter vertex to print: f
No adjacent vertices!
Command: |
```

Рисунок 5 – Задание 2

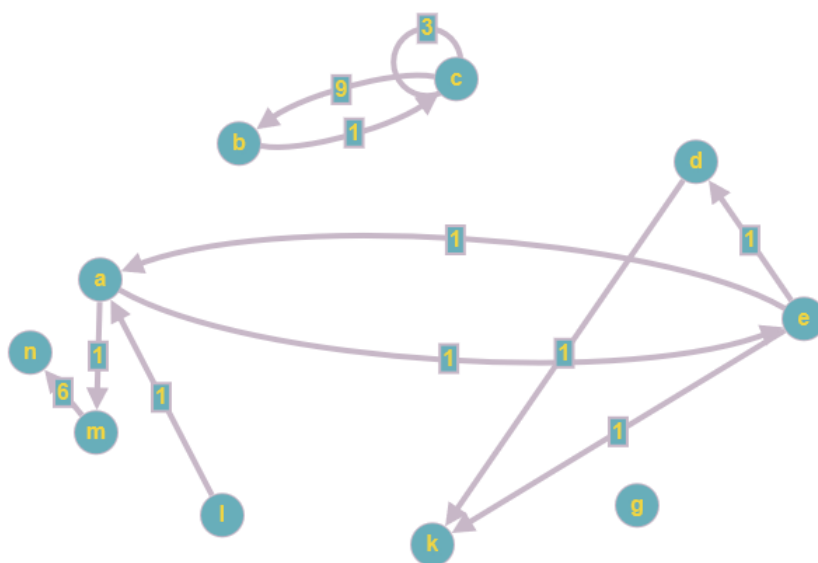


Рисунок 6 – Граф 2

3 Задание 3: Список смежности Ia

Задание: Вывести те вершины, полустепень исхода которых больше, чем у заданной вершины.

Сначала вводим заданную вершину, затем проходим по списку смежности (первый tar) и смотрим количество элементов в каждом внутреннем tar. Если их количество больше, то выводим вершину. При несуществующей вершине выводятся все вершины, т.е. изолированные вершины и вершины, отсутствующие в графе, по полустепени захода отождествляются.

```
Command: T3
Enter vertice: a
v
Command: T3
Enter vertice: k
a b d e l u v
Command: T3
Enter vertice: v
No such vertices!
Command: |
```

Рисунок 7 – Задание 3

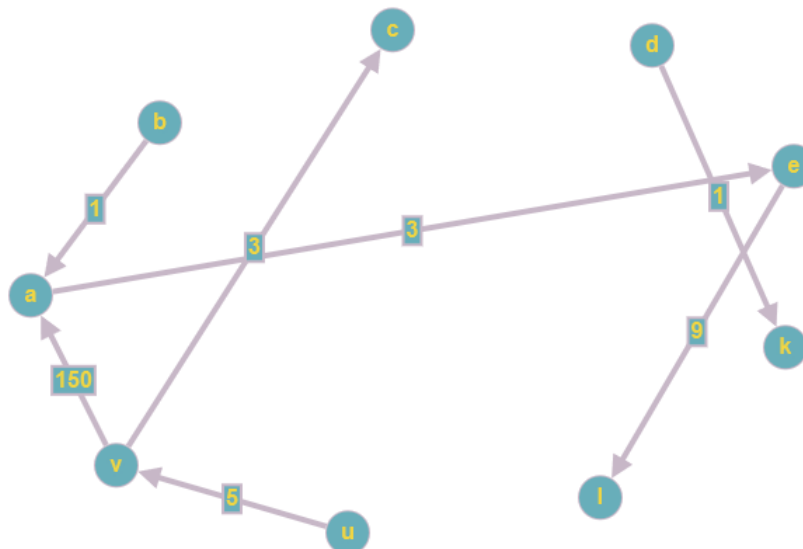


Рисунок 8 – Граф 3

4 Задание 4: Список смежности Ib: несколько графов

Задание: Построить орграф, являющийся симметрической разностью по дугам двух заданных орграфов (множество вершин получается объединением вершин исходных орграфов).

Если один граф неориентированный, а второй ориентированный (или наоборот), то алгоритм говорит, что графы несовместны.

Алгоритм просматривает списки смежности двух графов, добавляет в новый граф все вершины, которые он встретил (не более одного раза). Далее перебираем всевозможные пары вершин нового графа: если такая пара встретила лишь в одном списке смежности исходных двух графов (либо в первом, либо во втором), то добавляем эту пару (ребро) в новый граф, иначе игнорируем. В конце формируем граф на основе нового списка смежности.

```
Command: T4
a: (b, 1) (e, 1) (m, 1)
b: (a, 1)
c: (c, 3)
d: (k, 1)
e: (a, 1) (f, 5) (k, 1)
f: (e, -3) (n, 14)
g: (k, 33)
k:
l: (a, 1)
m:
n:
o:
Command: |
```

Рисунок 9 – Задание 4

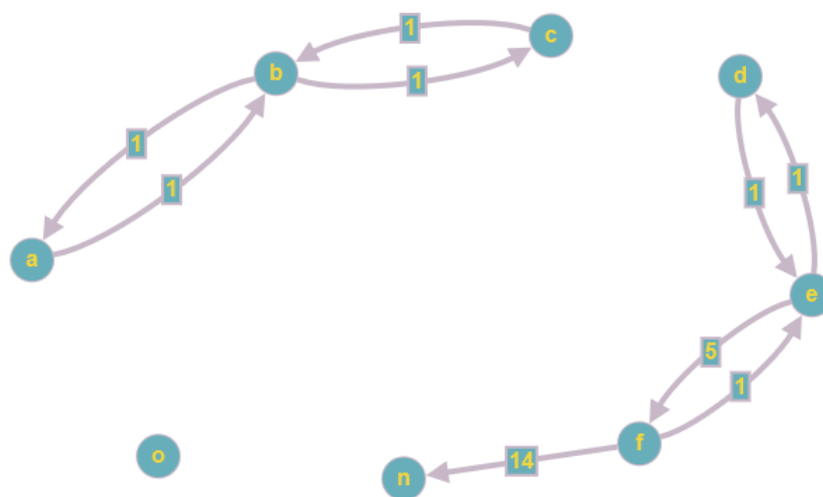


Рисунок 10 – Граф 4.1

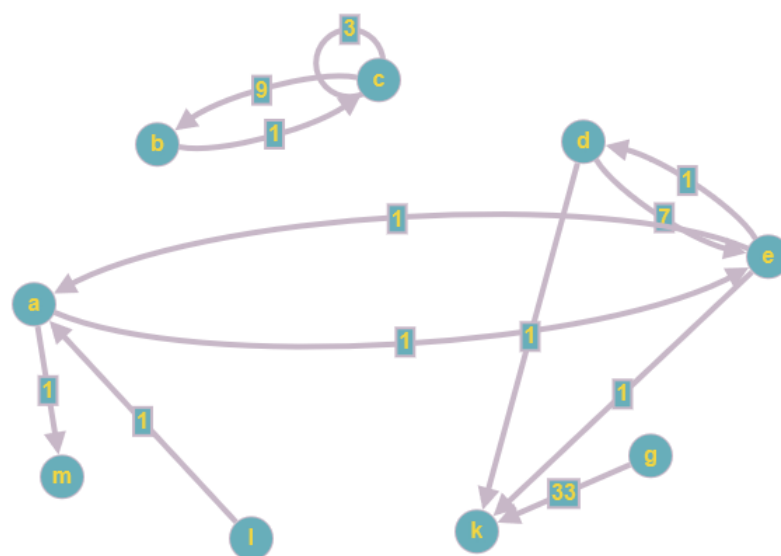


Рисунок 11 – Граф 4.2

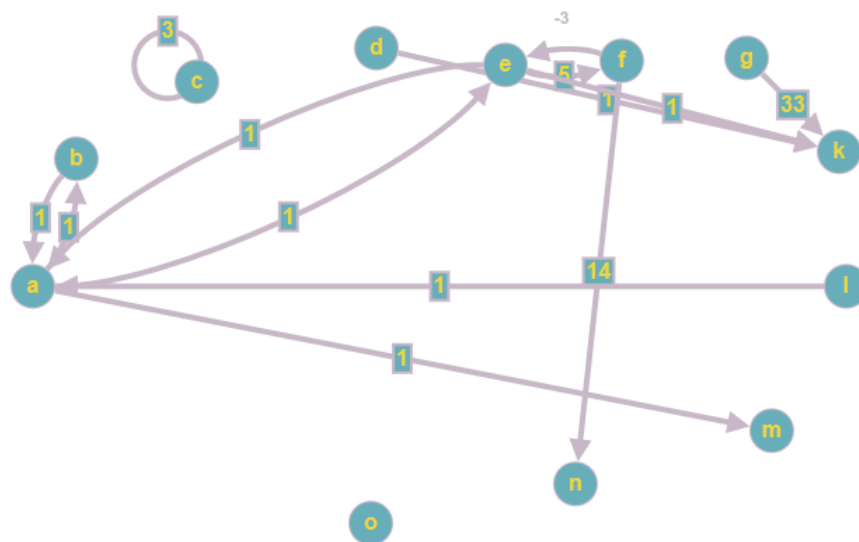


Рисунок 12 – Граф симметричной разности

5 Задание 5: Обходы графа II

Задание: Найти все вершины орграфа, недостижимые из данной.

Обычное использование BFS: выводим вершины, которые после прохода BFS остались непомеченными. Если все вершины достижимы, выводим об этом сообщение.

```
Command: T5
Enter target vertex: A
C E F K
Command: T5
Enter target vertex: C
K
Command: T5
Enter target vertex: L
A B C D E F G K
Command: T5
Enter target vertex: .
A B C D E F G K L
Command: |
```

Рисунок 13 – Задание 5

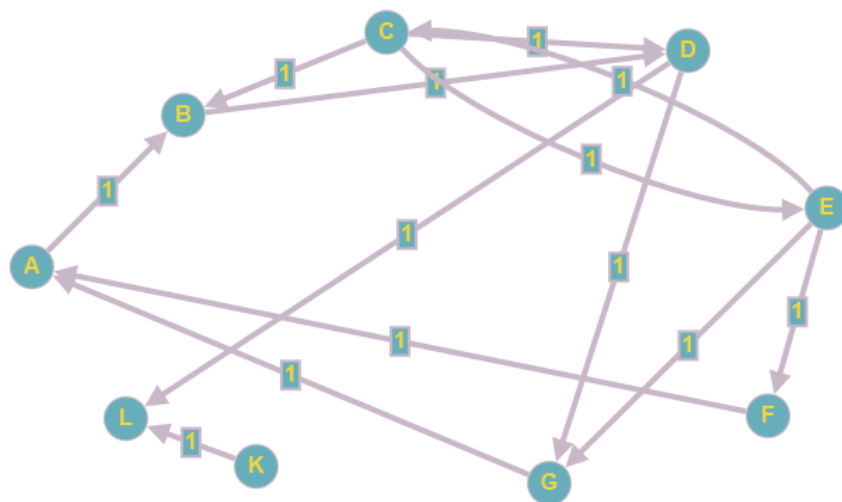


Рисунок 14 – Граф 5

6 Задание 6: Обходы графа II

Задание: Вывести все пути из u в v.

Используем модифицированный алгоритм DFS: запускаем обход из u в поиске v. По пути от u к v помечаем вершины пройденными. При достижении вершины v выводим путь, который мы запомнили (мы помещали его в vector). По пути назад помечаем вершины непройденными (т.к. одна вершина может участвовать в двух разных путях).

Всевозможные пути будут найдены, т.к. DFS переберёт всевозможные комбинации путей (поскольку при выборе очередной вершины для продолжения обхода он перебирает все смежные вершины).

```
Command: T6
Enter start vertice: A
Enter end vertice: Ff
No vertice 1 represented in graph
Command: T6
Enter start vertice: a
Enter end vertice: f
a->b->c->d->f
a->b->c->e->f
a->b->c->f
a->c->d->f
a->c->e->f
a->c->f
a->d->c->e->f
a->d->c->f
a->d->f
Command: T6
Enter start vertice: a
Enter end vertice: a
a
Command: T6
Enter start vertice: a
Enter end vertice: b
a->b
a->c->b
a->d->c->b
a->d->f->c->b
a->d->f->e->c->b
Command: |
```

Рисунок 15 – Задание 6

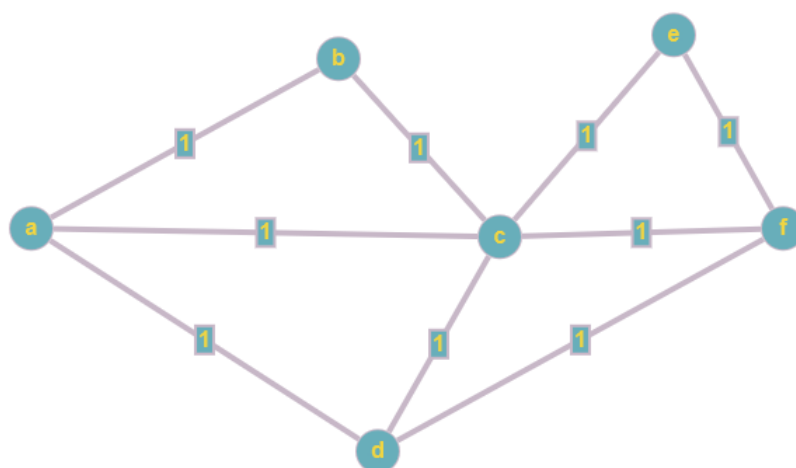


Рисунок 16 – Граф 6

7 Задание 7: Каркас III

Задание: найти во взвешенном неориентированном графе каркас минимального веса (алгоритмом Прима)

Обычная реализация алгоритма Прима: на каждом этапе будем жадно выбирать ребро с минимальным весом, расширяющее наше дерево.

Если граф ориентированный или несвязный, то возвращаем NULL. Иначе возвращаем указатель на новосозданный каркас.

```
Command: 1
a: (b, 2) (c, 4) (d, 1)
b: (a, 2) (c, 1)
c: (a, 4) (b, 1) (d, 1) (e, 1) (f, 1)
d: (a, 1) (c, 1) (f, 1)
e: (c, 1) (f, 1)
f: (c, 1) (d, 1) (e, 1)
Command: T7
a: (d, 1)
b: (c, 1)
c: (b, 1) (d, 1) (e, 1)
d: (a, 1) (c, 1) (f, 1)
e: (c, 1)
f: (d, 1)
Command: 6
Enter start vertice: a
Enter end vertice: c
Enter edge weight (default = 1):
Weight changed successfully
Command: T7
a: (c, 1) (d, 1)
b: (c, 1)
c: (a, 1) (b, 1) (e, 1) (f, 1)
d: (a, 1)
e: (c, 1)
f: (c, 1)
Command: |
```

Рисунок 17 – Задание 7

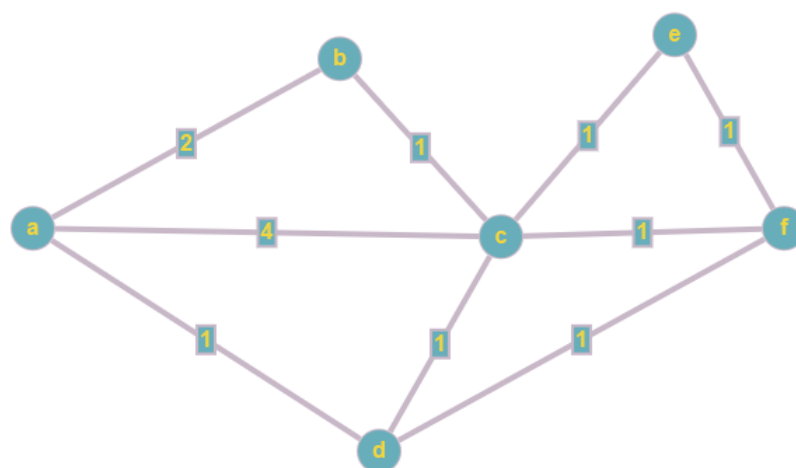


Рисунок 18 – Граф 7.1

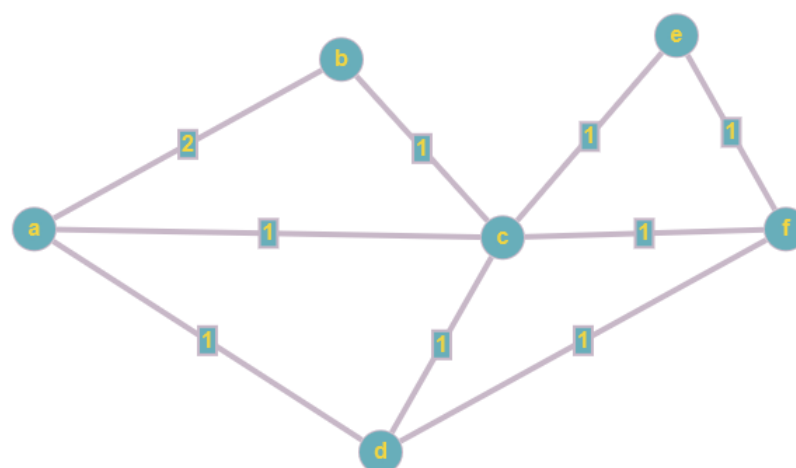


Рисунок 19 – Граф 7.2

8 Задание 8: Веса IVc

Задание: Эксцентриситет вершины — максимальное расстояние из всех минимальных расстояний от других вершин до данной вершины. Радиус графа — минимальный из эксцентриситетов его вершин. Найти центр графа — множество вершин, эксцентриситеты которых равны радиусу графа. В графе нет рёбер отрицательного веса.

Используем обычный алгоритм Дейкстры для обхода графа и нахождения всевозможных расстояний от одной вершины до другой. Затем считаем максимум среди этих расстояний (эксцентриситеты), а затем минимумы из этих максимумов (минимум эксцентриситетов – радиус графа). Затем соотносим найденный радиус с минимальным эксцентриситетом каждой вершины: если они совпадают, то эта вершина входит в центр графа.

Если граф несвязный, то его радиус – $+\infty$. В таком случае его нет, сообщаем об этом. В ином случае выводим множество вершин.

```
Command: 1
A: (B, 7) (D, 5)
B: (A, 7) (C, 8) (D, 9) (E, 7)
C: (B, 8) (E, 5)
D: (A, 5) (B, 9) (E, 15) (F, 6)
E: (B, 7) (C, 5) (D, 15) (F, 8) (G, 9)
F: (D, 6) (E, 8) (G, 11)
G: (E, 9) (F, 11)
Command: T8
E
Command: |
```

Рисунок 20 – Задание 8

```
Command: 1
A: (B, 1) (D, 1)
B: (A, 1) (C, 1)
C: (B, 1) (D, 1)
D: (A, 1) (C, 1)
K: (L, 7)
L: (K, 7)
Command: T8
No center vertex in graph (graph is not connected)
Command: |
```

Рисунок 21 – Задание 8

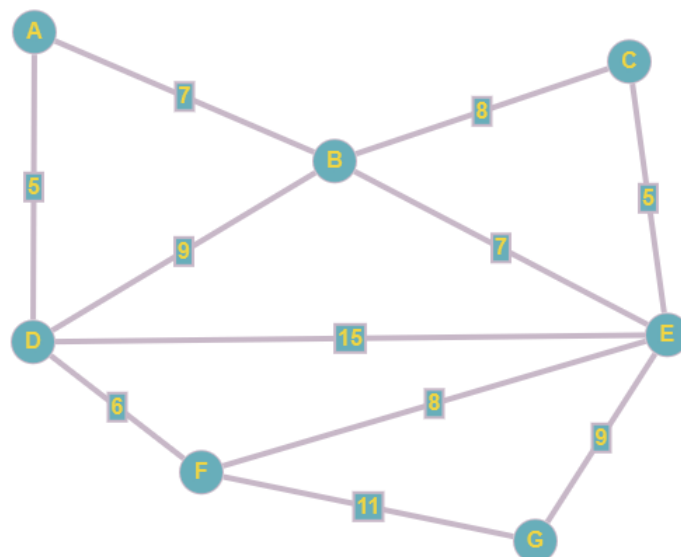


Рисунок 22 – Граф 8.2

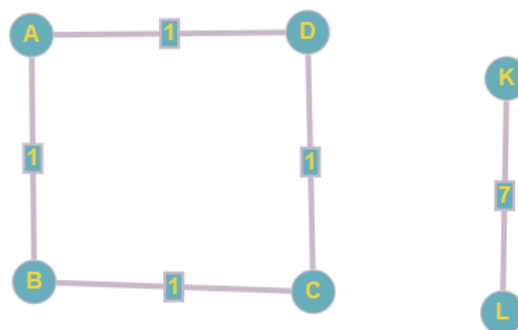


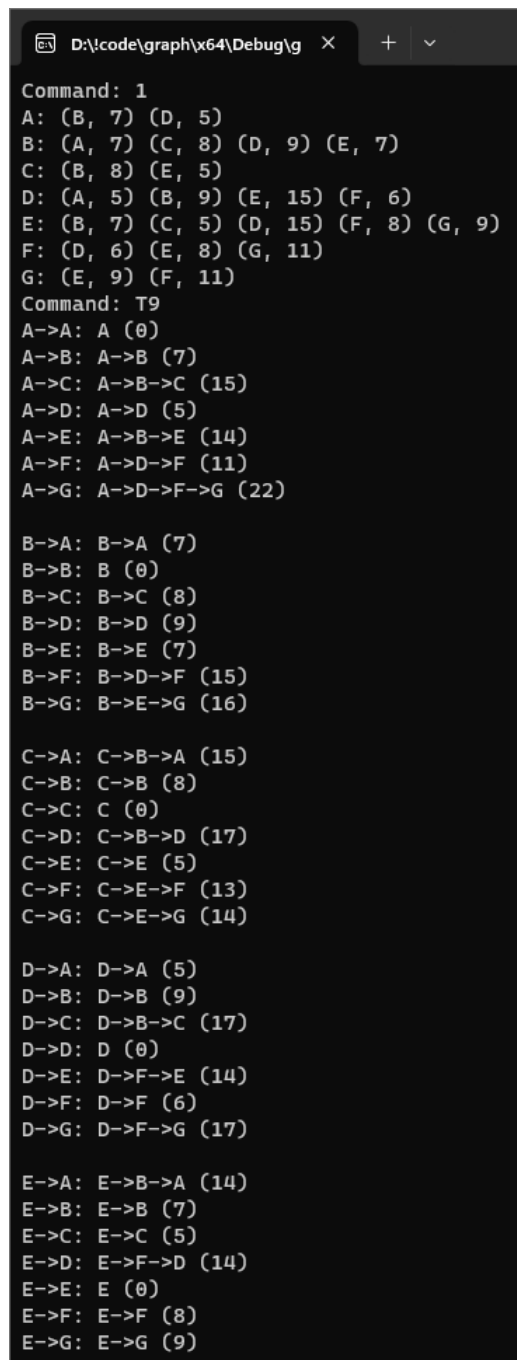
Рисунок 23 – Граф 8.1

9 Задание 9: Веса IVc

Задание: Вывести кратчайшие пути для всех пар вершин. В графе нет циклов отрицательного веса.

Воспользуемся алгоритмом Флойда, который также запоминает путь, по которому был получен данный кратчайший путь. Затем просто выводим эти кратчайшие пути с минимальным найденным расстоянием.

Если найден отрицательный цикл, то алгоритм останавливается и выводит сообщение об ошибке.



```
Command: 1
A: (B, 7) (D, 5)
B: (A, 7) (C, 8) (D, 9) (E, 7)
C: (B, 8) (E, 5)
D: (A, 5) (B, 9) (E, 15) (F, 6)
E: (B, 7) (C, 5) (D, 15) (F, 8) (G, 9)
F: (D, 6) (E, 8) (G, 11)
G: (E, 9) (F, 11)
Command: T9
A->A: A (0)
A->B: A->B (7)
A->C: A->B->C (15)
A->D: A->D (5)
A->E: A->B->E (14)
A->F: A->D->F (11)
A->G: A->D->F->G (22)

B->A: B->A (7)
B->B: B (0)
B->C: B->C (8)
B->D: B->D (9)
B->E: B->E (7)
B->F: B->D->F (15)
B->G: B->E->G (16)

C->A: C->B->A (15)
C->B: C->B (8)
C->C: C (0)
C->D: C->B->D (17)
C->E: C->E (5)
C->F: C->E->F (13)
C->G: C->E->G (14)

D->A: D->A (5)
D->B: D->B (9)
D->C: D->B->C (17)
D->D: D (0)
D->E: D->F->E (14)
D->F: D->F (6)
D->G: D->F->G (17)

E->A: E->B->A (14)
E->B: E->B (7)
E->C: E->C (5)
E->D: E->F->D (14)
E->E: E (0)
E->F: E->F (8)
E->G: E->G (9)
```

Рисунок 24 – Задание 9

```

F->A: F->D->A (11)
F->B: F->D->B (15)
F->C: F->E->C (13)
F->D: F->D (6)
F->E: F->E (8)
F->F: F (0)
F->G: F->G (11)

```

```

G->A: G->F->D->A (22)
G->B: G->E->B (16)
G->C: G->E->C (14)
G->D: G->F->D (17)
G->E: G->E (9)
G->F: G->F (11)
G->G: G (0)

```

Command: |

Рисунок 25 – Задание 9

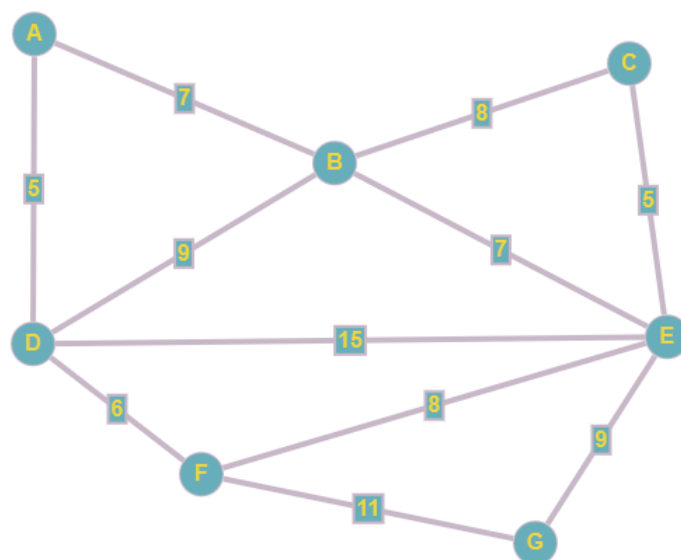


Рисунок 26 – Граф 9.1

```
D:\code\graph\x64\Debug\g  X + v
Command: 1
A: (B, 1) (D, 1)
B: (A, 1) (C, 1)
C: (B, 1) (D, 1)
D: (A, 1) (C, 1)
K: (L, 7)
L: (K, 7)
Command: T9
A->A: A (0)
A->B: A->B (1)
A->C: A->B->C (2)
A->D: A->D (1)
A->K: No path between these vertices
A->L: No path between these vertices

B->A: B->A (1)
B->B: B (0)
B->C: B->C (1)
B->D: B->A->D (2)
B->K: No path between these vertices
B->L: No path between these vertices

C->A: C->B->A (2)
C->B: C->B (1)
C->C: C (0)
C->D: C->D (1)
C->K: No path between these vertices
C->L: No path between these vertices

D->A: D->A (1)
D->B: D->A->B (2)
D->C: D->C (1)
D->D: D (0)
D->K: No path between these vertices
D->L: No path between these vertices

K->A: No path between these vertices
K->B: No path between these vertices
K->C: No path between these vertices
K->D: No path between these vertices
K->K: K (0)
K->L: K->L (7)

L->A: No path between these vertices
L->B: No path between these vertices
L->C: No path between these vertices
L->D: No path between these vertices
L->K: L->K (7)
L->L: L (0)

Command: |
```

Рисунок 27 – Задание 9

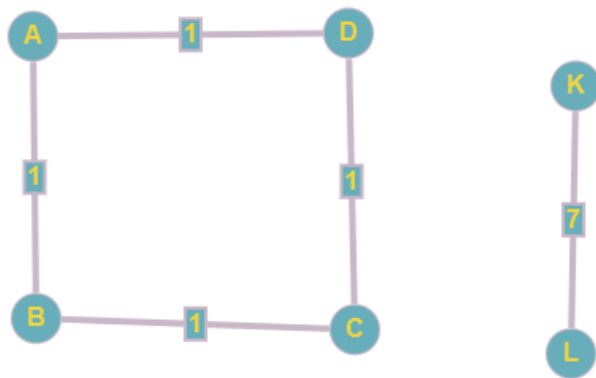


Рисунок 28 – Граф 9.2

```

Command: 1
A: (B, -5)
B: (C, 4)
C: (A, 2) (D, -4)
D: (A, 5)
Command: T9
A->A: A (0)
A->B: A->B (-5)
A->C: A->B->C (-1)
A->D: A->B->C->D (-5)

B->A: B->C->D->A (5)
B->B: B (0)
B->C: B->C (4)
B->D: B->C->D (0)

C->A: C->D->A (1)
C->B: C->D->A->B (-4)
C->C: C (0)
C->D: C->D (-4)

D->A: D->A (5)
D->B: D->A->B (0)
D->C: D->A->B->C (4)
D->D: D (0)

Command: |
  
```

Рисунок 29 – Задание 9

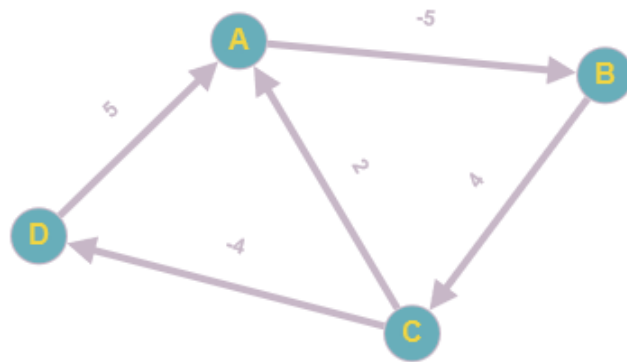


Рисунок 30 – Граф 9.3

10 Задание 10: Веса IVc

Задание: Определить, существует ли путь длиной не более L между двумя заданными вершинами графа.

Воспользуемся алгоритмом Форда-Беллмана. При нахождении отрицательного цикла останавливаем алгоритм и выводим сообщение об ошибке. Затем соотносим кратчайший путь с величиной L : если кратчайший путь меньше, то путь существует (предъявляем его), иначе выводим сообщение о том, что он не существует.

```
Command: 1
A: (B, 7) (D, 5)
B: (A, 7) (C, 8) (D, 9) (E, 7)
C: (B, 8) (E, 5)
D: (A, 5) (B, 9) (E, 15) (F, 6)
E: (B, 7) (C, 5) (D, 15) (F, 8) (G, 9)
F: (D, 6) (E, 8) (G, 11)
G: (E, 9) (F, 11)
Command: T10
Enter start vertex: A
Enter end vertex: C
Enter maximum path length: 5
Path shorter than 5 between A and C does not exist.
Command: T10
Enter start vertex: A
Enter end vertex: C
Enter maximum path length: 16
Path shorter than 16 between A and C exists:
A->B->C (15)
Command: T10
Enter start vertex: A
Enter end vertex: R
No such vertex in graph!
Command: |
```

Рисунок 31 – Задание 10

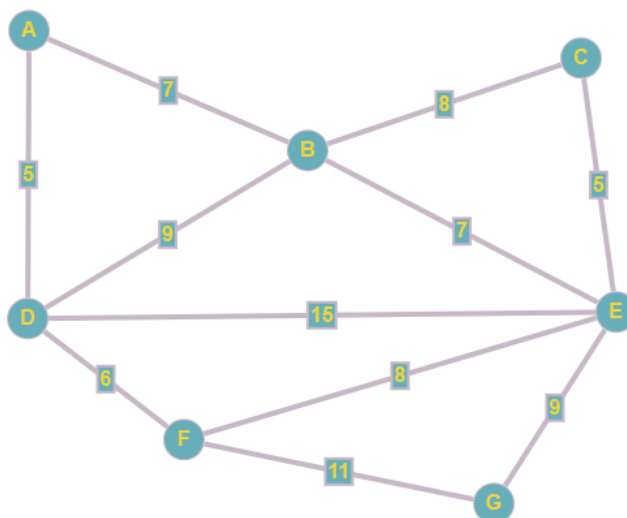


Рисунок 32 – Граф 10.1

```

Command: 1
A: (B, 1) (D, 1)
B: (A, 1) (C, 1)
C: (B, 1) (D, 1)
D: (A, 1) (C, 1)
K: (L, 7)
L: (K, 7)
Command: T10
Enter start vertice: A
Enter end vertice: D
Enter maximum path length: -2
Path shorter than -2 between A and D does not exist.
Command: T10
Enter start vertice: A
Enter end vertice: D
Enter maximum path length: 10
Path shorter than 10 between A and D exists:
A->D (1)
Command: T10
Enter start vertice: A
Enter end vertice: K
Enter maximum path length: 100
Path shorter than 100 between A and K does not exist.
Command: |

```

Рисунок 33 – Задание 10

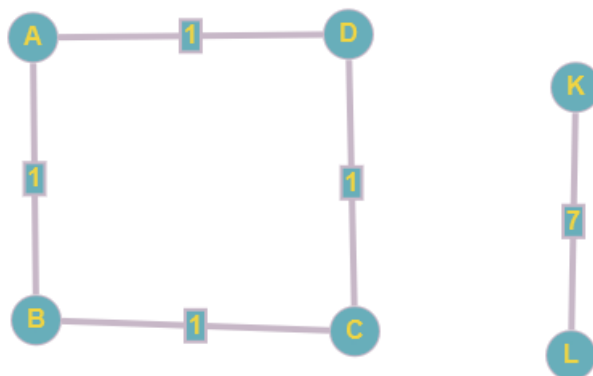


Рисунок 34 – Граф 10.2

```

Command: 1
A: (B, -5)
B: (C, 4)
C: (A, 2) (D, -4)
D: (A, 5)
Command: T10
Enter start vertex: A
Enter end vertex: C
Enter maximum path length: 2
Path shorter than 2 between A and C exists:
A->B->C (-1)
Command: T10
Enter start vertex: A
Enter end vertex: A
Enter maximum path length: -1
Path shorter than -1 between A and A does not exist.
Command: T10
Enter start vertex: A
Enter end vertex: A
Enter maximum path length: 1
Path shorter than 1 between A and A exists:
A (0)
Command: |

```

Рисунок 35 – Задание 10

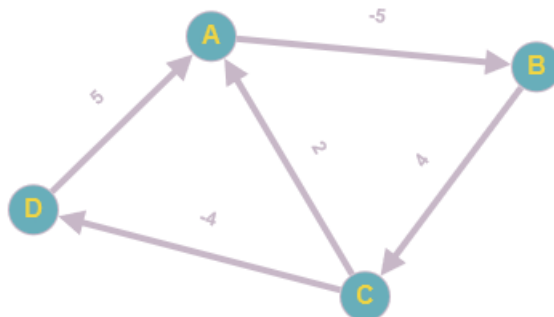


Рисунок 36 – Граф 10.3

```

Command: 6
Enter start vertice: B
Enter end vertice: C
Enter edge weight (default = 1): 2
Weight changed successfully
Command: 1
A: (B, -5)
B: (C, 2)
C: (A, 2) (D, -4)
D: (A, 5)
Command: T10
Enter start vertice: A
Enter end vertice: D
Enter maximum path length: 0
Negative cycle found, algorithm terminated
Command: |

```

Рисунок 37 – Задание 10

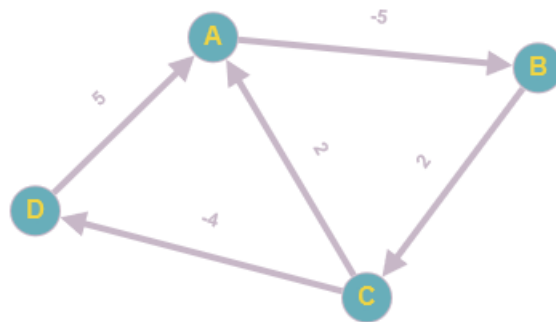


Рисунок 38 – Граф 10.4

11 Задание 11: Максимальный поток

Задание: Решить задачу на нахождение максимального потока любым алгоритмом.

Простейшая реализация алгоритма Форда-Фалкерсона с корректировкой остаточной сети после каждого нахождения пути. Для нахождения пути используется DFS.

```
Command: 1
A: (B, 17) (C, 9)
B: (C, 5) (E, 10)
C: (D, 13)
D: (F, 2) (G, 8)
E: (F, 6) (G, 7)
F: (G, 15)
G:
Command: T11
Enter source vertex: A
Enter target vertex: G
Maximum flow is 20.
Command: T11
Enter source vertex: A
Enter target vertex: C
Maximum flow is 14.
Command: T11
Enter source vertex: A
Enter target vertex: E
Maximum flow is 10.
Command: T11
Enter source vertex: A
Enter target vertex: R
No vertex 2 in graph
Command: |
```

Рисунок 39 – Задание 11

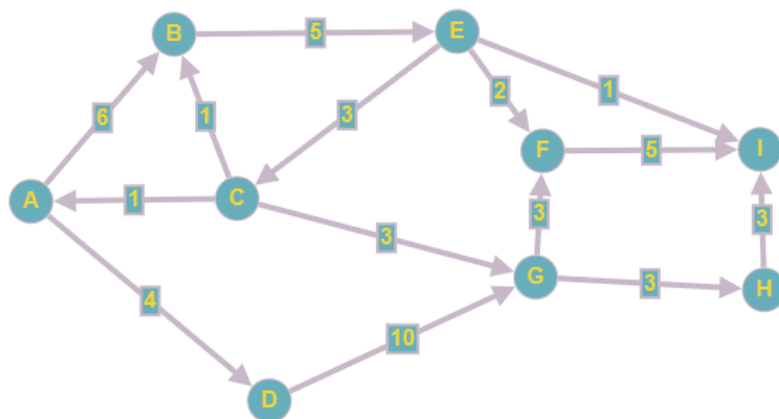


Рисунок 40 – Граф 11.1

```

Command: T11
Enter source vertex: A
Enter target vertex: I
Maximum flow is 8.
Command: 6
Enter start vertex: H
Enter end vertex: I
Enter edge weight (default = 1): 0
Weight changed successfully
Command: T11
Enter source vertex: A
Enter target vertex: I
Maximum flow is 5.
Command: T11
Enter source vertex: A
Enter target vertex: A
Infinite flow (source and targets are equal)
Command: |

```

Рисунок 41 – Задание 11

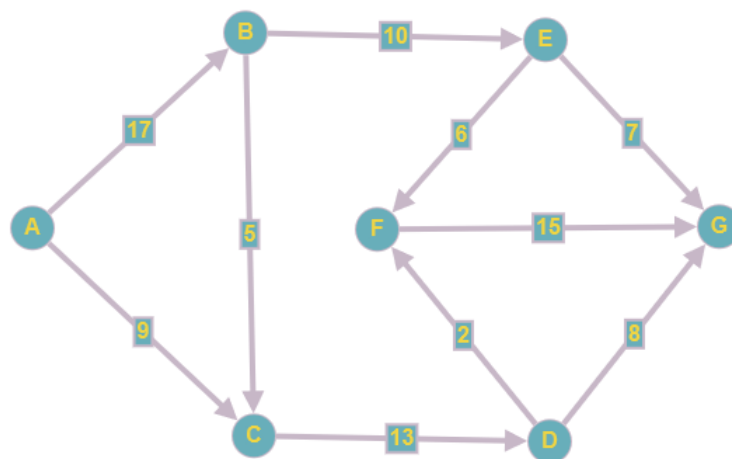


Рисунок 42 – Граф 11.2

ПРИЛОЖЕНИЕ А

Файлы класса графов

Graph.h файл:

```
1  #pragma once
2
3  #include <map>
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  #include <nlohmann/json.hpp>
8
9  using json = nlohmann::json;
10
11 using std::string;
12 using std::map;
13 using std::vector;
14 using std::pair;
15 using std::ofstream;
16 using std::ifstream;
17
18 class Graph
19 {
20     friend void to_json(json& j, const Graph& graph);
21 public:
22
23     enum graph_orientation
24     {
25         undirected = 0,
26         directed = 1
27     };
28
29     enum code_error
30     {
31         no_error = 0,
```

```

32
33         vertice_exists,
34         no_vertice1,
35         no_vertice2,
36         edge_exists,
37         no_edge
38     };
39
40     Graph(bool orient = true);
41     Graph(istream& file);
42     Graph(const Graph& copiedValue);
43     Graph(map<string, map<string, int32_t>>, bool isOriented);
44
45     const map<string, map<string, int32_t>> GetAdjacencyList()
↪     const;
46     bool GetOrientation();
47     void ChangeOrientation();
48
49     bool isVertice(string s);
50     bool isEdge(string s1, string s2);
51
52     static graph_orientation Hashing(string const& inString);
53
54     uint8_t AddVertice(const string& value);
55     uint8_t AddEdge(const string& startVertice, const string&
↪     endVertice, const int32_t& weight);
56
57     uint8_t RemoveVertice(const string& vertice);
58     uint8_t RemoveEdge(const string& startVertice, const
↪     string& endVertice);
59
60     uint8_t ChangeWeight(const string& startVertice, const
↪     string& endVertice, const int32_t& weight);
61     void Unweight();

```



```

62
63         void Save(string fileName);
64 private:
65         map<string, map<string, int32_t>> adjacencyList;
66         bool isOriented;
67 };

```

Graph.cpp файл:

```

1  #include "Graph.h"
2
3  Graph::Graph(bool isOriented)
4  {
5      this->isOriented = isOriented;
6  }
7
8  Graph::Graph(istream& file)
9  {
10     json j;
11     file >> j;
12     this->isOriented = j["orient"];
13     for (auto& adjList : j["vertices"].items())
14     {
15         this->adjacencyList[adjList.key()] =
↪ j["vertices"][adjList.key()].get<map<string, int32_t>>();
16     }
17 }
18
19 Graph::Graph(const Graph& copiedValue)
20 {
21     this->isOriented = copiedValue.isOriented;
22     this->adjacencyList = copiedValue.adjacencyList;
23 }
24
25 Graph::Graph(map <string, map<string, int32_t>> list, bool
↪ isOriented)

```

```

26 {
27     this->adjacencyList = list;
28     this->isOriented = isOriented;
29 }
30
31 //getters
32 const map<string, map<string, int32_t>> Graph::GetAdjacencyList()
    ↪ const
33 {
34     return adjacencyList;
35 }
36
37 bool Graph::GetOrientation()
38 {
39     return isOriented;
40 }
41
42 void Graph::ChangeOrientation()
43 {
44     isOriented = !isOriented;
45 }
46
47 bool Graph::isVertice(string s)
48 {
49     return !(adjacencyList.find(s) == adjacencyList.end());
50 }
51
52 bool Graph::isEdge(string s1, string s2)
53 {
54     return adjacencyList[s1].find(s2) !=
    ↪ adjacencyList[s1].end();
55 }
56
57

```

```

58 Graph::graph_orientation Graph::Hashing(std::string const&
    ↪ inString)
59 {
60     if (inString == "0")
61         return graph_orientation::undirected;
62     else
63         return graph_orientation::directed;
64 }
65
66 //methods
67 uint8_t Graph::AddVertice(const string& vertice)
68 {
69     if (!isVertice(vertice))
70     {
71         adjacencyList[vertice];
72         return code_error::no_error;
73     }
74     else
75         return code_error::vertice_exists;
76 }
77
78 uint8_t Graph::ChangeWeight(const string& startVertice, const
    ↪ string& endVertice, const int32_t& weight)
79 {
80     //fail: no first vertice
81     if (!isVertice(startVertice))
82         return code_error::no_vertice1;
83
84     //fail: no second vertice
85     if (!isVertice(endVertice))
86         return code_error::no_vertice2;
87
88     //fail: no such edge
89     if (!isEdge(startVertice, endVertice))

```

```

90         return code_error::no_edge;
91
92         //success
93         adjacencyList[startVertice][endVertice] = weight;
94         if (!isOriented)
95             adjacencyList[endVertice][startVertice] = weight;
96         return code_error::no_error;
97     }
98
99     uint8_t Graph::AddEdge(const string& startVertice, const string&
    ↪ endVertice, const int32_t& weight)
100 {
101     //fail: no first vertice
102     if (!isVertice(startVertice))
103         return code_error::no_vertice1;
104
105     //fail: no second vertice
106     if (!isVertice(endVertice))
107         return code_error::no_vertice2;
108
109     //fail: edge already exists
110     if (isEdge(startVertice, endVertice))
111         return code_error::edge_exists;
112
113     //success
114     adjacencyList[startVertice][endVertice] = weight;
115     if (!isOriented)
116         adjacencyList[endVertice][startVertice] = weight;
117     return code_error::no_error;
118 }
119
120     uint8_t Graph::RemoveVertice(const string& removedVertice)
121 {
122     //fail: no such vertice

```

```

123         if (!isVertice(removedVertice))
124             return code_error::no_vertice1;
125
126         //success, cleaning other vertices and their edges
127         for (auto vert : adjacencyList)
128         {
129             this->RemoveEdge(vert.first, removedVertice);
130             this->RemoveEdge(removedVertice, vert.first);
131         }
132         adjacencyList.erase(removedVertice);
133         return code_error::no_error;
134     }
135
136     uint8_t Graph::RemoveEdge(const string& startVertice, const
        ↪ string& endVertice)
137     {
138         if (adjacencyList[startVertice].find(endVertice) !=
        ↪ adjacencyList[startVertice].end())
139         {
140             adjacencyList[startVertice].erase(endVertice);
141             if (!isOriented)
142             {
143                 adjacencyList[endVertice].erase(startVertice);
144             }
145             //success
146             return no_error;
147         }
148         else
149             return code_error::no_edge;
150     }
151
152     void Graph::Unweight()
153     {

```

```

154         for (auto vert1 = adjacencyList.begin(); vert1 !=
↪ adjacencyList.end(); vert1++)
155         {
156             for (auto vert2 = adjacencyList.begin(); vert2 !=
↪ adjacencyList.end(); vert2++)
157             {
158                 if (isEdge(vert1->first, vert2->first))
159                     adjacencyList
↪ [vert1->first][vert2->first] = 1;
160             }
161         }
162     }
163
164 void Graph::Save(string fileName)
165 {
166     json j = *this;
167     std::ofstream data(fileName);
168     data << std::setw(4) << j;
169 }
170
171 void to_json(json& j, const Graph& graph)
172 {
173     j["orient"] = graph.isOriented;
174     for (auto const& mapEl : graph.adjacencyList)
175     {
176         j["vertices"][mapEl.first] = mapEl.second;
177     }
178 }

```

ПРИЛОЖЕНИЕ Б

Файлы класса приложения

App.h

```
1  #pragma once
2
3  #include <string>
4  #include <iostream>
5
6  #include "Graph.h"
7
8  enum string_code
9  {
10      printVertices = 1,
11      addVertice,
12      removeVertice,
13      addEdge,
14      removeEdge,
15      changeWeight,
16      saveGraph,
17      unweightGraph,
18
19      task2 = 10,
20      task3,
21      task4,
22      task5,
23      task6,
24      task7,
25      task8,
26      task9,
27      task10,
28      task11,
29
30      help = 30,
31      quit
```

```

32 };
33
34 string_code Hashing(std::string const& inString);
35 void CommandMessage();
36 Graph* CreateGraph(string& command);
37
38 bool is_number(const string& s);
39 void PrintVertices(Graph* graph);
40 void AddVertice(Graph* graph);
41 void RemoveVertice(Graph* graph);
42 void AddEdge(Graph* graph);
43 void RemoveEdge(Graph* graph);
44 void ChangeWeight(Graph* graph);
45 void Unweight(Graph* graph);

```

App.cpp

```

1  #include "App.h"
2
3  using std::string;
4  using std::getline;
5  using std::cin;
6  using std::cout;
7
8  string_code Hashing(std::string const& inString) {
9      if (inString == "1") return printVertices;
10     if (inString == "2") return addVertice;
11     if (inString == "3") return removeVertice;
12     if (inString == "4") return addEdge;
13     if (inString == "5") return removeEdge;
14     if (inString == "6") return changeWeight;
15     if (inString == "7") return saveGraph;
16     if (inString == "8") return unweightGraph;
17
18     if (inString == "T2") return task2;

```



```

19         if (inString == "T3") return task3;
20         if (inString == "T4") return task4;
21         if (inString == "T5") return task5;
22         if (inString == "T6") return task6;
23         if (inString == "T7") return task7;
24         if (inString == "T8") return task8;
25         if (inString == "T9") return task9;
26         if (inString == "T10") return task10;
27         if (inString == "T11") return task11;
28
29         if (inString == "h") return help;
30         if (inString == "q") return quit;
31     }
32
33 void CommandMessage()
34 {
35     std::cout << "Select command: \n "
36               << "1 - Print Vertices \n "
37               << "2 - Add vertice \n "
38               << "3 - Remove vertice \n "
39               << "4 - Add edge \n "
40               << "5 - Remove edge \n "
41               << "6 - Change edge's weight \n "
42               << "7 - Save graph \n "
43               << "8 - Unweight graph \n "
44               << '\n'
45               << "T2 - task 2 \n "
46               << "T3 - task 3 \n "
47               << "T4 - task 4 \n "
48               << "T5 - task 5 \n "
49               << "T6 - task 6 \n "
50               << "T7 - task 7 \n "
51               << "T8 - task 8 \n "
52               << "T9 - task 9 \n "

```

```

53         << "T10 - task 10\n"
54         << "T11 - task 11\n"
55         << '\n'
56         << "'h' to print this message\n"
57         << "'q' to exit program\n";
58     }
59
60     Graph* CreateGraph(string& command)
61     {
62         while (true)
63         {
64             std::cout << "No graph found. Choose directed or
↪ undirected graph\n"
65                 << "1 - Directed graph\n"
66                 << "2 - Undirected graph\n";
67             getline(std::cin, command);
68             switch (Graph::Hashing(command))
69             {
70                 case Graph::undirected:
71                     std::cout << "Created a new undirected
↪ graph\n";
72                     return new Graph(false);
73                 case Graph::directed:
74                     std::cout << "Created a new directed
↪ graph\n";
75                     return new Graph(true);
76                 default:
77                     break;
78             }
79         }
80     }
81
82     //Hidden
83     bool is_number(const std::string& s)

```

```

84 {
85     std::string::const_iterator it = s.begin();
86     while (it != s.end() && (std::isdigit(*it) || (*it) ==
    ↪ '-')) ++it;
87     return !s.empty() && it == s.end();
88 }
89
90 void PrintVertices(Graph* graph)
91 {
92     auto adjacencyList = graph->GetAdjacencyList();
93
94     for (auto& list : adjacencyList)
95     {
96         std::cout << list.first << ": ";
97         for (auto& el : list.second)
98         {
99             std::cout << "(" << el.first << ", " <<
    ↪ el.second << ") ";
100         }
101         std::cout << '\n';
102     }
103 }
104
105 void AddVertice(Graph* graph)
106 {
107     string vertice;
108     std::cout << "Enter vertice to add: ";
109     getline(cin, vertice);
110     switch (graph->AddVertice(vertice))
111     {
112     case Graph::code_error::no_error:
113         std::cout << "Vertice added succesfully\n";
114         break;
115     case Graph::code_error::vertice_exists:

```

```

116             std::cout << "Vertice already exists\n";
117             break;
118         }
119     }
120
121     void RemoveVertice(Graph* graph)
122     {
123         string vertice;
124         std::cout << "Enter vertice to remove: ";
125         getline(cin, vertice);
126         switch (graph->RemoveVertice(vertice))
127         {
128             case Graph::code_error::no_error:
129                 std::cout << "Vertice removed successfully\n";
130                 break;
131             case Graph::code_error::no_vertice1:
132                 std::cout << "No such vertice in graph\n";
133                 break;
134         }
135     }
136
137     void AddEdge(Graph* graph)
138     {
139         string vertice1;
140         string vertice2;
141         string weightMsg;
142         std::cout << "Enter start vertice: ";
143         getline(cin, vertice1);
144         std::cout << "Enter end vertice: ";
145         getline(cin, vertice2);
146         std::cout << "Enter edge weight (default = 1): ";
147         getline(cin, weightMsg);
148         if (weightMsg.empty())
149             weightMsg = "1";

```

```

150         while (!is_number(weightMsg))
151         {
152             std::cout << "Wrong weight value! Enter integer:
↪ ";
153             getline(cin, weightMsg);
154             if (weightMsg.empty())
155                 weightMsg = "1";
156         }
157
158         switch (graph->AddEdge(vertice1, vertice2,
↪ std::stoi(weightMsg)))
159         {
160             case Graph::no_vertice1:
161                 std::cout << "No vertice 1 represented in
↪ graph \n ";
162                 break;
163             case Graph::no_vertice2:
164                 std::cout << "No vertice 2 represented in
↪ graph \n ";
165                 break;
166             case Graph::edge_exists:
167                 std::cout << "Edge already exists between these 2
↪ vertices \n ";
168                 break;
169             case Graph::no_error:
170                 std::cout << "Edge added successfully \n ";
171                 break;
172         }
173     }
174
175 void RemoveEdge(Graph* graph)
176 {
177     string vertice1;
178     string vertice2;

```

```

179         std::cout << "Enter start vertice: ";
180         getline(cin, vertice1);
181         std::cout << "Enter end vertice: ";
182         getline(cin, vertice2);
183
184         switch (graph->RemoveEdge(vertice1, vertice2))
185         {
186             case Graph::code_error::no_edge:
187                 std::cout << "No such edge in graph\n";
188                 break;
189             case Graph::code_error::no_error:
190                 std::cout << "Edge removed successfully\n";
191                 break;
192         }
193     }
194
195     void ChangeWeight(Graph* graph)
196     {
197         string vertice1;
198         string vertice2;
199         string weightMsg;
200         std::cout << "Enter start vertice: ";
201         getline(cin, vertice1);
202         std::cout << "Enter end vertice: ";
203         getline(cin, vertice2);
204         std::cout << "Enter edge weight (default = 1): ";
205         getline(cin, weightMsg);
206         if (weightMsg.empty())
207             weightMsg = "1";
208         while (!is_number(weightMsg))
209         {
210             std::cout << "Wrong weight value! Enter integer:
↳ ";
211             getline(cin, weightMsg);

```

```

212     }
213
214     switch (graph->ChangeWeight(vertice1, vertice2,
↪ std::stoi(weightMsg)))
215     {
216         case Graph::no_vertice1:
217             std::cout << "No vertice 1 represented in
↪ graph \n ";
218             break;
219         case Graph::no_vertice2:
220             std::cout << "No vertice 2 represented in
↪ graph \n ";
221             break;
222         case Graph::no_edge:
223             std::cout << "No edge exists between these 2
↪ vertices \n ";
224             break;
225         case Graph::no_error:
226             std::cout << "Weight changed successfully \n ";
227             break;
228     }
229 }
230
231 void Unweight(Graph* graph)
232 {
233     graph->Unweight();
234     std::cout << "Weight of all edges changed to 1 \n ";
235 }

```

ПРИЛОЖЕНИЕ В

Файлы Tasks (заданий)

Tasks.h

```
1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <queue>
6  #include <string>
7  #include <iostream>
8  #include <set>
9  #include <stack>
10 #include <algorithm>
11
12 #include "Graph.h"
13
14 void task2_14(Graph* graph);
15 void task3_9(Graph* graph);
16 void task4_10(Graph* graph1, Graph* graph2);
17 void task5_2(Graph* graph);
18 void task6_20(Graph* graph);
19 void task7_prim(Graph* graph);
20 void task8_11(Graph* graph);
21 void task9_17(Graph* graph);
22 void task10_1(Graph* graph);
23 void task11_net(Graph* graph);
```

Tasks.cpp

```
1  #include "Tasks.h"
2  #include "App.h"
3  #include "algos.h"
4
5  using std::string;
```



```

6  using std::set;
7  using std::getline;
8  using std::cin;
9  using std::cout;
10
11  //just prints adjacent vertices
12  void task2_14(Graph* graph)
13  {
14      string vertice;
15      std::cout << "Enter vertice to print: ";
16      getline(cin, vertice);
17
18      auto list = graph->GetAdjacencyList();
19      auto map = list[vertice];
20      if (map.empty())
21      {
22          std::cout << "No adjacent vertices! \n";
23      }
24      else
25      {
26          std::cout << vertice << ": ";
27          for (auto& el : map)
28          {
29              std::cout << "(" << el.first << ", " <<
↪ el.second << ") ";
30          }
31          std::cout << '\n';
32      }
33  }
34
35  //oriented graphs: only vertices which are reachable from current
36  void task3_9(Graph* graph)
37  {
38      string vertice;

```

```

39     std::cout << "Enter vertice: ";
40     getline(cin, vertice);
41
42     auto list = graph->GetAdjacencyList();
43     if (list.find(vertice) == list.end())
44     {
45         std::cout << "No such vertice in graph\n";
46         return;
47     }
48     auto mapTarget = list[vertice];
49     uint16_t counter = 0;
50     {
51         for (auto el : list)
52         {
53             if (el.second.size() > mapTarget.size())
54             {
55                 std::cout << el.first << ' ';
56                 ++counter;
57             }
58         }
59         if (!counter)
60             std::cout << "No such vertices!";
61         std::cout << '\n';
62     }
63 }
64
65 void task4_10(Graph* graph1, Graph* graph2)
66 {
67     if (graph1->GetOrientation() != graph2->GetOrientation())
68     {
69         std::cout << "Graphs are incompatible\n";
70         return;
71     }
72

```

```

73         auto list1 = graph1->GetAdjacencyList();
74         auto list2 = graph2->GetAdjacencyList();
75
76         map<string, map<string, int32_t>> newMap;
77         for (auto it : list1)
78             newMap[it.first];
79         for (auto it : list2)
80             newMap[it.first];
81
82         for (auto it1 : newMap)
83         {
84             for (auto it2 : newMap)
85             {
86                 auto f1 =
↪ list1[it1.first].find(it2.first);
87                 auto f2 =
↪ list2[it1.first].find(it2.first);
88
89                 if (f1 != list1[it1.first].end() && f2 ==
↪ list2[it1.first].end())
90                 {
91                     newMap[it1.first][it2.first] =
↪ list1[it1.first][it2.first];
92                 }
93                 else if (f1 == list1[it1.first].end() &&
↪ f2 != list2[it1.first].end())
94                 {
95                     newMap[it1.first][it2.first] =
↪ list2[it1.first][it2.first];
96                 }
97             }
98         }
99     }

```

```

100         Graph* newGraph = new Graph(newMap,
    ↪     graph1->GetOrientation());
101         PrintVertices(newGraph);
102     }
103
104 void task5_2(Graph* graph)
105 {
106     string vertice;
107     std::cout << "Enter target vertice: ";
108     getline(cin, vertice);
109     map<string, bool> used;
110     auto list = graph->GetAdjacencyList();
111     bool allFound = true;
112     for (auto it : list)
113     {
114         used[it.first] = false;
115     }
116     bfs(list, vertice, used);
117     for (auto it : used)
118     {
119         if (!used[it.first])
120         {
121             allFound = false;
122             std::cout << it.first << ' ';
123         }
124     }
125     if (allFound)
126         std::cout << "All vertices are reachable from "
    ↪     << vertice << '!';
127     std::cout << '\n';
128 }
129
130
131

```

```

132 void task6_20(Graph* graph)
133 {
134     string startVertice;
135     string endVertice;
136     auto list = graph->GetAdjacencyList();
137     std::cout << "Enter start vertice: ";
138     getline(cin, startVertice);
139     std::cout << "Enter end vertice: ";
140     getline(cin, endVertice);
141     if (list.find(startVertice) == list.end())
142     {
143         std::cout << "No vertice 1 represented in
↪ graph \n ";
144         return;
145     }
146     if (list.find(endVertice) == list.end())
147     {
148         std::cout << "No vertice 2 represented in
↪ graph \n ";
149         return;
150     }
151     map<string, bool> used;
152     vector<string> path;
153     int32_t ans = 0;
154
155     for (auto it : list)
156     {
157         used[it.first] = false;
158     }
159     dfs_modified(list, startVertice, endVertice, used, path,
↪ ans);
160     if (!ans)
161         std::cout << "No paths between these 2
↪ vertices! \n ";

```

```

162 }
163
164 //Prim algorithm realisation
165 void task7_prim(Graph* graph)
166 {
167     Graph* tree = prim(graph);
168     if (tree == nullptr)
169         std::cout << "No spanning tree exists (graph
↪ should be connected and undirected) \n ";
170     else
171         PrintVertices(tree);
172 }
173
174 void task8_11(Graph* graph)
175 {
176     map<string, map<string, int32_t>> minimalDistance;
177     auto list = graph->GetAdjacencyList();
178     for (auto vert : list)
179     {
180         minimalDistance[vert.first] = dijkstra(graph,
↪ vert.first);
181     }
182     map<string, int32_t> eccentricity;
183     int32_t radius = INT32_MAX;
184     for (auto vert : minimalDistance)
185     {
186         eccentricity[vert.first] =
↪ std::max_element(vert.second.begin(), vert.second.end(),
187                 [&](const auto p1, const auto p2) {return
↪ p1.second < p2.second; })->second;
188         radius = std::min(eccentricity[vert.first],
↪ radius);
189     }
190

```

```

191         if (radius == INT32_MAX)
192         {
193             std::cout << "No center vertice in graph (graph is
↪ not connected) \n";
194             return;
195         }
196
197         for (auto vert : eccentricity)
198         {
199             if (vert.second == radius)
200                 std::cout << vert.first << ' ';
201         }
202         cout << '\n';
203         return;
204     }
205
206 void task9_17(Graph* graph)
207 {
208     auto floydRes = floyd(graph);
209     map<string, map<string, int32_t>> minimalDistance =
↪ floydRes.first;
210     map<string, map<string, string>> pathVertices =
↪ floydRes.second;
211
212     for (auto vert : minimalDistance)
213     {
214         if (vert.second[vert.first] < 0)
215         {
216             cout << "Negative cycle found, algorithm
↪ terminated \n";
217             return;
218         }
219     }
220

```

```

221         auto list = graph->GetAdjacencyList();
222         std::stack<string> path;
223         for (auto vert1 : list)
224         {
225             for (auto vert2 : list)
226             {
227                 cout << vert1.first << "->" <<
↪ vert2.first << ": ";
228                 if
↪ (minimalDistance[vert1.first][vert2.first] == INT32_MAX)
229                 {
230                     cout << "No path between these
↪ vertices\n";
231                 }
232                 else
233                 {
234                     WayBack(vert1.first, vert2.first,
↪ path, pathVertices);
235                     cout << path.top();
236                     path.pop();
237                     while (!path.empty())
238                     {
239                         cout << "->" <<
↪ path.top();
240                         path.pop();
241                     }
242                     cout << " (" <<
↪ minimalDistance[vert1.first][vert2.first] << ")" << '\n';
243                 }
244             }
245             cout << '\n';
246         }
247     }
248

```



```

249 void task10_1(Graph* graph)
250 {
251     //ADD VERTICE CHECKING
252     string vertice1;
253     string vertice2;
254     string L;
255     std::cout << "Enter start vertice: ";
256     getline(cin, vertice1);
257     if (!graph->isVertice(vertice1))
258     {
259         cout << "No such vertice in graph! \n";
260         return;
261     }
262     std::cout << "Enter end vertice: ";
263     getline(cin, vertice2);
264     if (!graph->isVertice(vertice2))
265     {
266         cout << "No such vertice in graph! \n";
267         return;
268     }
269     std::cout << "Enter maximum path length: ";
270     getline(cin, L);
271     if (!is_number(L))
272     {
273         cout << "Wrong weight value! Enter integer: ";
274         return;
275     }
276
277     auto list = graph->GetAdjacencyList();
278     map<string, map<string, int32_t>> minimalDistance;
279     map<string, map<string, string>> pathVertices;
280     for (auto vert : list)
281     {
282         auto algRes = ford_bellman(graph, vert.first);

```

```

283             if (std::get<0>(algRes) == true)
284             {
285                 cout << "Negative cycle found, algorithm
↪ terminated\n";
286                 return;
287             }
288             else
289             {
290                 minimalDistance[vert.first] =
↪ std::get<1>(algRes);
291                 if (vert.first == vertice1)
292                     pathVertices[vert.first] =
↪ std::get<2>(algRes);
293             }
294         }
295
296         if (minimalDistance[vertice1][vertice2] < std::stoi(L))
297         {
298             cout << "Path shorter than " << L << " between "
↪ << vertice1 << " and " << vertice2
299                 << " exists:\n";
300             std::stack<string> path;
301             WayBack(vertice1, vertice2, path, pathVertices);
302             cout << path.top();
303             path.pop();
304             while (!path.empty())
305             {
306                 cout << "->" << path.top();
307                 path.pop();
308             }
309             cout << " (" <<
↪ minimalDistance[vertice1][vertice2] << ")" << '\n';
310         }
311         else

```

```

312         {
313             cout << "Path shorter than " << L << " between "
↪ << vertice1 << " and " << vertice2
314             << " does not exist.\n";
315         }
316     }
317
318 void task11_net(Graph* graph)
319 {
320     string vertice1;
321     string vertice2;
322     std::cout << "Enter source vertice: ";
323     getline(cin, vertice1);
324     std::cout << "Enter target vertice: ";
325     getline(cin, vertice2);
326     if (vertice1 == vertice2)
327     {
328         cout << "Infinite flow (source and targets are
↪ equal) \n ";
329         return;
330     }
331     if (!graph->isVertice(vertice1))
332     {
333         cout << "No vertice 1 in graph \n ";
334         return;
335     }
336     if (!graph->isVertice(vertice2))
337     {
338         cout << "No vertice 2 in graph \n ";
339         return;
340     }
341     cout << "Maximum flow is " << ford_fulkerson(graph,
↪ vertice1, vertice2) << ". \n ";
342 }

```

ПРИЛОЖЕНИЕ Г

Файлы алгоритмов

Algos.h

```
1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <queue>
6  #include <iostream>
7  #include <map>
8  #include <stack>
9
10 #include "Graph.h"
11
12 void dfs_modified(
13     map<string, map<string, int32_t>>& list,
14     const string& current,
15     const string& end,
16     map<string, bool>& used,
17     vector<string>& path,
18     int32_t& counter);
19 void dfs(map<string, map<string, int32_t>>& list, const string&
    ↪ vertex, map<string, bool>& used);
20 void bfs(map<string, map<string, int32_t>>& list, const string&
    ↪ vertex, map<string, bool>& used);
21 Graph* prim(Graph* graph, string root = "");
22
23 void WayBack(const string& startSource, const string&
    ↪ targetSource, std::stack<string>& path,
24     map<string, map<string, string>>& pathVertices);
25
26 map<string, int32_t> dijkstra(Graph* graph, string root);
27 pair<map<string, map<string, int32_t>>,
28     map<string, map<string, string>>> floyd(Graph* graph);
```

```

29
30 std::tuple<bool, map<string, int32_t>, map<string, string>>
    ↪ ford_bellman(Graph* graph, string root);
31
32 int ford_fulkerson(Graph* graph, string source, string target);

```

Algos.cpp

```

1  #include "algos.h"
2
3  void dfs_modified(
4      map<string, map<string, int32_t>>& list,
5      const string& current,
6      const string& end,
7      map<string, bool>& used,
8      vector<string>& path,
9      int32_t& counter)
10 {
11     if (current == end)
12     {
13         for (auto it = path.begin(); it != path.end();
14             ↪ it++)
15         {
16             std::cout << *it << "->";
17         }
18         ++counter;
19         std::cout << end << '\n';
20         return;
21     }
22     used[current] = true;
23     for (auto it : list[current])
24     {
25         if (!used[it.first])
26         {

```

```

27             path.push_back(current);
28             dfs_modified(list, it.first, end, used,
↪ path, counter);
29             path.pop_back();
30         }
31     }
32     used[current] = false;
33 }
34
35 void dfs(map<string, map<string, int32_t>>& list, const string&
↪ vertice, map<string, bool>& used)
36 {
37     used[vertice] = true;
38     for (auto it : list[vertice])
39     {
40         if (!used[it.first])
41         {
42             dfs(list, it.first, used);
43         }
44     }
45 }
46
47 void bfs(map<string, map<string, int32_t>>& list, const string&
↪ vertice, map<string, bool>& used)
48 {
49     string cur;
50     std::queue<string> q;
51     used[vertice] = true;
52     q.push(vertice);
53     while (!q.empty())
54     {
55         cur = q.front();
56         q.pop();
57         for (auto it : list[cur])

```

```

58         {
59             if (!used[it.first])
60             {
61                 used[it.first] = true;
62                 q.push(it.first);
63             }
64         }
65     }
66 }
67
68 Graph* prim(Graph* graph, string root)
69 {
70     Graph* tree = new Graph(graph->GetOrientation());
71     if (graph->GetOrientation() == true)
72         return nullptr;
73     auto list = graph->GetAdjacencyList();
74
75     map<string, bool> used;
76     map<string, int32_t> minEdge;
77     map<string, string> prevEdge;
78     for (auto it : list)
79     {
80         used[it.first] = false;
81         minEdge[it.first] = INT32_MAX;
82         prevEdge[it.first] = "";
83     }
84
85     dfs(list, list.begin()->first, used);
86     for (auto it : used)
87     {
88         if (used[it.first] == false)
89             return nullptr;
90         used[it.first] = false;
91     }

```

```

92
93     if (root == "")
94     {
95         minEdge[list.begin()->first] = 0;
96         tree->AddVertice(list.begin()->first);
97     }
98     else
99     {
100         minEdge[root] = 0;
101         tree->AddVertice(root);
102     }
103
104
105
106     for (auto it : list)
107     {
108         string v = "";
109         for (auto minVert : list)
110         {
111             if (!used[minVert.first] && (v == "" ||
↪ minEdge[minVert.first] < minEdge[v]))
112                 v = minVert.first;
113         }
114         if (v == "")
115         {
116             //no MST
117             std::cout << "No MST\n";
118             return new Graph();
119         }
120
121         used[v] = true;
122         if (prevEdge[v] != "")
123         {
124             auto treeList = tree->GetAdjacencyList();

```



```

125             tree->AddVertice(v);
126             tree->AddEdge(v, prevEdge[v],
↪ list[v][prevEdge[v]]);
127             //std::cout << v << ' ' << prevEdge[v] <<
↪ '\n';
128         }
129
130         for (auto vert : list[v])
131             if (vert.second < minEdge[vert.first])
132             {
133                 minEdge[vert.first] = vert.second;
134                 prevEdge[vert.first] = v;
135             }
136     }
137     return tree;
138 }
139
140 map<string, int32_t> dijkstra(Graph* graph, string root)
141 {
142     auto list = graph->GetAdjacencyList();
143     map<string, int32_t> minimalDistance;
144     //map<string, string> pathVertices;
145     map<string, bool> used;
146     for (auto vert1 : list)
147     {
148         minimalDistance[vert1.first] = INT32_MAX;
149         //pathVertices[vert1.first] = "";
150         used[vert1.first] = false;
151     }
152     minimalDistance[root] = 0;
153     int32_t newDistance;
154     string currentClosest;
155
156     for (auto clos : list)

```

```

157         {
158             currentClosest = "";
159             for (auto vert : minimalDistance)
160             {
161                 if (!used[vert.first] &&
162                     (currentClosest == "" ||
↪ minimalDistance[vert.first] <
↪ minimalDistance[currentClosest]))
163                     {
164                         currentClosest = vert.first;
165                     }
166             }
167             if (minimalDistance[currentClosest] == INT32_MAX)
168                 break;
169             used[currentClosest] = true;
170             for (auto vert : list[currentClosest])
171             {
172                 newDistance =
↪ minimalDistance[currentClosest] + vert.second;
173                 if (newDistance <
↪ minimalDistance[vert.first])
174                     {
175                         minimalDistance[vert.first] =
↪ newDistance;
176                         //pathVertices[vert.first] =
↪ currentClosest;
177                     }
178             }
179         }
180         return minimalDistance;
181     }
182
183     pair<map<string, map<string, int32_t>>,
184         map<string, map<string, string>>> floyd(Graph* graph)

```

```

185 {
186     auto list = graph->GetAdjacencyList();
187     map<string, map<string, int32_t>> minimalDistance;
188     map<string, map<string, string>> pathVertices;
189     for (auto vert1 : list)
190     {
191         minimalDistance[vert1.first];
192         pathVertices[vert1.first];
193         for (auto vert2 : list)
194         {
195             if (vert1.second.find(vert2.first) !=
↪ vert1.second.end())
196             {
197                 minimalDistance[vert1.first][vert2.first]
↪ = vert1.second[vert2.first];
198                 pathVertices[vert1.first][vert2.first]
↪ = vert1.first;
199             }
200             else
201             {
202                 minimalDistance[vert1.first][vert2.first]
↪ = vert1.first == vert2.first ? 0 : INT32_MAX;
203                 pathVertices[vert1.first][vert2.first]
↪ = "";
204             }
205         }
206     }
207
208     int32_t newDistance;
209     int32_t edge1, edge2;
210     for (auto relaxVertice : list)
211     {
212         for (auto vert1 : list)
213         {

```

```

214             for (auto vert2 : list)
215             {
216                 edge1 =
↪ minimalDistance[vert1.first][relaxVertice.first];
217                 edge2 =
↪ minimalDistance[relaxVertice.first][vert2.first];
218                 if (edge1 != INT32_MAX && edge2 !=
↪ INT32_MAX)
219                 {
220                     newDistance = edge1 +
↪ edge2;
221                     if (newDistance <
↪ minimalDistance[vert1.first][vert2.first])
222                     {
223                         minimalDistance[vert1.fi
↪ = newDistance;
224                         pathVertices[vert1.first
↪ = pathVertices[relaxVertice.first][vert2.first];
225                     }
226                 }
227             }
228         }
229     }
230     return std::make_pair(minimalDistance, pathVertices);
231 }
232
233 void WayBack(const string& startSource, const string&
↪ targetSource, std::stack<string>& path,
234             map<string, map<string, string>>& pathVertices)
235 {
236     path.push(targetSource);
237     if (targetSource == startSource)
238         return;

```

```

239         WayBack(startSource,
    ↪   pathVertices[startSource][targetSource], path,
240             pathVertices);
241     }
242
243     std::tuple<bool, map<string, int32_t>, map<string, string>>
    ↪   ford_bellman(Graph* graph, string root)
244     {
245         auto list = graph->GetAdjacencyList();
246         map<string, int32_t> minimalDistance;
247         map<string, string> pathVertices;
248
249         for (auto vert1 : list)
250         {
251             pathVertices[vert1.first] = "";
252             minimalDistance[vert1.first] = vert1.first == root
    ↪   ? 0 : INT32_MAX;
253         }
254
255         int32_t newDistance;
256         for (int k = 1; k < list.size() + 1; k++)
257         {
258             for (auto u : list)
259             {
260                 for (auto v : list[u.first])
261                 {
262                     if (minimalDistance[u.first] ==
    ↪   INT32_MAX)
263                         continue;
264                     newDistance =
    ↪   minimalDistance[u.first] + list[u.first][v.first];
265                     if (newDistance <
    ↪   minimalDistance[v.first])
266                     {

```

```

267                                     if (k == list.size())
268                                     {
269                                         return
↪ std::make_tuple(true, minimalDistance, pathVertices);
270                                     }
271                                     minimalDistance[v.first] =
↪ newDistance;
272                                     pathVertices[v.first] =
↪ u.first;
273                                     }
274                                 }
275                            }
276                    }
277                    return std::make_tuple(false, minimalDistance,
↪ pathVertices);
278 }
279
280 int dfs_network(const string& cur, const string& target, const
↪ int& minDelta,
281                map<string, map<string, int32_t>>& list,
282                map<string, map<string, int32_t>>& flow,
283                map<string, bool>& used)
284 {
285     if (cur == target)
286         return minDelta;
287     if (used[cur] == true)
288         return 0;
289     used[cur] = true;
290     int minDeltaNew;
291     int curDelta;
292     for (auto vert : list[cur])
293     {
294         curDelta = list[cur][vert.first] -
↪ flow[cur][vert.first];

```

```

295         if (curDelta > 0)
296         {
297             minDeltaNew = dfs_network(vert.first,
↪ target, std::min(minDelta, curDelta), list, flow, used);
298             if (minDeltaNew > 0)
299             {
300                 flow[cur][vert.first] +=
↪ minDeltaNew;
301                 flow[vert.first][cur] -=
↪ minDeltaNew;
302                 return minDeltaNew;
303             }
304         }
305     }
306     return 0;
307 }
308
309 //Вес ребра - это его пропускная способность
310 //Создаём новый тар - это будет текущим потоком
311 int ford_fulkerson(Graph* graph, string source, string target)
312 {
313     auto list = graph->GetAdjacencyList();
314     map<string, map<string, int32_t>> flow;
315     map<string, bool> used;
316
317     for (auto vert1 : list)
318     {
319         used[vert1.first] = false;
320         for (auto vert2 : list[vert1.first])
321             flow[vert1.first][vert2.first] = 0;
322     }
323
324     int delta;
325     int ans = 0;

```

```

326         while (true)
327         {
328             for (auto vert1 : list)
329                 used[vert1.first] = false;
330             delta = dfs_network(source, target, INT32_MAX,
↪ list, flow, used);
331             if (delta > 0)
332                 ans += delta;
333             else
334                 break;
335         }
336
337         return ans;
338     }

```


ПРИЛОЖЕНИЕ Д

Файл Main.cpp

main.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <map>
5
6  #include "Graph.h"
7  #include "App.h"
8  #include "Tasks.h"
9
10 const string DATA_FILE1 = "task11_2.json";
11 const string DATA_FILE2 = "data4.json";
12
13 int main()
14 {
15     string command;
16
17     Graph* graph1;
18     std::ifstream file(DATA_FILE1);
19     if (file.is_open())
20         graph1 = new Graph(file);
21     else
22         graph1 = CreateGraph(command);
23     file.close();
24
25     file.open(DATA_FILE2);
26     Graph* graph2 = new Graph(file);
27     file.close();
28
29     CommandMessage();
30     while (true)
31     {
```

```

32         std::cout << "Command: ";
33         std::getline(std::cin, command);
34         switch (Hashing(command))
35         {
36             case string_code::printVertices:
37                 PrintVertices(graph1);
38                 break;
39             case string_code::addVertice:
40                 AddVertice(graph1);
41                 break;
42             case string_code::removeVertice:
43                 RemoveVertice(graph1);
44                 break;
45             case string_code::addEdge:
46                 AddEdge(graph1);
47                 break;
48             case string_code::removeEdge:
49                 RemoveEdge(graph1);
50                 break;
51             case string_code::changeWeight:
52                 ChangeWeight(graph1);
53                 break;
54             case string_code::unweightGraph:
55                 Unweight(graph1);
56                 break;
57
58             case string_code::saveGraph:
59                 graph1->Save(DATA_FILE1);
60                 std::cout << "Graph saved succesfully \n ";
61                 break;
62
63             case string_code::task2:
64                 task2_14(graph1);
65                 break;

```

```

66         case string_code::task3:
67             task3_9(graph1);
68             break;
69         case string_code::task4:
70             task4_10(graph1, graph2);
71             break;
72         case string_code::task5:
73             task5_2(graph1);
74             break;
75         case string_code::task6:
76             task6_20(graph1);
77             break;
78         case string_code::task7:
79             task7_prim(graph1);
80             break;
81         case string_code::task8:
82             task8_11(graph1);
83             break;
84         case string_code::task9:
85             task9_17(graph1);
86             break;
87         case string_code::task10:
88             task10_1(graph1);
89             break;
90         case string_code::task11:
91             task11_net(graph1);
92             break;
93
94         case string_code::help:
95             CommandMessage();
96             break;
97         case string_code::quit:
98             graph1->Save(DATA_FILE1);
99             return 0;

```

```
100
101         default:
102             std::cout << "Wrong Command Number \n";
103         }
104     }
105 }
```

ПРИЛОЖЕНИЕ Е

Используемые для тестирования файлы

task1_1.json

```
1  {
2      "orient": false,
3      "vertices": {
4          "A": {
5              "B": 1,
6              "C": 1
7          },
8          "B": {
9              "A": 1,
10             "C": 1
11         },
12         "C": {
13             "A": 1,
14             "B": 1
15         }
16     }
17 }
```

task2_1.json

```
1  {
2      "orient": true,
3      "vertices": {
4          "a": {
5              "e": 1,
6              "m": 1
7          },
8          "b": {
9              "c": 1
10         },
11         "c": {
12             "b": 9,
```

```

13         "c": 3
14     },
15     "d": {
16         "k": 1
17     },
18     "e": {
19         "a": 1,
20         "d": 1,
21         "k": 1
22     },
23     "g": {},
24     "k": {},
25     "l": {
26         "a": 1
27     },
28     "m": {
29         "n": 6
30     },
31     "n": {}
32 }
33 }

```

task3_1.json

```

1 {
2     "orient": true,
3     "vertices": {
4         "a": {
5             "e": 3
6         },
7         "b": {
8             "a": 1
9         },
10        "c": {},
11        "d": {
12            "k": 1

```

```

13     },
14     "e": {
15         "l": 9
16     },
17     "k": {},
18     "l": {
19         "k": 1
20     },
21     "u": {
22         "v": 5
23     },
24     "v": {
25         "a": 150,
26         "c": 3
27     }
28 }
29 }

```

task4_1.json

```

1  {
2      "orient": true,
3      "vertices": {
4          "a": {
5              "b": 1
6          },
7          "b": {
8              "a": 1,
9              "c": 1
10         },
11         "c": {
12             "b": 1
13         },
14         "d": {
15             "e": 1
16         },

```

```

17         "e": {
18             "d": 1,
19             "f": 5
20         },
21         "f": {
22             "e": -3,
23             "n": 14
24         },
25         "n": {},
26         "o": {}
27     }
28 }

```

task4_2.json

```

1 {
2     "orient": true,
3     "vertices": {
4         "a": {
5             "e": 1,
6             "m": 1
7         },
8         "b": {
9             "c": 1
10        },
11        "c": {
12            "b": 9,
13            "c": 3
14        },
15        "d": {
16            "e": 7,
17            "k": 1
18        },
19        "e": {
20            "a": 1,
21            "d": 1,

```



```

22         "k": 1
23     },
24     "g": {
25         "k": 33
26     },
27     "k": {},
28     "l": {
29         "a": 1
30     },
31     "m": {}
32 }
33 }

```

task5_1.json

```

1 {
2     "orient": true,
3     "vertices": {
4         "A": {
5             "B": 1
6         },
7         "B": {
8             "D": 1
9         },
10        "C": {
11            "B": 1,
12            "D": 1,
13            "E": 1
14        },
15        "D": {
16            "G": 1,
17            "L": 1
18        },
19        "E": {
20            "C": 1,
21            "F": 1,

```

```

22         "G": 1
23     },
24     "F": {
25         "A": 1
26     },
27     "G": {
28         "A": 1
29     },
30     "K": {
31         "L": 1
32     },
33     "L": {}
34 }
35 }

```

task6_1.json

```

1  {
2      "orient": false,
3      "vertices": {
4          "a": {
5              "b": 1,
6              "c": 1,
7              "d": 1
8          },
9          "b": {
10             "a": 1,
11             "c": 1
12         },
13         "c": {
14             "a": 1,
15             "b": 1,
16             "d": 1,
17             "e": 1,
18             "f": 1
19         },

```

```

20     "d": {
21         "a": 1,
22         "c": 1,
23         "f": 1
24     },
25     "e": {
26         "c": 1,
27         "f": 1
28     },
29     "f": {
30         "c": 1,
31         "d": 1,
32         "e": 1
33     }
34 }
35 }

```

task7_1.json

```

1  {
2      "orient": false,
3      "vertices": {
4          "a": {
5              "b": 2,
6              "c": 1,
7              "d": 1
8          },
9          "b": {
10             "a": 2,
11             "c": 1
12         },
13         "c": {
14             "a": 1,
15             "b": 1,
16             "d": 1,
17             "e": 1,

```

```

18         "f": 1
19     },
20     "d": {
21         "a": 1,
22         "c": 1,
23         "f": 1
24     },
25     "e": {
26         "c": 1,
27         "f": 1
28     },
29     "f": {
30         "c": 1,
31         "d": 1,
32         "e": 1
33     }
34 }
35 }

```

task8_1.json

```

1  {
2      "orient": false,
3      "vertices": {
4          "A": {
5              "B": 7,
6              "D": 5
7          },
8          "B": {
9              "A": 7,
10             "C": 8,
11             "D": 9,
12             "E": 7
13         },
14         "C": {
15             "B": 8,

```

```

16         "E": 5
17     },
18     "D": {
19         "A": 5,
20         "B": 9,
21         "E": 15,
22         "F": 6
23     },
24     "E": {
25         "B": 7,
26         "C": 5,
27         "D": 15,
28         "F": 8,
29         "G": 9
30     },
31     "F": {
32         "D": 6,
33         "E": 8,
34         "G": 11
35     },
36     "G": {
37         "E": 9,
38         "F": 11
39     }
40 }
41 }

```

task8_2.json

```

1 {
2     "orient": false,
3     "vertices": {
4         "A": {
5             "B": 1,
6             "D": 1
7         },

```

```

8      "B": {
9          "A": 1,
10         "C": 1
11     },
12     "C": {
13         "B": 1,
14         "D": 1
15     },
16     "D": {
17         "A": 1,
18         "C": 1
19     },
20     "K": {
21         "L": 7
22     },
23     "L": {
24         "K": 7
25     }
26 }
27 }

```

task9_1.json

```

1  {
2      "orient": false,
3      "vertices": {
4          "A": {
5              "B": 7,
6              "D": 5
7          },
8          "B": {
9              "A": 7,
10             "C": 8,
11             "D": 9,
12             "E": 7
13         },

```

```

14         "C": {
15             "B": 8,
16             "E": 5
17         },
18         "D": {
19             "A": 5,
20             "B": 9,
21             "E": 15,
22             "F": 6
23         },
24         "E": {
25             "B": 7,
26             "C": 5,
27             "D": 15,
28             "F": 8,
29             "G": 9
30         },
31         "F": {
32             "D": 6,
33             "E": 8,
34             "G": 11
35         },
36         "G": {
37             "E": 9,
38             "F": 11
39         }
40     }
41 }

```

task9_2.json

```

1 {
2     "orient": false,
3     "vertices": {
4         "A": {
5             "B": 1,

```

```

6         "D": 1
7     },
8     "B": {
9         "A": 1,
10        "C": 1
11    },
12    "C": {
13        "B": 1,
14        "D": 1
15    },
16    "D": {
17        "A": 1,
18        "C": 1
19    },
20    "K": {
21        "L": 7
22    },
23    "L": {
24        "K": 7
25    }
26 }
27 }

```

task9_3.json

```

1  {
2      "orient": true,
3      "vertices": {
4          "A": {
5              "B": -5
6          },
7          "B": {
8              "C": 4
9          },
10         "C": {
11             "A": 2,

```



```

12         "D": -4
13     },
14     "D": {
15         "A": 5
16     }
17 }
18 }

```

task10_1.json

```

1 {
2     "orient": false,
3     "vertices": {
4         "A": {
5             "B": 7,
6             "D": 5
7         },
8         "B": {
9             "A": 7,
10            "C": 8,
11            "D": 9,
12            "E": 7
13        },
14        "C": {
15            "B": 8,
16            "E": 5
17        },
18        "D": {
19            "A": 5,
20            "B": 9,
21            "E": 15,
22            "F": 6
23        },
24        "E": {
25            "B": 7,
26            "C": 5,

```

```

27         "D": 15,
28         "F": 8,
29         "G": 9
30     },
31     "F": {
32         "D": 6,
33         "E": 8,
34         "G": 11
35     },
36     "G": {
37         "E": 9,
38         "F": 11
39     }
40 }
41 }

```

task10_2.json

```

1  {
2      "orient": false,
3      "vertices": {
4          "A": {
5              "B": 1,
6              "D": 1
7          },
8          "B": {
9              "A": 1,
10             "C": 1
11         },
12         "C": {
13             "B": 1,
14             "D": 1
15         },
16         "D": {
17             "A": 1,
18             "C": 1

```

```

19     },
20     "K": {
21         "L": 7
22     },
23     "L": {
24         "K": 7
25     }
26 }
27 }

```

task10_3.json

```

1  {
2      "orient": true,
3      "vertices": {
4          "A": {
5              "B": -5
6          },
7          "B": {
8              "C": 2
9          },
10         "C": {
11             "A": 2,
12             "D": -4
13         },
14         "D": {
15             "A": 5
16         }
17     }
18 }

```

task11_1.json

```

1  {
2      "orient": true,
3      "vertices": {

```

```

4      "A": {
5          "B": 17,
6          "C": 9
7      },
8      "B": {
9          "C": 5,
10         "E": 10
11     },
12     "C": {
13         "D": 13
14     },
15     "D": {
16         "F": 2,
17         "G": 8
18     },
19     "E": {
20         "F": 6,
21         "G": 7
22     },
23     "F": {
24         "G": 15
25     },
26     "G": {}
27 }
28 }

```

task11_2.json

```

1  {
2      "orient": true,
3      "vertices": {
4          "A": {
5              "B": 6,
6              "D": 4
7          },
8          "B": {

```

```

9           "E" : 5
10        },
11        "C" : {
12            "A" : 1,
13            "B" : 1,
14            "G" : 3
15        },
16        "D" : {
17            "G" : 10
18        },
19        "E" : {
20            "C" : 3,
21            "F" : 2,
22            "I" : 1
23        },
24        "F" : {
25            "I" : 5
26        },
27        "G" : {
28            "F" : 3,
29            "H" : 3
30        },
31        "H" : {
32            "I" : 0
33        },
34        "I" : {}
35    }
36 }

```