

НАУЧНО-ИССЛЕДОВАТЕЛЬСКАЯ КОРПОРАЦИЯ ИТМО



Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Лабораторная работа № 2

Выполнил
студент

Нестеров Иван Алексеевич

Группа Р33302

Преподаватель: Королёва Юлия Александровна

г. Санкт-Петербург

2022

Цель задания:

Исследование алгоритмов решения задач методом поиска. Описание предметной области. Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т. е. допускают движение в обоих направлениях. Необходимо проложить маршрут из одной заданной точки в другую.

Ход работы:

Код задания:

Vertex.kt – класс с инкапсуляцией логики, присущей вершине графа.

```
data class Vertex(val name: String): Comparable<Vertex> {

    private val neighbors = mutableMapOf<Vertex, Int>()
    var g = 0
        set(value) {
            field += value
            f = value + h
        }
    var h = 0
        set(value) {
            field = value
            f = value + g
        }
    var f = 0
    private set

    /**
     * Соединить вершину с вершиной [v]. [length] - вес ребра.
     */
    fun link(v: Vertex, length: Int) {
        this.neighbors[v] = length
        v.neighbors[this] = length
    }

    /**
     * Соединить вершину с вершиной [v].
     */
    //В предыдущей реализации параметр по умолчанию не использовался, чтобы
    //сохранить инфиксную форму.
    infix fun link(v: Vertex) = link(v, 1)

    /**
     * Разорвать связь с вершиной [v].
     */
    infix fun unlink(v: Vertex) {
        this.neighbors.remove(v)
        v.neighbors.remove(this)
    }

    /**
     * @return прямых соседей вершины, без возможности редактировать их, во
     * избежание нарушения целостности графа.
     */
    fun getNeighbors() = this.neighbors.toMap()

    /**
     * @return является ли [neighbor] прямым соседом вершины.
     */
    operator fun contains(neighbor: Vertex) = neighbors.containsKey(neighbor)
}
```

```

        override fun compareTo(other: Vertex) = this.f - other.f
    }

```

App.kt – класс, предназначенный для запуска программы. Содержит объект с методами – алгоритмами поиска пути по графу.

```

import GraphsAlgorithms.A_StarSearch
import GraphsAlgorithms.bestFirstSearch
import GraphsAlgorithms.bidirectionalSearch
import GraphsAlgorithms.breadthFirstSearch
import GraphsAlgorithms.depthFirstSearch
import GraphsAlgorithms.depthLimitSearch
import GraphsAlgorithms.iterativeDeepeningDepthFirstSearch
import java.io.File
import java.util.*

object GraphsAlgorithms {
    /**
     * Алгоритм поиск в глубину на графах.
     * @return путь от [start] до [finish] и его длину (если веса рёбер не
     заданы, то они = 1).
     */
    fun depthFirstSearch(start: Vertex, finish: Vertex, limit: Int =
Int.MAX_VALUE): Pair<List<Vertex>, Int> {
        val path = Stack<Vertex>()
        //необходимо использовать вложенную функцию, чтобы создать замыкания
        для переменной path
        fun innerDFS(current: Vertex, limit: Int, visited: Set<Vertex> =
setOf()): Boolean =
            when {
                current == finish -> true
                limit == 0 -> false
                else -> {
                    current.getNeighbors().keys.filter { it !in visited
                }.forEach {
                    if (innerDFS(it, limit - 1, visited + current)) {
                        path.push(it)
                        return true
                    }
                }
                false
            }
        innerDFS(start, limit)
        path.push(start) //добавляем в начало пути вершину, из которой начали
поиск
        return path.reversed() to path.size - 1 //вычитаем из длины пути
вершину, из которой начали поиск
    }

    /**
     * Алгоритм поиска в ширину на графах.
     * @return путь от [start] до [finish] и его длину (если веса рёбер не
     заданы, то они = 1).
     */
    fun breadthFirstSearch(start: Vertex, finish: Vertex): Pair<List<Vertex>,
Int> {
        val queue = Stack<Vertex>().apply { this.push(start) }
        val parents = mutableMapOf<Vertex, Vertex?>(start to null) //ключ -
вершина, значение - родитель
        val visited = mutableSetOf(start)
        while (queue.isNotEmpty()) {

```

```

        val next = queue.pop()
        visited.add(next)
        if (next == finish) {
            val path = generateSequence(next) { parents[it]
}.toList().reversed()
            return path to path.size - 1
        }
        next.getNeighbors().keys.filter { it !in visited }.forEach {
            queue.add(it)
            parents[it] = next
        }
    }
    return listOf(start) to 0
}

/**
 * Алгоритм поиск в глубину на графах с ограничением глубины [limit].
 * @return путь от [start] до [finish] и его длину (если веса рёбер не
заданы, то они = 1).
 */
fun depthLimitSearch(start: Vertex, finish: Vertex, limit: Int) =
depthFirstSearch(start, finish, limit)

/**
 * Алгоритм поиск с итеративным углублением. Использует в своей
реализации [breadthFirstSearch].
 * @return путь от [start] до [finish] и его длину (если веса рёбер не
заданы, то они = 1).
 */
fun iterativeDeepeningDepthFirstSearch(start: Vertex, finish: Vertex):
Pair<List<Vertex>, Int> =
    generateSequence(1) { it + 1 }.map { depthFirstSearch(start, finish,
it) }.find { it.first.size > 1 }?: listOf(start) to 0
    // it.first.size > 1 потому что мы должны проверить не длину пути, а
    // больше ли вершин в пути чем 1

/**
 * Алгоритм двунаправленного поиск в графе.
 * @return путь от [start] до [finish] и его длину (если веса рёбер не
заданы, то они = 1).
 */
fun bidirectionalSearch(start: Vertex, finish: Vertex):
Pair<List<Vertex>, Int> {
    val sData = Triple(
        mutableListOf(start), //очередь
        mutableSetOf(start), //посещённые
        mutableMapOf<Vertex, Vertex?>(start to null) //родители
    )
    val fData = Triple(
        mutableListOf(finish),
        mutableSetOf(finish),
        mutableMapOf<Vertex, Vertex?>(finish to null)
    )
    fun innerBFS(vertexData: Triple<MutableList<Vertex>,
MutableSet<Vertex>, MutableMap<Vertex, Vertex?>>) {
        val current = vertexData.first.first()
        vertexData.first.removeFirst()
        current.getNeighbors().keys.filter { it !in vertexData.second
}.forEach {
            vertexData.third[it] = current
            vertexData.second.add(it)
            vertexData.first.add(it)
        }
    }
}

```

```

        while (sData.first.isNotEmpty() && fData.first.isNotEmpty()) {
            innerBFS(sData)
            innerBFS(fData)
            val intersectVertices = sData.second intersect fData.second
            if (intersectVertices.isNotEmpty()) {
                val path = mutableListOf(intersectVertices.first())
                fun buildHalfPath(v: Vertex, data:
Triple<MutableList<Vertex>, MutableSet<Vertex>, MutableMap<Vertex, Vertex?>>)
{
                    var intersect = intersectVertices.first()
                    while (intersect != v) {
                        val parent = data.third[intersect]?: break
                        path.add(parent)
                        intersect = parent
                    }
                }
                buildHalfPath(start, sData)
                path.reverse()
                buildHalfPath(finish, fData)
                return path to path.size - 1
            }
        }
        return listOf(start) to 0
    }

    private fun buildInformPath(parents: Map<Vertex, Vertex?>, lastVertex:
Vertex?): Pair<List<Vertex>, Int> {
        var current = lastVertex
        val path = mutableListOf<Vertex>()
        var roadLength = 0
        while (current != null) {
            path.add(current)
            val parent = parents[current]
            roadLength += current.getNeighbors()[parent]?: 0
            current = parent
        }
        return path.reversed() to roadLength
    }

    /**
     * Алгоритм жадного поиска на графах по первому наилучшему соответствию.
     * @return путь от [start] до [finish] и его длину (если веса рёбер не
заданы, то они = 1).
     */
    fun bestFirstSearch(start: Vertex, finish: Vertex): Pair<List<Vertex>,
Int> {
        val queue = mutableListOf(start)
        val visited = mutableSetOf<Vertex>()
        val parents = mutableMapOf<Vertex, Vertex?>(start to null)
        while (queue.isNotEmpty()) {
            queue.sortByDescending { it.h }
            val current = queue.removeFirstOrNull()!!
            visited.add(current)
            if (current == finish) return buildInformPath(parents, current)
            current.getNeighbors().keys.filter { it !in visited }.forEach { v
->
                if (queue.none { v == it && v.h >= it.h }) {
                    queue.add(v)
                    parents[v] = current
                }
            }
        }
        return listOf(start) to 0
    }
}

```

```

/**
 * Поиск методом минимизации суммарной оценки A* на графах.
 * @return путь от [start] до [finish] и его длину (если веса рёбер не
 * заданы, то они = 1).
 */
fun A_StarSearch(start: Vertex, finish: Vertex): Pair<List<Vertex>, Int>
{
    //используем очередь с приоритетом, чтобы автоматически сортировать
    очередь по параметру Vertex.f
    val queue = PriorityQueue<Vertex>().apply { this.add(start) }
    val visited = mutableSetOf<Vertex>()
    val parents = mutableMapOf<Vertex, Vertex?>(start to null)
    while (queue.isNotEmpty()) {
        val current = queue.poll()
        visited.add(current)
        if (current == finish) return buildInformPath(parents, current)
        current.getNeighbors().keys.filter { it !in visited }.forEach { v
->
            v.g = current.getNeighbors()[v]?: 0
            if (queue.none { v == it && v.g >= it.g }) {
                queue.add(v)
                parents[v] = current
            }
        }
    }
    return listOf(start) to 0
}

fun main(args: Array<String>) {
    val vertices = File("${args[0]}heuristics.txt").readLines()
        .map { it.split(" ") }
        .map { Vertex(it[0]).apply { this.h = it[1].toInt() } }
        .toSet()
    File("${args[0]}graphData.txt").readLines().map { it.split(" ") }.forEach
{ r ->
    val (a, b) = vertices.filter { it.name == r[0] || it.name == r[1] }
        a.link(b, r[2].toInt())
    }
    val (a, b) = vertices.filter { it.name == "Брест" || it.name == "Казань"
}
    println("""
        DFS ${depthFirstSearch(a, b)}
        BFS ${breadthFirstSearch(a, b)}
        DLS ${depthLimitSearch(a, b, 5)}
        IDDFS ${iterativeDeepeningDepthFirstSearch(a, b)}
        BDS ${bidirectionalSearch(a, b)}
        BFS ${bestFirstSearch(a, b)}
        A* ${A_StarSearch(a, b)}
    """).trimIndent()
}

```

Вывод результатов работы каждого из 7 алгоритмов:

```

DFS      ([Vertex(name=Брест),      Vertex(name=Вильнюс),      Vertex(name=Витебск),
Vertex(name=Воронеж),      Vertex(name=Волгоград),      Vertex(name=Житомир),
Vertex(name=Киев), Vertex(name=Кишинев), Vertex(name=Донецк), Vertex(name=Москва),
Vertex(name=Казань)], 10)

```

BFS ([Vertex(name=Брест), Vertex(name=Калининград), Vertex(name=СПб),
Vertex(name=Мурманск), Vertex(name=Минск), Vertex(name=Ярославль),
Vertex(name=Воронеж), Vertex(name=Волгоград), Vertex(name=Житомир),
Vertex(name=Донецк), Vertex(name=Орел), Vertex(name=Москва), Vertex(name=Казань)],
12)

DLS ([Vertex(name=Брест), Vertex(name=Вильнюс), Vertex(name=Витебск),
Vertex(name=Ниж.Новгород), Vertex(name=Москва), Vertex(name=Казань)], 5)

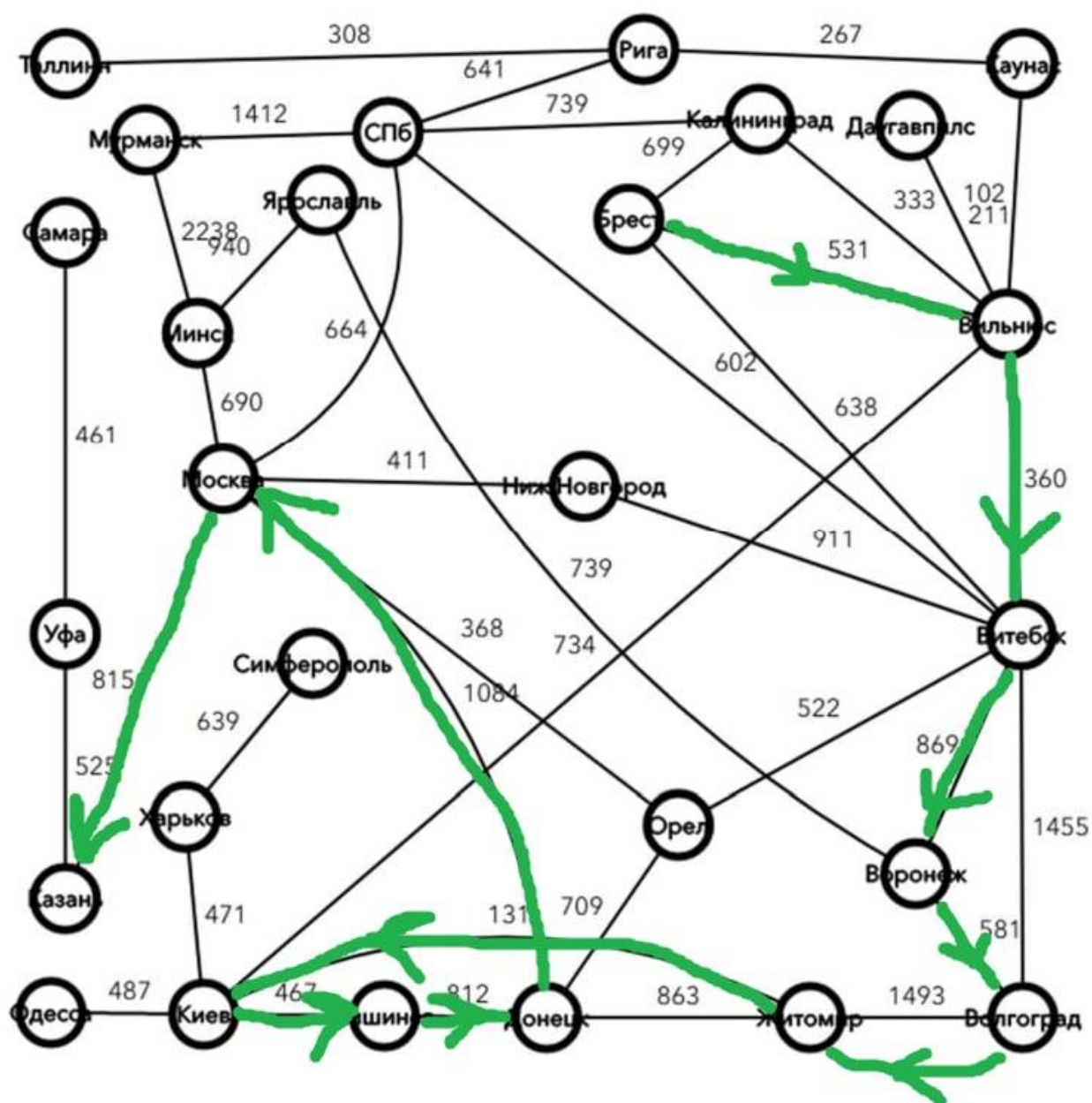
IDDFS ([Vertex(name=Брест), Vertex(name=Витебск), Vertex(name=Ниж.Новгород),
Vertex(name=Москва), Vertex(name=Казань)], 4)

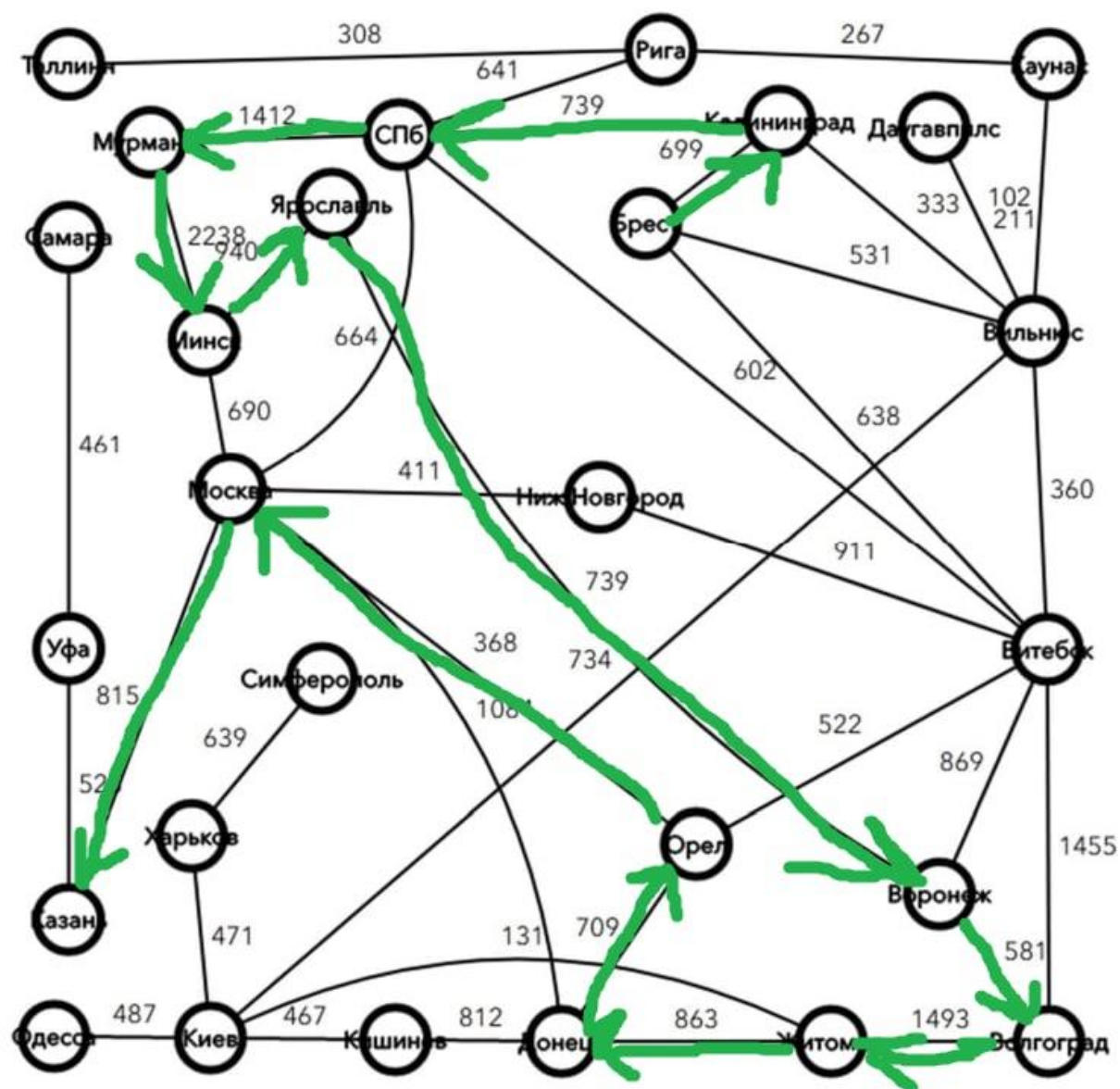
BDS ([Vertex(name=Брест), Vertex(name=Витебск), Vertex(name=Ниж.Новгород),
Vertex(name=Москва), Vertex(name=Казань)], 4)

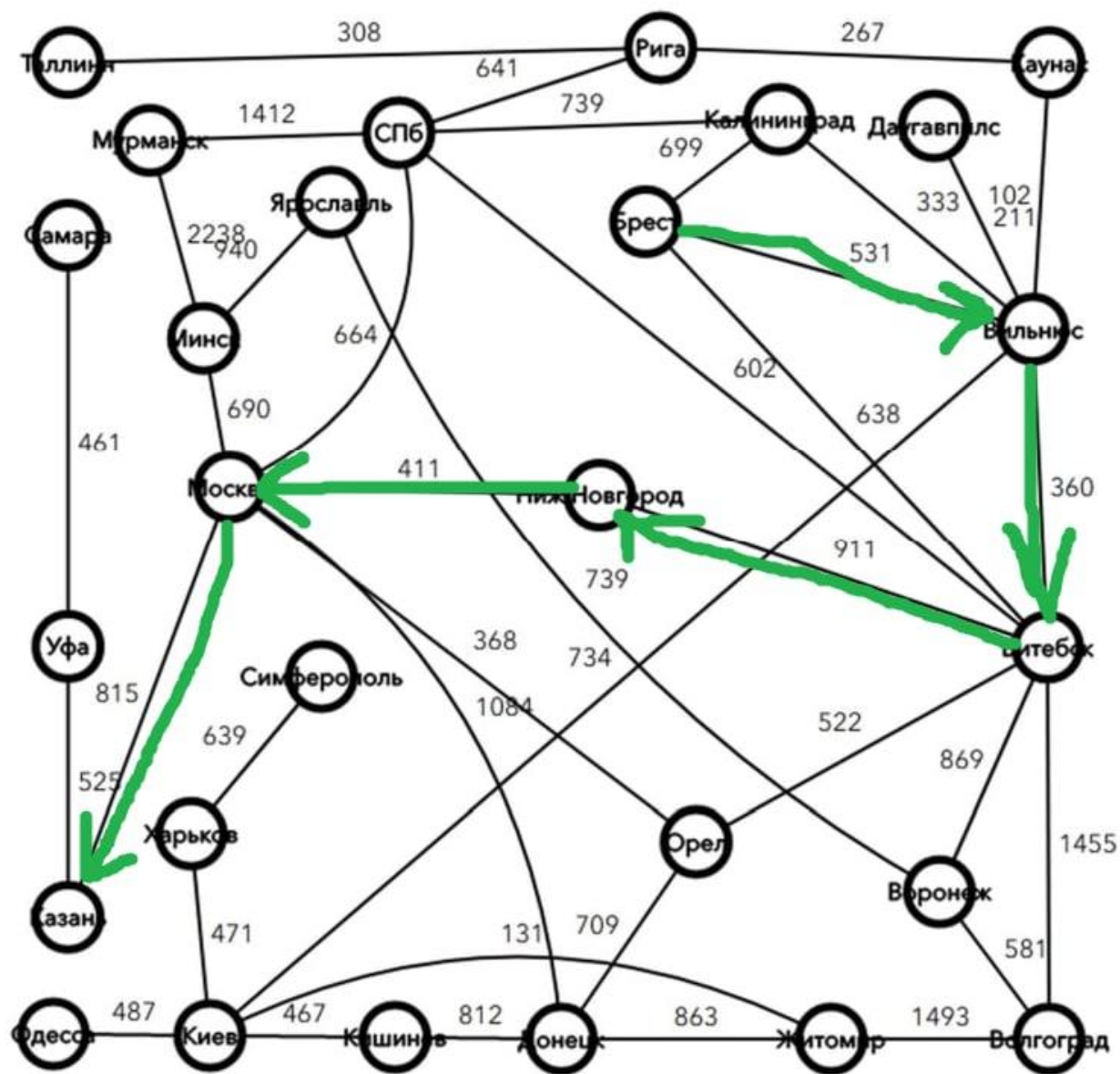
BFS ([Vertex(name=Брест), Vertex(name=Витебск), Vertex(name=Ниж.Новгород),
Vertex(name=Москва), Vertex(name=Казань)], 2775)

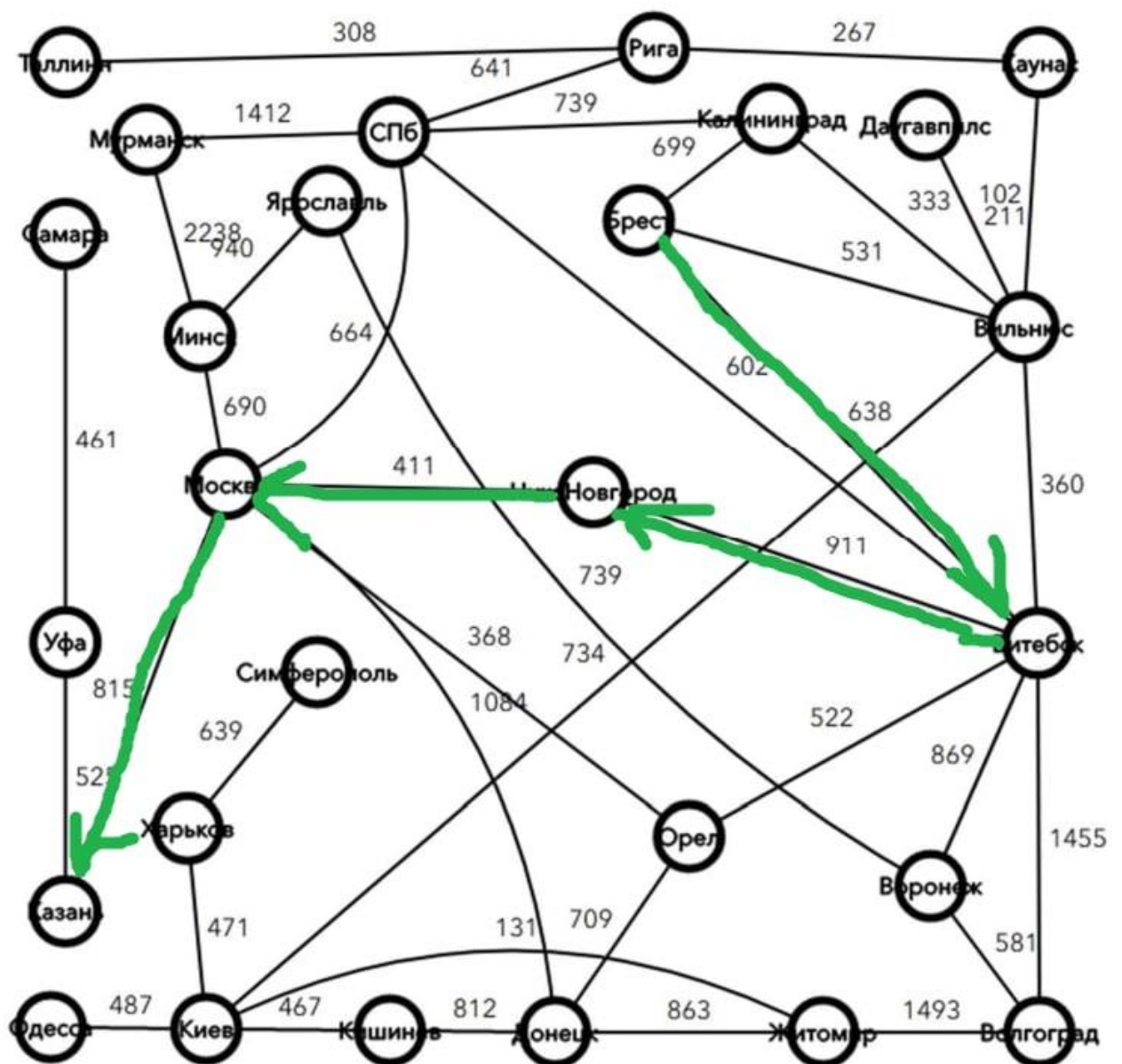
A* ([Vertex(name=Брест), Vertex(name=Витебск), Vertex(name=Орел),
Vertex(name=Москва), Vertex(name=Казань)], 2343)

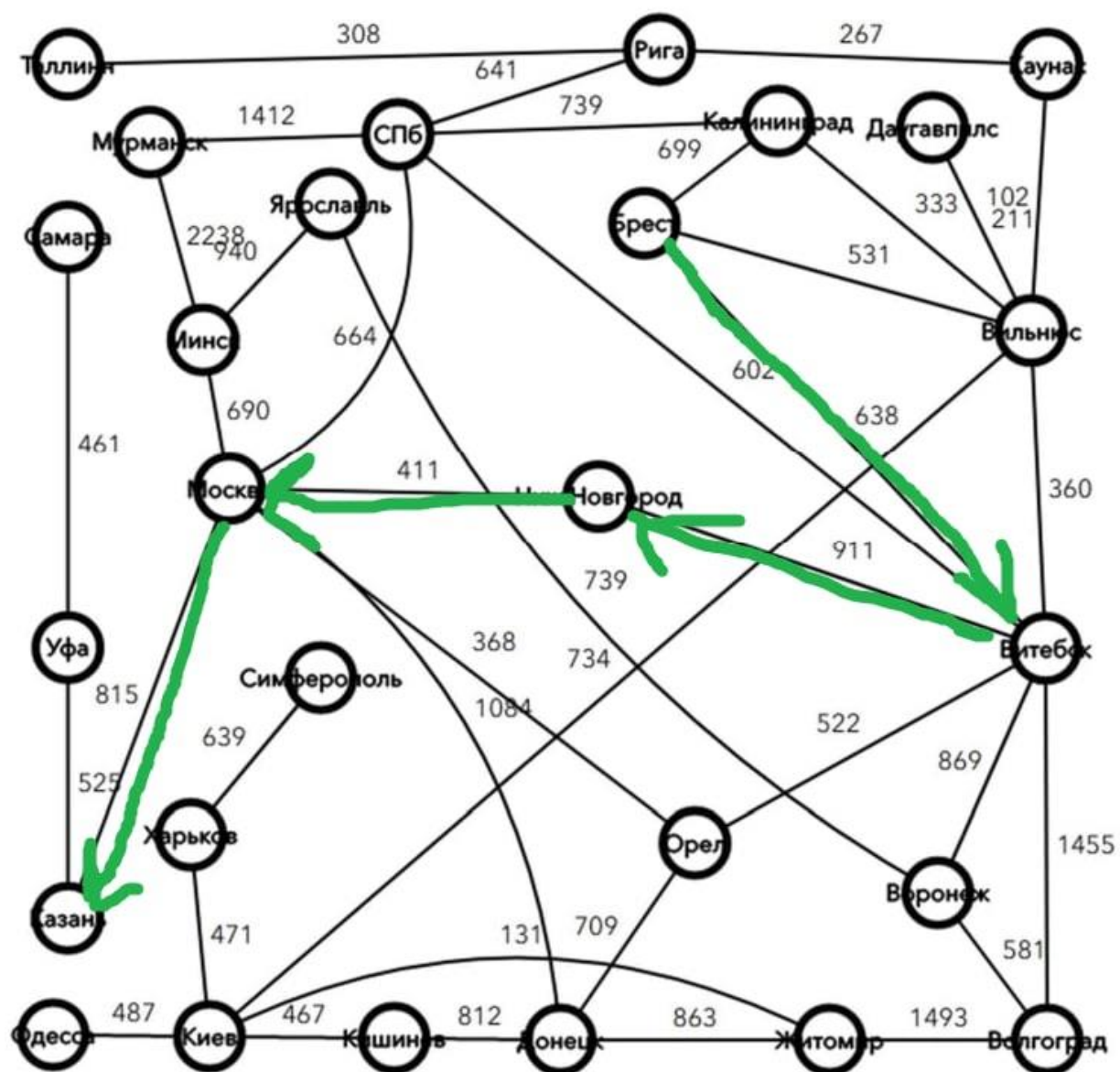
Изобразим на графе, где вершины – это города, а ребра – это пути между городами (с весом ребра = расстоянию между городами, найденные алгоритмами расстояния:

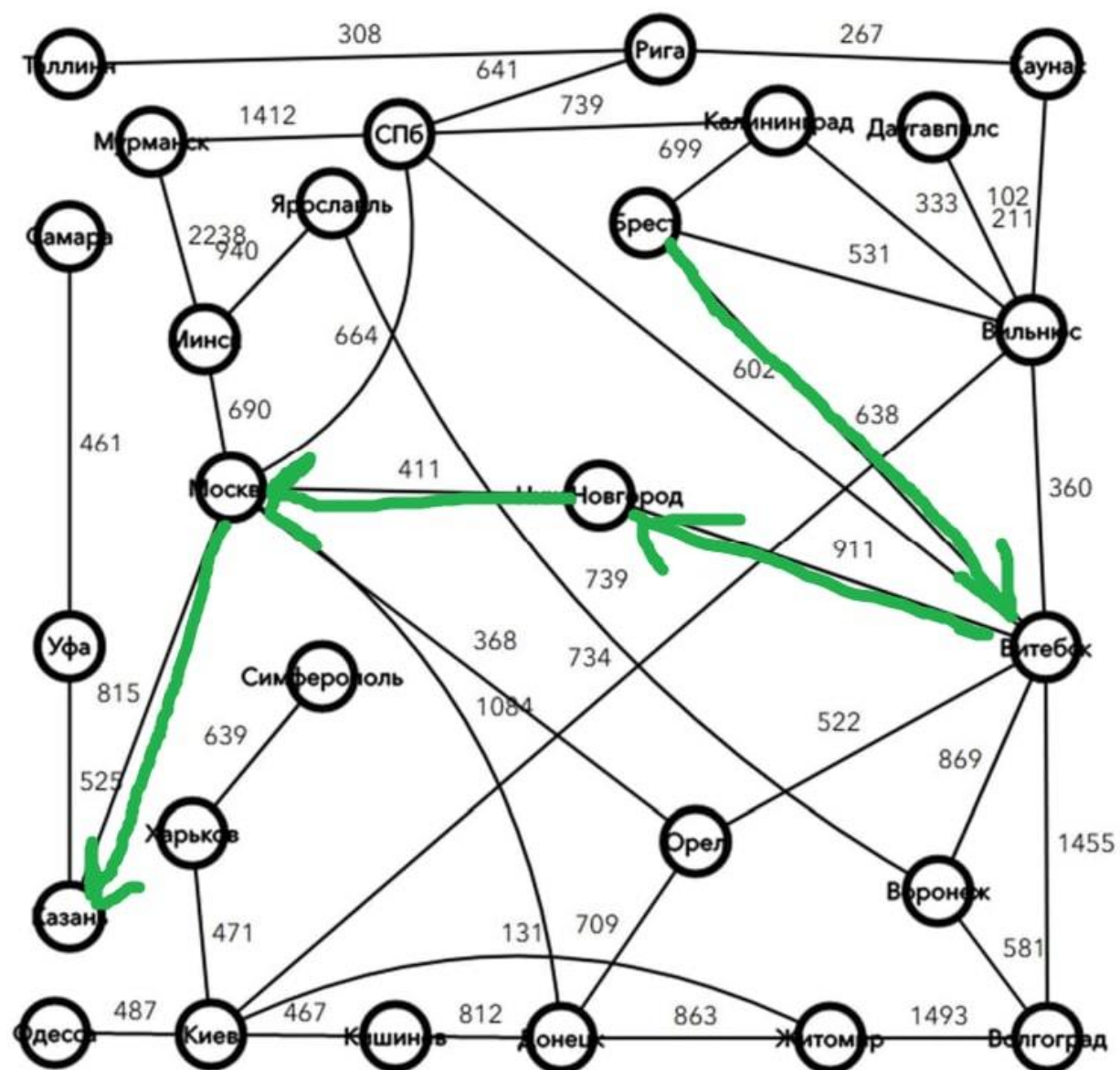


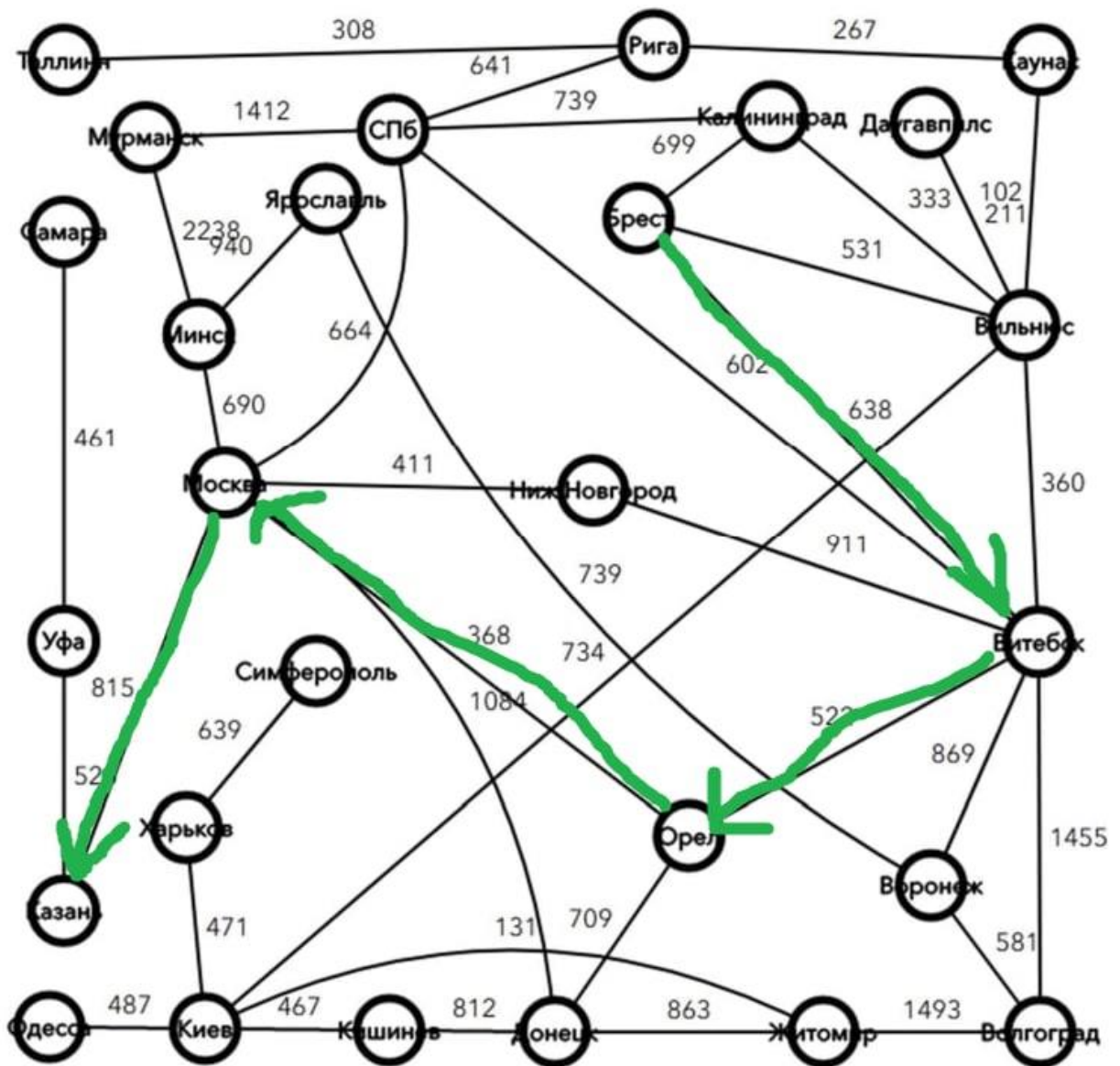












DFS. Т. к. этот алгоритм обладает информацией только о связях между вершинами, его дерево решений будет самым объёмным, ибо следующий рассматриваемый узел будет браться случайным образом, следовательно путь от начала до него может быть не оптимальным (в виду отсутствия эвристической оценки, отображать дерево решений смысла нет).

BFS. Отличается от DFS порядком обхода вершин, но также не принимает решений, основанный на оценки пройденного пути до вершины. Быстрее обнаружит решение на неглубоких уровнях, но также может пойти по неоптимальному пути и выдать это за решение.

DLS поступит умнее. Как только будет достигнута ограничительная глубина, цепочка поиска будет разорвана и будет предпринята попытка найти другой путь. Подобным образом отработает и

IDDFS, но сначала запустится с ограничением 1, 2, 3, а на 4-ый найдет то же самое решение.

BDS. Один поиск запускается с начального узла, другой – с конечного. Основан на BFS, но как только в оба BFS-а находят общий узел, поиск сворачивается.

BFS. На каждой итерации алгоритм будет выбирать ту вершину из соседей текущей, которая максимально приблизит нас к конечной вершине (максимальная эвристика).

A* отличается от BFS функцией оценки. Теперь это сумма эвристики и пути от стартовой вершины (в реальных приложениях используют теорему Пифагора, а не сумму, а сам алгоритм для определения пути движения противников к игроку).

Вывод: по ходу выполнения лабораторной работы, было реализовано несколько алгоритмов поиска маршрута по графу, описан каждый из них, и найденный путь изображен на графе цветом. Исходя из описания алгоритмов, все их можно сравнить между собой.