**ЛАБОРАТОРНАЯ РАБОТА №3**

**ВНИМАНИЕ GIT!**

**Задание ОБЯЗАТЕЛЬНО должно выполняться под системой контроля версий git**

Принцип работы тот же, что и на семинарах, а именно:

- Необходимо завести **публичный** удалённый репозиторий на сервере МИЭМ: https://git.miem.hse.ru/
- Синхронизировать локальный репозиторий с удалённым и настроить конфигурацию так, чтобы:
  - **user.name** = ваши **Фамилия** и **Имя** (на английском с заглавных букв)
  - **user.email** = ваша **корпоративная** почта
- Наличие истории коммитов, **КАЖДЫЙ** из которых:
  - является работоспособной версией программы
  - имеет однострочное сообщение, наглядно передающее суть данного небольшого изменения
  - вы являетесь как его автором, так и коммитером (т.е. в обоих случаях указаны ваши **Фамилия Имя** и **корпоративная почта**)

  (Если в истории всего несколько коммитов, то условие считается невыполненным)
- В Smart LMS нужно загрузить ссылку на удалённый репозиторий с вашей работой (это можно сделать в самом начале работы над лабораторной, чтобы потом не было проблем с дедлайном)

**ВНИМАНИЕ CMAKE!**

**Использовать автоматизированную систему сборки cmake для организации проекта.**

**Невыполнение любого из этих условий приводит к тому, что ЛР вовсе не проверяется!**

**ОЦЕНИВАНИЕ**

- За каждый из пунктов можно получить максимум 1 балл, т.е. за полностью выполненное задание можно получить максимум 8 баллов.
- Дополнительные баллы на оценку 9/10 можно получить при выполнении чего-то сверх требуемого в задании.
- Наличие правильно работающего кода без способности объяснить его расценивается как невыполненное задание т.е. 0 баллов.

**РЕКОМЕНДАЦИИ**

В условиях данной ЛР встречаются «*Рекомендации*: …» - их исполнение не обязательно, однако качество кода может стать лучше, если их придерживаться.

**СПИСЫВАНИЕ**

Приводит к **ОБНУЛЕНИЮ** накопленной…

Поэтому лучше сделать меньше, но самостоятельно.

**ЗАДАНИЕ:**

В данной лабораторной работе предлагается разработать консольное приложение для поиска на графе компонент сильной связности и кратчайшего пути. Через командную строку пользователь будет указывать текстовый файл, в котором записана матрица смежности, по которой строится граф, а также имена узлов откуда и до куда необходимо построить кратчайший маршрут. В результате работы программы в консоль будет выводиться информация о кратчайшем пути (последовательность вершин и его вес), а также перечисление сильных компонент связности графа в следующем виде:

```
D:\LR>graph --file from_LR.txt --from 2 --to 1
Shortest route (weight 14.5): 2 4 3 1
Strongly connected components:
0) 0
1) 1 3
2) 2
3) 4

D:\LR>graph --file no_edges.txt --from 1 --to 0
Threre is no way from '1' to '0'!
Strongly connected components:
0) 0
1) 1
2) 2
3) 3
4) 4

D:\LR>graph --file two_positive_triangles.txt --from 0 --to 5
Threre is no way from '0' to '5'!
Strongly connected components:
0) 0 1 2
1) 3 4 5

D:\LR>graph --file two_negative_triangles.txt --from 1 --to 2
Threre is negative cycle on the way from '1' to '2'!
Strongly connected components:
0) 0 1 2
1) 3 4 5
```

**Пункт 1** (подмодули)

Для разработки такого приложения необходимо создать новый проект и с помощью механизма git-подмодулей подключить в него разработанные ранее в ЛР-2 header-only библиотеки, позволяющие работать с матрицами (класс `linalg::Matrix`) и графами (класс `graph::Graph`).

*Примечание*: устройство класса матрицы и графа в данной ЛР проверяться не будет, но всё же важно подключить себе надёжно работающую версию (предпочтительнее, чтобы это была ваша версия из прошлых ЛР, но также допускается подключение версии кого-то из одногруппников – очевидно)

*Рекомендация*: предлагается сделать следующие модернизации:

1) Добавить методы **.begin()** и **.end()** в `Matrix` => так сможете ходить по элементам матрицы с помощью STL алгоритмов.
2) Добавить метод **.find(…)**, в `Graph` => так сможете по ключу находить итератор, нацеленный на пару {ключ: вершина} => удобно при реализации алгоритмов

Это добавляет удобства при дальнейшей разработке. Если вы используете чужие классы, а не свои, попросите их владельца (вашего одногруппника) внести необходимые для вас изменения.

**Пункт 2** (псевдонимы и константы)

Для гибкости, при разработке вашего ПО, предлагается ввести следующие псевдонимы и глобальные константы:

| | |
|---|---|
| node_name_t | Тип данных для имен вершин графа (целые >= 0) |
| weight_t | Тип данных, представляющий веса рёбер и маршрутов (double) |
| matrix_t | Тип данных, представляющий матрицу смежности с весами weight_t |
| graph_t<T> | Шаблонный тип данных – граф, где имена вершин – node_name_t, веса в рёбрах – weight_t, а данные в вершинах – T<br>T - тип данных, хранящийся в вершинах графа, который будет назначаться в каждом алгоритме по своему (см. пункты 5-7) |
| components_t | Тип данных, представляющий из себя отсортированное множество сильных компонент связности, где каждый из таких компонент хранится в виде отсортированного множества номеров вершин node_name_t (см. пункт 5). |
| route_t | Тип данных – маршрут, т.е. последовательность из ключей node_name_t (см. пункты 6 и 7). |
| INF | Вещественная константа, обозначающая бесконечность |
| EPS | Вещественная константа, обозначающая точность при сравнении двух вещественных чисел (двух double) |

*Рекомендация*: *приведённые псевдонимы и константы можно вынести в отдельный файл, который подключать туда, где они требуются.*

**Пункт 3** (командные аргументы)

Взаимодействие с пользователем будет происходить через командную строку, а значит необходимо разработать функцию, занимающуюся парсингом аргументов (при чем командные аргументы могут быть расположены в произвольном порядке):

```
std::tuple<const char*, node_name_t, node_name_t> parse_args(int arg_count, char* arg_vars[])
```

```
graph.exe --file "graph.txt" --from 0 --to 5
graph.exe --file "graph.txt" --to 5 --from 0
graph.exe --from 0 --file "graph.txt" --to 5
graph.exe --to 5 --from 0 --file "graph.txt"
```

В случае нештатных ситуаций кидать исключения, например:

- *недостаточное количество аргументов;*
- *неправильный формат аргументов;*
- *дублирующийся аргумент.*

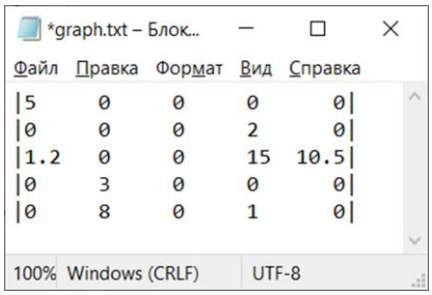
```
D:\LR>graph --file two_negative_triangles.txt --from 1 --to
Invalid number of arguments
Usage: graph --file <path-name> --from <node-index> --to <node-index>

D:\LR>graph --file two_negative_triangles.txt --from 1 --to a32
Invalid node: a32
Usage: graph --file <path-name> --from <node-index> --to <node-index>

D:\LR>graph --file two_negative_triangles.txt --from 1 --tooo 3
Invalid argument: --tooo
Usage: graph --file <path-name> --from <node-index> --to <node-index>

D:\LR>graph --file two_negative_triangles.txt --from 1 --from 3
Dublicated argument: --from
Usage: graph --file <path-name> --from <node-index> --to <node-index>
```

## **Пункт 4** (файл -> матрица -> граф)

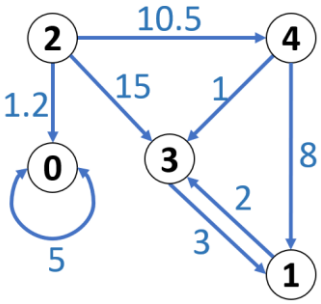
Разработать функцию для чтения матрицы смежности из текстового файла:

```
// Возвращает граф и положительны ли веса рёбер графа:
matrix_t load_matrix(const char* filename);
```

Строки матрицы в файле должны быть обособлены с двух сторон символами вертикальных палочек, например:




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | 5 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 2 | 0 |
| **2** | 1.2 | 0 | 0 | 15 | 10.5 |
| **3** | 0 | 3 | 0 | 0 | 0 |
| **4** | 0 | 8 | 0 | 1 | 0 |




В случае нештатных ситуаций кидать исключения, например:

- *файла с таким именем не найден;*
- *некорректное содержимое файла;*
- *матрица из аргументов не квадратная;*
- *разное кол-во элементов в строках;*

```
D:\LR>graph --file not_found.txt --from 1 --to 3
File not found: not_found.txt

D:\LR>graph --file invalid.txt --from 1 --to 3
Invalid row end boundaries 'sdf' in row 2 (should be '|')

D:\LR>graph --file not_squared_long.txt --from 1 --to 3
Some trash exists after the squared matrix!

D:\LR>graph --file not_squared_wide.txt --from 1 --to 3
The matrix is not squared (not enough rows)!

D:\LR>graph --file different_rows.txt --from 1 --to 3
Invalid elements number in row: 2 (expected: 5, actual: 4)
```

*Рекомендация:* после считывания матрицы, её можно проанализировать на наличие рёбер с отрицательным весом и исходя из этого далее выбрать алгоритм для поиска кратчайшего пути.

Также необходимо реализовать шаблонную функцию, конвертирующую матрицу в граф:

```
template<typename Node> graph_t<Node> create_graph(const matrix_t& matr) noexcept;
```

Создание графов по матрице смежности с нужным типом вершин будет происходить внутри тела реализации алгоритмов => каждый из алгоритмов сможет графу назначить удобный для него тип вершин (описано далее в пунктах 5-7).

## **Пункт 5** (Косараджу-Шарир)

Исследовать граф на количество сильных компонент связности с помощью алгоритма Косараджу-Шарира.

```
components_t compute_components(const matrix_t& matr) noexcept;
```

Сначала по матрице смежности необходимо построить граф `graph_t<bool>`, где `bool` – это тип данных вершины, означающий посещена ли вершина (при обходе с помощью DFS).

*Рекомендация:* предложение о том, как может выглядеть основное тело алгоритма:

```
graph_t<bool> graph_initial = create_graph<bool>(matr);
// 1 этап: создание инвертированного графа
graph_t<bool> graph_inverted = create_graph<bool>(transpose(matr));
// 2 этап: топологическая сортировка инвертированного графа
std::vector<node_name_t> sorted_nodes = topology_sort(graph_inverted);
// 3 этап: обход по исходному графу с помощью DFS в порядке топологич. сортировки
return components_by_dfs(graph_initial, sorted_nodes);
```

**Пункт 6** (Дейкстра)

Реализовать алгоритм Дейкстры, который будет возвращать пару из веса кратчайшего маршрута и сам маршрут (от key_from до key_to)

```
std::pair<weight_t, route_t> dijkstra(const matrix_t& matr,
                                      node_name_t key_from,
                                      node_name_t key_to) noexcept;
```

Сначала по матрице смежности необходимо построить граф graph_t<NodeDijkstra>, где NodeDijkstra – это тип данных вершины, хранящий в себе:

- текущую метку (по дефолту бесконечность)
- «посещена» ли вершина (по дефолту нет)
- откуда пришла вершина (имеет смысл только если метка не бесконечность)

При реализации алгоритма использовать собственный контейнер NodesToBeVisited, реализованный на базе std::vector который хранит в себе итераторы на {ключ: вершина} графа. В таком классе удобно реализовать следующие конструктор и методы:

- конструктор, который принимает граф и на основе всех его вершин заполняет свой внутренний контейнер – т.е. сначала все вершины графа «не посещены».
- **.pop_min_weight()**, который извлекает итератор из контейнера с минимальным весом, меняя статус соответствующей вершины на «посещена»
- **.empty()**, который возвращает пуст ли контейнер из непосещённых вершин.

Если путь не удалось найти (т.к. рассматриваемые вершины находятся в разных компонентах связности), то алгоритм должен возвращать маршрут из двух вершин {key_from, key_to} бесконечно большого веса (т.е. INF)

*Рекомендация:* в целях оптимизации можно игнорировать петли положительного веса.

*Рекомендация:* предложение о том, как может выглядеть основное тело алгоритма:

```
// Загрузим граф из матрицы смежности:
graph_t<NodeDijkstra> graph = create_graph<NodeDijkstra>(matr);
// Выполним инициализацию алгоритма:
graph.at(key_from) = 0.0;
NodesToBeVisited nodes_to_be_visited{ graph };
// Запускаем основной цикл алгоритма (работающий с непосещёнными вершинами):
while (!nodes_to_be_visited.empty())
    dijkstra_step(graph, nodes_to_be_visited);
// После чего восстановим маршрут.
return // вернём маршут и его вес.
```

**Пункт 7** (SPFA)

Реализовать алгоритм SPFA, который будет возвращать пару из веса кратчайшего маршрута и сам маршрут (от key_from до key_to)

```
std::pair<weight_t, route_t> spfa(const matrix_t& matr,
                                  node_name_t key_from,
                                  node_name_t key_to) noexcept;
```

Сначала по матрице смежности необходимо построить граф graph_t<NodeSPFA>, где NodeSPFA – это тип данных вершины, хранящий в себе:

- текущую метку (по дефолту бесконечность)
- количество обновлений текущей метки (по дефолту 0 – для поиска отриц. циклов)
- «в работе» ли вершина (по дефолту нет)
- откуда пришла вершина (имеет смысл только если метка не бесконечность)

При реализации алгоритма использовать собственную очередь NodeQueue, реализованную на базе std::deque который хранит в себе итераторы на {ключ: вершина} графа. В таком классе удобно реализовать следующие методы:

- **.push**(it), который помечает вершину статусом «в работе» и добавляет соответствующий итератор в конец очереди.
- **.pop**(), который извлекает итератор из начала очереди, сняв с соответствующей вершины статус «в работе»
- **.empty**(), который возвращает пуста ли очередь.

Возвращаемые значения алгоритма в случае, если путь не удалось найти:

- если рассматриваемые вершины находятся в разных компонентах связности, то алгоритм должен возвращать маршрут из двух вершин {key_from, key_to} бесконечно большого веса (т.е. INF);
- если на пути между рассматриваемыми вершинами встретился отрицательный цикл, то алгоритм должен возвращать маршрут из двух вершин {key_from, key_to} бесконечно маленького веса (т.е. -INF).

*Примечания*: при реализации алгоритма важно корректно обрабатывать случаи с отрицательными циклами, иначе алгоритм рискует зайти в бесконечный цикл…

*Рекомендация:* в целях оптимизации можно игнорировать петли положительного веса, а для петли отрицательного веса можно сразу назначать метку -INF.

*Рекомендация:* предложение о том, как может выглядеть основное тело алгоритма:

```
// Загрузим граф из матрицы смежности:
graph_t<NodeSPFA> graph = create_graph<NodeSPFA>(matr);
// Выполним инициализацию алгоритма:
auto& node_start_it = graph.find(key_from);
node_start_it->second.value() = 0.0;
NodeQueue nodes_in_work;
nodes_in_work.push(node_start_it);
// Запускаем основной цикл алгоритма (работающий с вершинами из очереди):
while (!nodes_in_work.empty())
    spfa_step(graph, nodes_in_work);
// После чего восстановим маршрут.
return // вернём маршут и его вес.
```

**Пункт 8** (Многопоточность)

Оптимизировать время работы программы с помощью распараллеливания независимых друг от друга процессов, например:

1) Процедуру поиска сильных компонент связности можно проводить параллельно с процедурой поиска кратчайшего пути.
2) В алгоритме Косараджу-Шарира можно проводить построение исходного графа параллельно с построением и топологической сортировкой инвертированного графа.

При защите ЛР, результаты оптимизации, необходимо продемонстрировать замеряя время работы программы с «большими» графами.

*Предложения для получения дополнительных баллов* (это лишь некоторые идеи):

1) Если на пути от $key\_to$, до $key\_from$ обнаружен цикл отрицательного веса, то не просто оповещать о нём, но и выводить последовательность вершин, из которых он состоит.
2) Реализовать тестирование с помощью сторонних средств, например: **CTest**, **Google Test**, **Native Unit Tests** (VS) и т.д.
3) Отрисовать исследуемый граф и выделить кратчайший маршрут (например, с помощью **graphviz**).