Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-1 «Системы автоматического управления»

ОТЧЕТ

по домашней работе №1

«Численные методы решения ОДУ»

по дисциплине

«Основы теории систем»

Выполнили: **Шевченко А.Д.**
Уткин Т.О.
Степанян А.Н.

Группа: **ИУ1-42Б**

Проверил: **Лукьянов Н.В.**

Работа выполнена: **03.04.2023**

Отчет сдан: **04.04.2023**

Оценка:

Москва 2023

Оглавление

Цель работы

Реализация метода обхода графа 1-k BFS и поиска кратчайшего пути между заданными вершинами, на основе Московского метрополитена.

Описание

Поиск в ширину (англ. breadth-first search) — один из основных алгоритмов на графах, позволяющий находить все кратчайшие пути от заданной вершины и решать многие другие задачи.

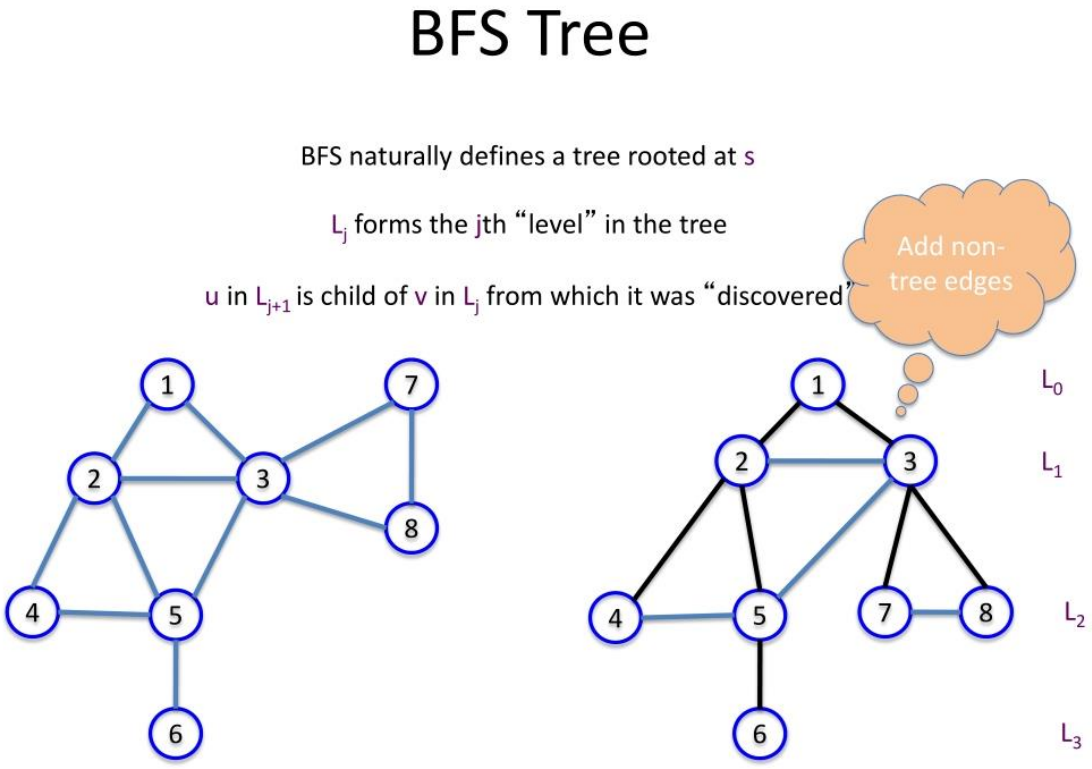


Рис. BFS Tree

Основную идею алгоритма можно понимать как процесс «поджигания» графа: на нулевом шаге мы поджигаем вершину s, а на каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей, в конечном счете поджигая весь граф.

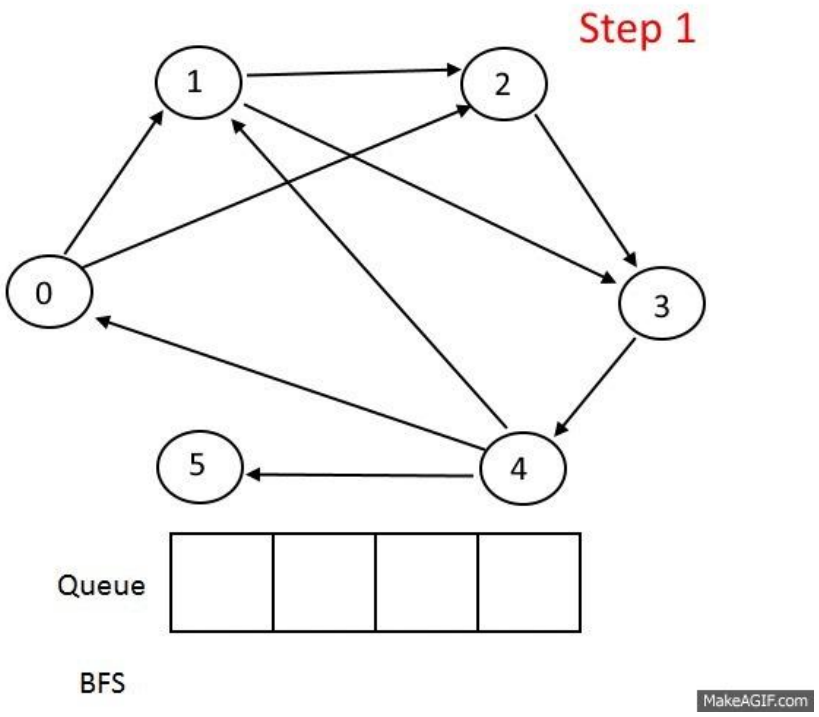


Рис. Наглядная демонстрация работы алгоритма

Белый — вершина, которая ещё не обнаружена. Синий — вершина, извлечённая из очереди.

Создадим очередь, в которую будут помещаться горящие вершины.

Затем алгоритм представляет собой такой цикл: пока очередь не пуста, достать из её головы одну вершину v, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из смежных вершин u ещё не горят, поджечь их и поместить в конец очереди.

```
while (atDist[pos % 4].empty()) { // Выбираем длину пути, на котором исследуем новые вершины
  ++pos;
}

int u = atDist[pos % 4].front(); // Выбираем вершину
atDist[pos % 4].pop();
--kol;
```

Рис. Реализация алгоритма в программе на C++

В неявных графах

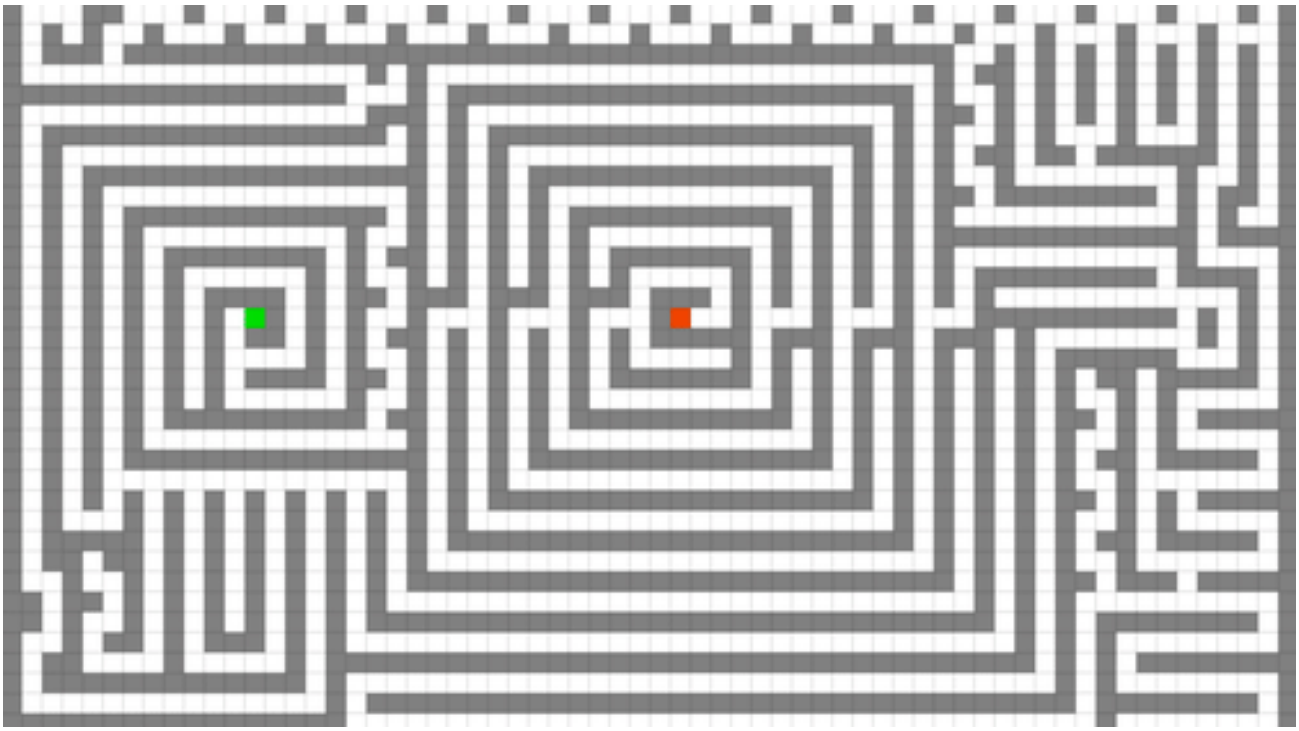
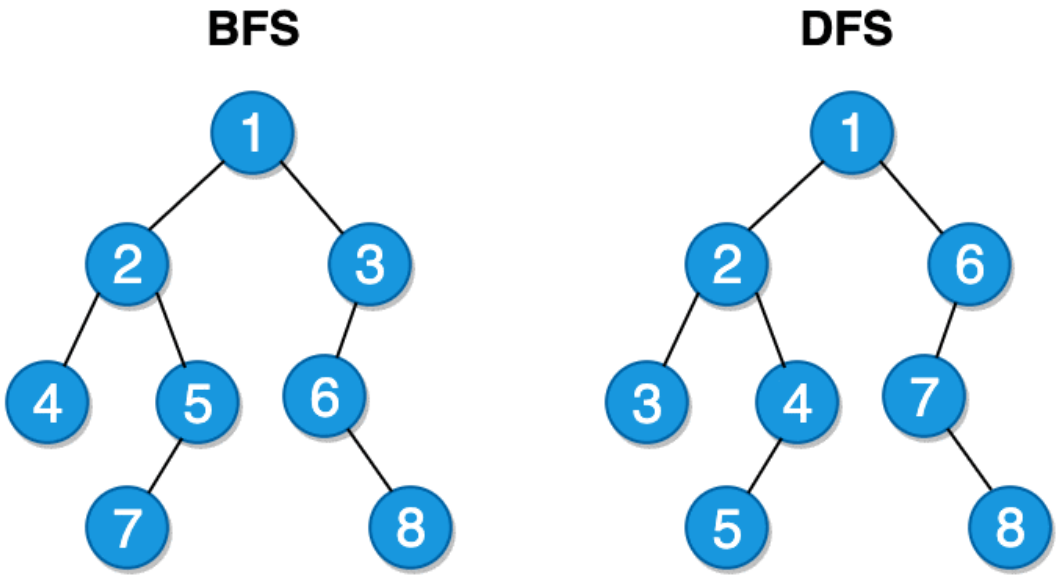


Рис. Поиск в ширину часто применяется для поиска кратчайшего пути в неявно заданных графах.



GIF. Обход графа в ширину и глубину

Теперь веса рёбер принимают значения от 1 до некоторого небольшого k, и всё так же требуется найти кратчайшие расстояния от вершины s, но уже в плане суммарного веса.

Наблюдение: максимальное кратчайшее расстояние в графе равно (n-1)*k

```
std::vector<std::queue<int> > atDist(4);
```

Рис. Структура данных поиска вершины

Если у каждого узла имеется конечное число преемников, алгоритм является полным: если решение существует, алгоритм поиска в ширину его находит, независимо от того, является ли граф конечным. Однако если решения не существует, на бесконечном графе поиск не завершается.

Временная сложность

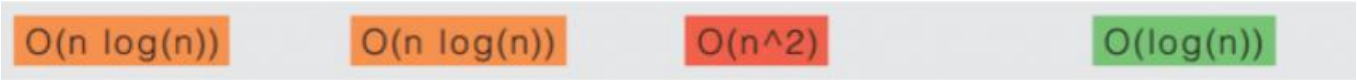


Рис. Алгоритмическая сложность алгоритма поиска

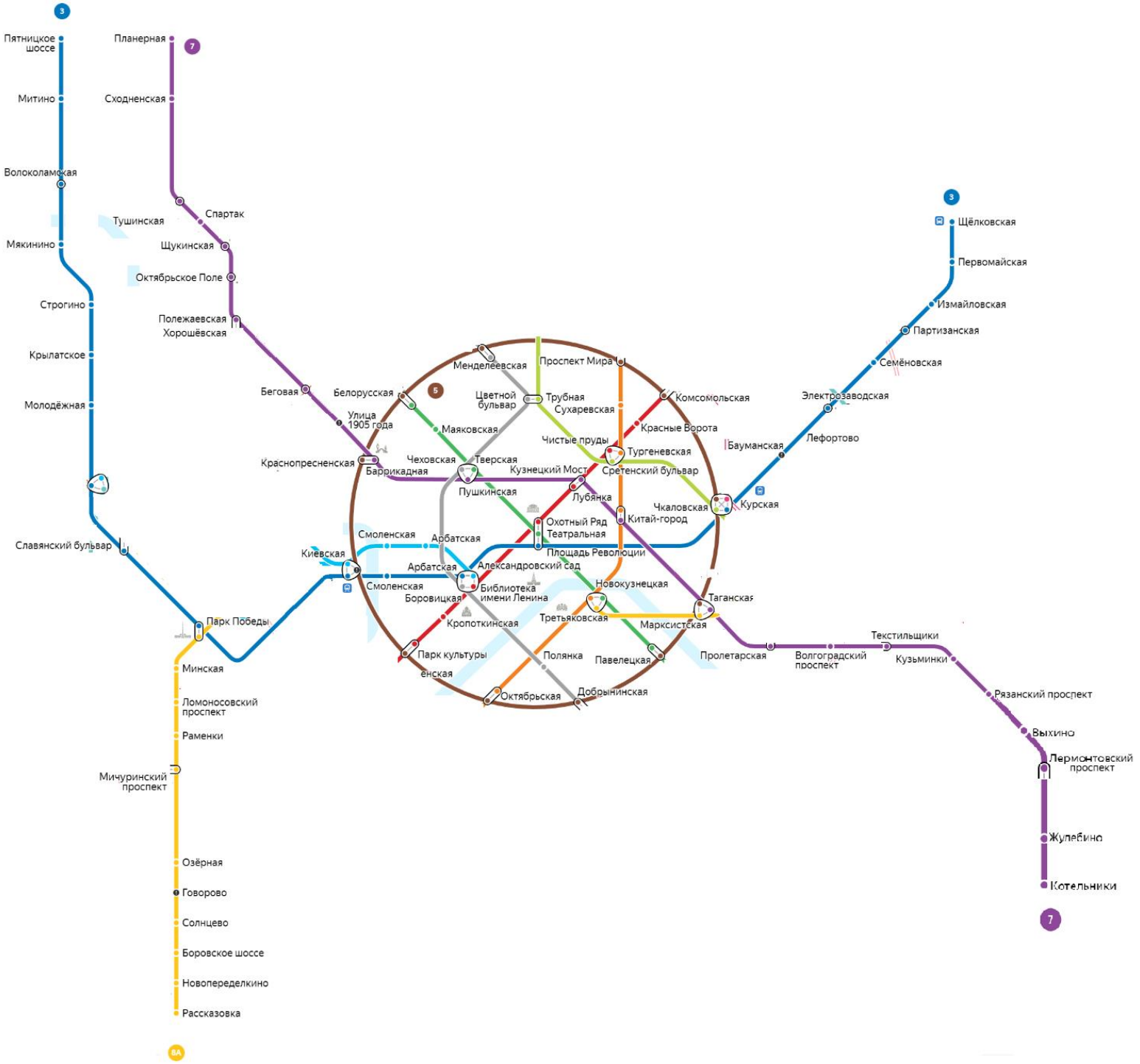
Сложность такого алгоритма будет O(kn+m), поскольку каждую вершину мы можем прорелаксировать и добавить в другую очередь не более k раз, а просматривать рёбра, исходящие из вершины мы будем только когда обработаем эту вершину в самый первый раз.

Полнота

Если у каждого узла имеется конечное число преемников, алгоритм является полным: если решение существует, алгоритм поиска в ширину его находит, независимо от того, является ли граф конечным. Однако если решения не существует, на бесконечном графе поиск не завершается.

Карта Московского метрополитена

Наша реализованная матрица на сегодняшний день



1) Все станции в пределах Московской Кольцевой, включая её саму.
2) Арбатско-Покровскую (синяя)
3) Таганско-Краснопресненскую (фиолетовая)
4) Солнцевская (жёлтая)

Итого на сегодняшний день у нас потсроен граф на 105 станций.

С полным списком можете ознакомиться по ссылке
https://github.com/BMSTU-Automatic-Control-Systems-IU1-1/fundamentals-of-systems-theory/blob/main/lab-01/stations.txt

Реализация алгоритма

Программу поиска графа кратчайшего пути реализовывали на языке C++
Программа имеет следующую структуру:

1) Заголовочный файл stations.h
2) Исполняемый файл main.cpp
3) Список станций stations.txt

Файл stations.h содержит иниациализацию этого графа
https://github.com/BMSTU-Automatic-Control-Systems-IU1-1/fundamentals-of-systems-theory/blob/main/lab-01/stations.h

```
// Инициализация списка станций
const std::vector<std::string> stations = { "Охотный ряд",  "Театральная", "Площадь Революции", "Тверская",
                                            "Пушкинская", "Чекаловская", "Кузнецкий Мост", "Лубянка",  "Чистые пруды",
                                            "Тургеневская",  "Сретненский бульвар",  "Трубная",  "Цветной бульвар",  "Александровский сад",
                                            "Арбатская 3",  "Боровитская",  "Библиотека имени Ленина",  "Арбатская 4",
                                            "Китай-город 6",  "Китай-город 7",  "Третьяковская 6",  "Третьяковская 8",
                                            "Новокузнецкая",  "Полянка",  "Кропоткинская",  "Смоленская 3",
                                            "Смоленская 4",  "Маяковская",  "Сухаревская",  "Красные ворота",
                                            "Комсомольская 1",  "Комсомольская 5",  "Курская 3",  "Чкаловская",
                                            "Курская 5",  "Марсистская",  "Таганская 7",  "Таганская 5",  "Павелецка 2",
                                            "Павелецкая 5",  "Серпуховская",  "Добровитская",  "Октябрьская 6",
                                            "Октябрьская 5",  "Парк культуры 1",  "Парк культуры 5",  "Киевская 3",  "Киевская 4",
                                            "Киевская 5",  "Баррикадная",  "Краснопресненская",  "Белорусская 2",
                                            "Белорусская 5",  "Менделеевская",  "Новослободская",  "Проспект мира 6",
                                            "Проспект мира 5" };
```

Рис. Описание станций версии графа включающего только Кольцевую

```
// Инициализация графа, номера станций - индексы в списке станций
std::map<int, std::vector<std::pair<int, int> > > metro = {
        { Val1: 0,  Val2: {{ Val1: 1,  Val2: 3}, { Val1: 2,  Val2: 3}, { Val1: 7,  Val2: 1}, { Val1: 16,  Val2: 1}}},
        { Val1: 1,  Val2: {{ Val1: 0,  Val2: 3}, { Val1: 2,  Val2: 3}, { Val1: 3,  Val2: 1}, { Val1: 22,  Val2: 2}}},
        { Val1: 2,  Val2: {{ Val1: 0,  Val2: 3}, { Val1: 1,  Val2: 3}, { Val1: 14,  Val2: 2}, { Val1: 32,  Val2: 2}}},
        { Val1: 3,  Val2: {{ Val1: 4,  Val2: 3}, { Val1: 5,  Val2: 3}, { Val1: 1,  Val2: 1}, { Val1: 27,  Val2: 1}}},
        { Val1: 4,  Val2: {{ Val1: 3,  Val2: 3}, { Val1: 5,  Val2: 3}, { Val1: 6,  Val2: 2}, { Val1: 49,  Val2: 2}}},
        { Val1: 5,  Val2: {{ Val1: 3,  Val2: 3}, { Val1: 5,  Val2: 3}, { Val1: 12,  Val2: 1}, { Val1: 15,  Val2: 2}}},
        { Val1: 6,  Val2: {{ Val1: 7,  Val2: 3}, { Val1: 4,  Val2: 2}, { Val1: 19,  Val2: 1}}},
        { Val1: 7,  Val2: {{ Val1: 6,  Val2: 3}, { Val1: 0,  Val2: 1}, { Val1: 8,  Val2: 2}}},
```

Рис. Часть написанной зависимости между станциями

Файл main.cpp содержит алгоритм поиска 1-k BFS

https://github.com/BMSTU-Automatic-Control-Systems-IU1-1/fundamentals-of-systems-theory/blob/main/lab-01/main.cpp

```cpp
std::vector<int> routes( Count: stations.size(),  Val: 1000);  // Массив кратчайших путей до каждой точки, изначально - 1000
std::vector<int> visited( Count: stations.size(),  Val: 0);         // Массив исследованных вершин, изначально заполнен нулями
std::vector<int> routesVector( Count: stations.size()); // Массив пройденных станций для восстановления пути
```

Рис. Структуры данных, используемые для алгоритма поиска

```cpp
void OneKBFS(const int& from, const int& to) {
    std::vector<std::queue<int> > atDist( Count: 4); // Создаём массив очередей станций, по увеличению пути
    atDist[0].push( Val: from);
    routes[from] = 0;
    int pos = 0, kol = 1; // pos - позиция, на которой рассматривается элемент в массиве очередей
        // kol - общее количество вершин во всех очередях
    routesVector[from] = -1;
    while (kol > 0 && visited[to] == 0) { // Исполняется, пока конечная вершина не исследована или пока не кончатся вершины в массиве очередей

        while (atDist[pos % 4].empty()) { // Выбираем длину пути, на котором исследуем новые вершины
            ++pos;
        }

        int u = atDist[pos % 4].front(); // Выбираем вершину
        atDist[pos % 4].pop();
        --kol;

        if (visited[u] == 0) {

            visited[u] = 1; // Помечаем как посещённую

            for (std::pair<int, int> station : metro[u]) { // Рассматриваем соседние с ней станции
                if (routes[station.first] > (routes[u] + station.second)) {

                    routes[station.first] = routes[u] + station.second;
                    routesVector[station.first] = u;

                    atDist[routes[station.first] % 4].push( Val: station.first); // Добавляем рассмотренные станции в очередь к исследованию

                    ++kol;
                }
            }
        }
    }
}
```

Рис. Алгоритм 1-k BFS

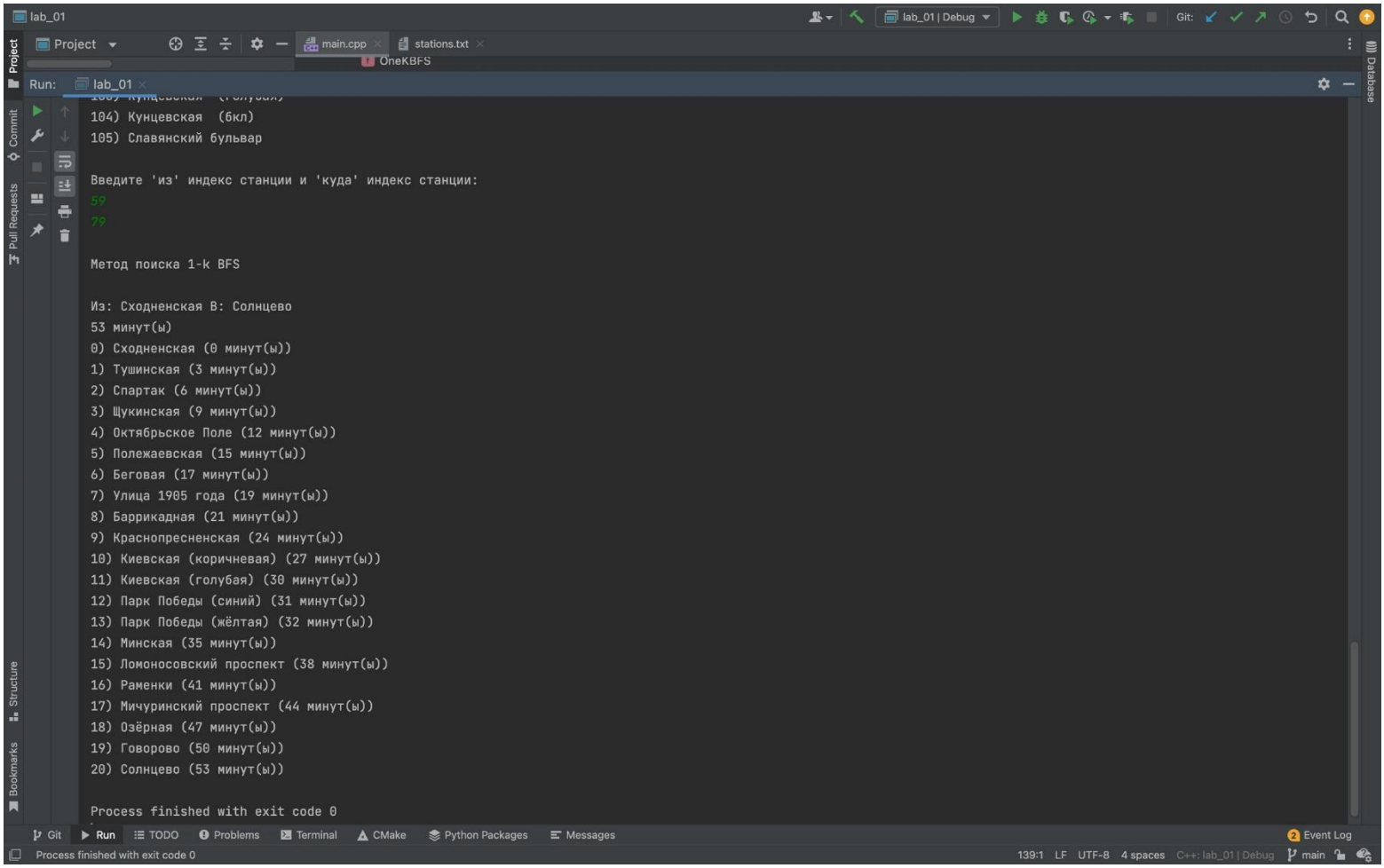


Рис. Пример работы алгоритма на маршруте Сходненская – Солнцево

Вывод:

В данной лабораторной работе был реализован метод обхода графа и поиска кратчайшего пути между двумя заданными вершинами 1-k BFS на языке C++ на примере Московского метрополитена.

Если длины рёбер графа равны между собой, поиск в ширину является оптимальным, то есть всегда находит кратчайший путь. В случае взвешенного графа bfs находит путь, содержащий минимальное количество рёбер, но не обязательно кратчайший. 1-k bfs же всегда находит кратчайший путь по затратам времени