Задание 6. Алгоритмы на графах. Алгоритмы поиска пути на взвешенных графах
Варвара Кошман, С4113, 02.12.2019

Использование алгоритмов поиска пути на взвешенных графах (алгоритм Дейкстры, Беллмана-Форда, A*).
I. a) На случайном неориентированном взвешенном графе (100 вершин, 500 ребер с положительными весами) найти кратчайшие пути между начальной вершиной и другими.
(алгоритмы находят все кратчайшие пути, возвращая 1 кратчайший путь до случайно выбранной вершины)

Вывод консоли:
11:23:32.720 [main] INFO algorithms2019.Graph - minimum distance from 25 to 75 with DA is 40

shortest path between 25 and 75 by DA is [25, 85, 83, 76, 75]

11:23:33.618 [main] INFO algorithms2019.Graph - minimum distance from 25 to 75 with BFA is 40

shortest path between 25 and 75 by BFA is [25, 85, 83, 76, 75]

b) Для одних и тех же начальных данных сделать замеры работы алгоритмов и найти среднее по 10 экспериметам время.

Вывод консоли:

Warmup: 5 iterations, 1 s each
Measurement: 10 iterations, 1 s each
Timeout: 10 min per iteration
Threads: 1 thread, will synchronize iterations
Benchmark mode: Average time, time/op
Benchmark: algorithms2019.Graph.bellmanFordAlgorithm

Run progress: 0.00% complete, ETA 00:00:30
Fork: 1 of 1
Warmup Iteration   1: 891.040 us/op
Warmup Iteration   2: 589.878 us/op
Warmup Iteration   3: 631.533 us/op
Warmup Iteration   4: 531.120 us/op
Warmup Iteration   5: 511.385 us/op

Iteration   1: 567.873 us/op
Iteration   2: 581.189 us/op
Iteration   3: 666.557 us/op
Iteration   4: 628.852 us/op
Iteration   5: 530.908 us/op
Iteration   6: 625.065 us/op
Iteration   7: 560.819 us/op
Iteration   8: 579.957 us/op
Iteration   9: 565.077 us/op
Iteration 10: 554.653 us/op

Result "algorithms2019.Graph.bellmanFordAlgorithm":
586.095 ±(99.9%) 62.395 us/op [Average]
(min, avg, max) = (530.908, 586.095, 666.557), stdev = 41.271

CI (99.9%): [523.700, 648.491] (assumes normal distribution)

Warmup: 5 iterations, 1 s each
Measurement: 10 iterations, 1 s each
Timeout: 10 min per iteration
Threads: 1 thread, will synchronize iterations
Benchmark mode: Average time, time/op
Benchmark: algorithms2019.Graph.dijkstraAlgorithm
Run progress: 50.00% complete, ETA 00:00:16
Fork: 1 of 1

Warmup Iteration 1: 231.805 us/op
Warmup Iteration 2: 203.630 us/op
Warmup Iteration 3: 190.725 us/op
Warmup Iteration 4: 188.065 us/op
Warmup Iteration 5: 189.198 us/op

Iteration 1: 192.935 us/op
Iteration 2: 190.770 us/op
Iteration 3: 198.305 us/op
Iteration 4: 187.995 us/op
Iteration 5: 197.174 us/op
Iteration 6: 178.840 us/op
Iteration 7: 188.333 us/op
Iteration 8: 193.509 us/op
Iteration 9: 189.590 us/op
Iteration 10: 198.332 us/op

Result "algorithms2019.Graph.dijkstraAlgorithm":
191.578 ±(99.9%) 8.986 us/op [Average]
(min, avg, max) = (178.840, 191.578, 198.332), stdev = 5.944
CI (99.9%): [182.592, 200.565] (assumes normal distribution)

Run complete. Total time: 00:00:32

Benchmark	Mode	Cnt	Score	Error	Units
Graph.bellmanFordAlgorithm	avgt	10	586.095	± 62.395	us/op
Graph.dijkstraAlgorithm	avgt	10	191.578	± 8.986	us/op

shortest path between 33 and 11 by DA is [33, 58, 84, 11]
shortest path between 33 and 11 by BFA is [33, 58, 84, 11]

Выводы:

По результатам замеров видно, что на одних и тех же вершинах алгоритм Дейкстры(191.578 мкс) работает быстрее Беллмана-Форда(586.095 мкс) в среднем в 3 раза. Асимптотика для алгоритма Дейкстры в этой реализации $O(n^2)$ (тк для каждой из n вершин ищется минимум по расстояниям (требуется просмотреть n значений) + релаксация по ребрам, порпорциональная числу ребер => $O(n^2 + m)$ => $O(n^2)$, тк $m \leq n(n-1)$), асимптотика для Беллмана-Форда - $O(n \times m)$ (тк все ребра (m) просматриваются n число раз).
Так как в нашем примере число ребер в 5 раз больше вершин, такие результаты понятны (например, если ради эксперимента установить $n = m$, то Беллман-Форд работает быстрее по тем же замерам - 208.817 мкс по сравнению с Дейкстрой с 284.024 мкс, где разница в результате, очевидно, за счет прибавляемой релаксации ребер в асимптотике Дейкстры, а если поставить худший случай для Беллмана-Форда, те когда граф плотный и $m = n(n-1)/2$), то сложность для него вырождается в

кубическую и путь находится за 9235.033 мкс, тогда как Дейкстра за 557.267 мкс). Судя по цифрам для крайних случаев, те цифры, что получены для примера с 500 ребрами, можно считать в принципе близкими.

Код (без конфигурационных подробностей):

```
// NB logging should be commented when benchmarking

@Getter
@Slf4j
@BenchmarkMode(Mode.AverageTime)
@Fork(value = 1)
@Warmup(iterations = 5)
@Measurement(iterations = 10)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
public class Graph {
    private static final int UPPER_BOUND = 100; // upper bound for weights in a graph
    public static final int N_VERTICES = 100;
    public static final int N_EDGES = 500;
    public static final Integer source = new Random().nextInt(N_VERTICES);
    public static final Integer target = new Random().nextInt(N_VERTICES);
    public static final Integer[][] adjMatrix = genRandomAdjMatrix(N_VERTICES, N_EDGES);

    @Benchmark
    public static List<Integer> dijkstraAlgorithm() {
        if (source.equals(target)) {
            return Collections.emptyList();
        }
        List<Integer> sptSet = new ArrayList<>();
        List<Integer> notSptSet = new ArrayList<>();
        IntStream.range(0, adjMatrix.length).forEach(notSptSet::add);
        List<Integer> predecessors = new
ArrayList<>(Collections.nCopies(adjMatrix.length, -1));
        List<Integer> distances = new ArrayList<>(Collections.nCopies(adjMatrix.length,
Integer.MAX_VALUE / 2));
        distances.set(source, 0); //set dist(source, source)=0
        while (sptSet.size() != adjMatrix.length) {
            Integer newVertex = notSptSet.stream()
                    .min(Comparator.comparingInt(distances::get)).get(); //find new
vertex from notSptSet, so that distance from source to it is min
            sptSet.add(newVertex);
            notSptSet.remove(newVertex);
            List<Integer> neighbours = getNeighbours(newVertex);
            neighbours.forEach(neighbour -> {
                if (distances.get(newVertex) + adjMatrix[newVertex][neighbour] <
distances.get(neighbour)) {
                    distances.set(neighbour, distances.get(newVertex) +
adjMatrix[newVertex][neighbour]);
                    predecessors.set(neighbour, newVertex);
                }
            });
        }
        log.info(String.format("minimum distance from %d to %d with DA is %d", source,
target, distances.get(target)));
        return unrollPath(predecessors, target);
```

```
    }

    private static List<Integer> unrollPath(List<Integer> predecessors, Integer target)
{
        List<Integer> path = new ArrayList<>();
        Integer current = target;
        path.add(current);
        while (predecessors.get(current) != -1) {
            Integer pred = predecessors.get(current);
            path.add(pred);
            current = pred;
        }
        Collections.reverse(path);
        return path;
    }

    private static List<Integer> getNeighbours(Integer vertex) {
        return IntStream.range(0, adjMatrix.length)
                .filter(otherVertex -> adjMatrix[vertex][otherVertex] != 0)
                .boxed()
                .collect(Collectors.toList());
    }

    @Benchmark
    public static List<Integer> bellmanFordAlgorithm() {
        List<Integer> distances = new ArrayList<>(Collections.nCopies(adjMatrix.length,
Integer.MAX_VALUE / 2));
        List<Integer> predecessors = new
ArrayList<>(Collections.nCopies(adjMatrix.length, -1));
        distances.set(source, 0); //set dist(source, source)=0
        List<Edge> allEdges = getAllEdges();
        for (int i = 0; i < adjMatrix.length - 1; i++) {
            for (Edge edge : allEdges) {
                if (distances.get(edge.getFrom()) + edge.getWeight() <
distances.get(edge.getTo())) {
                    distances.set(edge.getTo(), distances.get(edge.getFrom()) +
edge.getWeight());
                    predecessors.set(edge.getTo(), edge.getFrom());
                }
                // as graph is undirected, so check paths an edge in both directions
                if (distances.get(edge.getTo()) + edge.getWeight() <
distances.get(edge.getFrom())) {
                    distances.set(edge.getFrom(), distances.get(edge.getTo()) +
edge.getWeight());
                    predecessors.set(edge.getFrom(), edge.getTo());
                }
            }
        }
        for (Edge edge : allEdges) { //the last walk through to find negative cycle
            if (distances.get(edge.getFrom()) + edge.getWeight() <
distances.get(edge.getTo())) {
                log.warn("Negative weight cycle found");
            }
        }
        log.info(String.format("minimum distance from %d to %d with BFA is %d", source,
target, distances.get(target)));
        return unrollPath(predecessors, target);
```

```
    }


    public static List<Edge> getAllEdges() {
        List<Edge> edges = new ArrayList<>();
        for (int i = 0; i < adjMatrix.length - 1; i++) {
            for (int j = i + 1; j < adjMatrix.length; j++) {
                if (adjMatrix[i][j] != 0) {
                    edges.add(new Edge(i, j, adjMatrix[i][j]));
                }
            }
        }
        return edges;
    }

    /*
       as graph in task is undirected, its adjacency matrix is symmetric
       and as it doesn't have any loops, its matrix has zeros on the main diagonal
   */
    public static Integer[][] genRandomAdjMatrix(int nVertices, int nEdges) {
        Integer[][] adjMatrix = new Integer[nVertices][nVertices];
        List<Integer> allCells = new ArrayList<>(Collections.nCopies(nVertices *
nVertices / 2 - (nVertices / 2), 0));
        Random rnd = new Random();
        for (int i = 0; i < nEdges; i++) {
            allCells.set(i, rnd.nextInt(UPPER_BOUND));
        }
        for (int i = allCells.size() - 1; i > 0; i--) { // generate a random permutation
            Collections.swap(allCells, i, rnd.nextInt(i));
        }
        int currentIndex = 0;     // fill the adj adjMatrix with random permutation
        outer:
        for (int i = 0; i < nVertices - 1; i++) {
            adjMatrix[i][i] = 0;
            for (int j = i + 1; j < nVertices; j++) {
                if (i != j) {
                    adjMatrix[i][j] = allCells.get(currentIndex);
                    adjMatrix[j][i] = allCells.get(currentIndex);
                    currentIndex++;
                }
                if (currentIndex == allCells.size()) {
                    break outer;
                }
            }
        }
        adjMatrix[nVertices - 1][nVertices - 1] = 0;
        return adjMatrix;
    }
}
```

***Вспомогательный код*:**

Класс Ребро:

```
@AllArgsConstructor
@Getter
```

```java
public class Edge {
    private Integer from;
    private Integer to;
    private Integer weight;
}
```

II. Найти кратчайший путь между двумя случайными ячейками на сетке 10×10 с 30 препятствиями с помощью алгоритма A*.

Вывод консоли:

```
# Warmup: 5 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: algorithms2019.lab6.part2.GraphII.aStarAlgorithm
# Run progress: 0.00% complete, ETA 00:00:15
# Fork: 1 of 1
# Warmup Iteration   1: 54.750 us/op
# Warmup Iteration   2: 40.480 us/op
# Warmup Iteration   3: 37.030 us/op
# Warmup Iteration   4: 33.264 us/op
# Warmup Iteration   5: 34.137 us/op
Iteration   1: 34.156 us/op
Iteration   2: 32.002 us/op
Iteration   3: 39.618 us/op
Iteration   4: 35.972 us/op
Iteration   5: 33.886 us/op
Iteration   6: 38.584 us/op
Iteration   7: 38.531 us/op
Iteration   8: 39.868 us/op
Iteration   9: 41.360 us/op
Iteration  10: 38.822 us/op

Result "algorithms2019.lab6.part2.GraphII.aStarAlgorithm":
37.280 ±(99.9%) 4.654 us/op [Average]
 (min, avg, max) = (32.002, 37.280, 41.360), stdev = 3.078
CI (99.9%): [32.626, 41.934] (assumes normal distribution)

# Run complete. Total time: 00:00:16
Benchmark                  Mode  Cnt  Score   Error  Units
GraphII.aStarAlgorithm     avgt   10  37.280 ± 4.654  us/op
```

```
shortest path between (1, 0) and (9, 5) by A* is [(1, 0), (2, 1), (3, 2), (4, 2), (5, 3), (6, 4), (7, 5), (8, 5), (9, 5)]
|    |null|    |null|    |    |null|    |    |    |
|  * |    |    |    |null|null|    |null|null|    |
|    |  * |    |    |null|    |    |    |null|null|
|    |null|  * |null|    |    |null|    |null|null|
|null|    |  * |null|    |    |null|    |    |    |
|    |    |null|  * |    |null|    |    |null|null|
|    |    |    |    |  * |    |    |    |    |null|
|null|    |null|    |    |  * |    |    |    |null|
|    |    |    |    |    |  * |null|    |    |    |
|    |    |    |    |null|  * |null|    |    |null|
```

На другой случайной паре вершин:

```
# Warmup: 5 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: algorithms2019.lab6.part2.GraphII.aStarAlgorithm

# Run progress: 0.00% complete, ETA 00:00:15
# Fork: 1 of 1
# Warmup Iteration   1: 114.495 us/op
# Warmup Iteration   2: 82.534 us/op
# Warmup Iteration   3: 96.416 us/op
# Warmup Iteration   4: 68.997 us/op
# Warmup Iteration   5: 67.655 us/op

Iteration   1: 65.368 us/op
Iteration   2: 70.786 us/op
Iteration   3: 70.009 us/op
Iteration   4: 78.884 us/op
Iteration   5: 67.160 us/op
Iteration   6: 67.285 us/op
Iteration   7: 71.837 us/op
Iteration   8: 71.087 us/op
Iteration   9: 70.161 us/op
Iteration  10: 67.720 us/op

Result "algorithms2019.lab6.part2.GraphII.aStarAlgorithm":
70.030 ±(99.9%) 5.662 us/op [Average]
(min, avg, max) = (65.368, 70.030, 78.884), stdev = 3.745
CI (99.9%): [64.368, 75.691] (assumes normal distribution)

# Run complete. Total time: 00:00:16
Benchmark                  Mode  Cnt   Score   Error   Units
GraphII.aStarAlgorithm     avgt    10   70.030 ± 5.662  us/op
```

```
shortest path between (2, 8) and (7, 2) by A* is [(2, 8), (3, 7), (3, 6), (4, 5), (4, 4), (5, 3), (6, 4), (7, 3), (7, 2)]
|    |    |    |null|null|null|    |null|    |null|
|    |null|    |    |    |    |null|    |    |    |
|    |    |    |    |    |    |    |null|  * |    |
|null|null|    |null|null|null|  * |  * |null|null|
|    |    |null|null|  * |  * |null|    |    |null|
|null|    |null|  * |null|    |    |    |null|    |
|    |    |null|null|  * |    |    |    |null|    |
|null|    |  * |  * |null|    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |
|    |null|    |    |    |    |null|    |    |    |
```

И еще на другой:

Benchmark	Mode	Cnt	Score	Error	Units
GraphII.aStarAlgorithm	avgt	10	14.618	± 2.756	us/op

```
shortest path between (2, 6) and (8, 2) by A* is [(2, 6), (3, 6), (4, 5), (5, 4), (6, 3), (7, 2), (8, 2)]
|    |    |    |null|null|null|    |    |    |null|
|    |    |    |    |    |    |null|    |    |    |
|null|    |    |    |    |    |  * |null|    |    |
|null|null|null|    |    |null|  * |    |null|    |
|    |    |    |null|    |  * |null|    |null|    |
|null|    |null|    |  * |    |    |null|null|    |
|    |    |null|  * |null|null|    |null|null|    |
|    |    |  * |    |    |    |    |    |    |    |
|null|    |  * |null|    |null|null|    |    |    |
|    |    |    |    |    |    |    |    |null|null|
```

Выводы:

Полученные замерами 37.280 мкс, 70.030 мкс, 14.618 мкс для разных пар вершин (при теоретической сложности O(m)) на порядок меньше результатов, полученных алгоритмами Дейкстры и Беллмана-Форда. Это показывает эффективность применения эвристической оценки для принципа алгоритма Дейкстры. Хотя и очевидно, что этот алгоритм требует больше памяти – нужно хранить все сгенерированные вершины.

Код (без конфигурационных подробностей)**:**

```java
@Slf4j
@BenchmarkMode(Mode.AverageTime)
@Fork(value = 1)
@Warmup(iterations = 5)
@Measurement(iterations = 10)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
public class GraphII {
    private static final int N_CELLS = 10;
    private static final int N_BARRIERS = 30;
    private static final Random rnd = new Random();

    public static final Integer[][] adjMatrix = genRandomGrid();
    public static final List<Vertex> allVertices = getAllVertices();
```

```
    public static final Vertex source = pickRandomVertex();
    public static final Vertex target = pickRandomVertex();

    @Benchmark
    public static Optional<List<Vertex>> aStarAlgorithm() {
        Map<Vertex, Vertex> predecessors = new HashMap<>();
        allVertices.forEach(vertex -> predecessors.put(vertex, null));
        List<Vertex> sptSet = new ArrayList<>();// visited
        List<Vertex> notSptSet = new ArrayList<>(); //to be visited
        notSptSet.add(source);
        source.setG(0); //set dist(source, source)=0
        source.setF(source.getG() + h(source));
        while (!notSptSet.isEmpty()) {
            Vertex current = getNextVertex(notSptSet);
            if (target.equals(current)) {
                return Optional.of(unrollPath(predecessors));
            }
            notSptSet.remove(current);
            sptSet.add(current);
            List<Vertex> successors = getSuccessors(current);
            for (Vertex v : successors) {
                int tentativeScore = current.getG() + v.getCost();
                if (!sptSet.contains(v) || tentativeScore < v.getG()) {
                    predecessors.put(v, current);
                    v.setG(tentativeScore);
                    v.setF(v.getG() + h(v));
                    if (!notSptSet.contains(v)) {
                        notSptSet.add(v);
                    }
                }
            }
        }
        return Optional.empty();
    }

/*
Euclidean distance is an approximation heuristic
 */
    private static double h(Vertex vertex) {
        return Math.sqrt(Math.pow(vertex.getRow() - target.getRow(), 2)
                + Math.pow(vertex.getColumn() - target.getColumn(), 2));
    }

/*
assuming we can move in 8 directions (we're going to use Euclidean distance, if
Manhattan was chosen,
then it would be only 4), this function returns a list of possible Vertex-s to go next
excluding barriers.
 */
    private static List<Vertex> getSuccessors(Vertex vertex) {
        Set<Vertex> neighbours = new HashSet<>();
        int row = vertex.getRow();
        int col = vertex.getColumn();
        if (row == N_CELLS - 2 || col == N_CELLS - 2 || row == 0 || col == 0) {
            if (row + 1 <= N_CELLS - 1) {
                getVertexByij(row + 1, col).ifPresent(neighbours::add);
                if (col + 1 <= N_CELLS - 1) {
```

```
                    getVertexByij(row + 1, col + 1).ifPresent(neighbours::add);
                    getVertexByij(row, col + 1).ifPresent(neighbours::add);
                }
                if (col - 1 >= 0) {
                    getVertexByij(row + 1, col - 1).ifPresent(neighbours::add);
                    getVertexByij(row, col - 1).ifPresent(neighbours::add);
                }
            }
            if (row - 1 >= 0) {
                getVertexByij(row - 1, col).ifPresent(neighbours::add);
                if (col + 1 <= N_CELLS - 1) {
                    getVertexByij(row - 1, col + 1).ifPresent(neighbours::add);
                    getVertexByij(row, col + 1).ifPresent(neighbours::add);
                }
                if (col - 1 >= 0) {
                    getVertexByij(row - 1, col - 1).ifPresent(neighbours::add);
                    getVertexByij(row, col - 1).ifPresent(neighbours::add);
                }
            }
        } else {
            getVertexByij(row, col + 1).ifPresent(neighbours::add);
            getVertexByij(row, col - 1).ifPresent(neighbours::add);
            getVertexByij(row + 1, col).ifPresent(neighbours::add);
            getVertexByij(row - 1, col).ifPresent(neighbours::add);
            getVertexByij(row + 1, col + 1).ifPresent(neighbours::add);
            getVertexByij(row - 1, col + 1).ifPresent(neighbours::add);
            getVertexByij(row - 1, col - 1).ifPresent(neighbours::add);
            getVertexByij(row + 1, col - 1).ifPresent(neighbours::add);
        }
        return new ArrayList<>(neighbours);
    }
/*
get a reversed list of found path
 */
    private static List<Vertex> unrollPath(Map<Vertex, Vertex> predecessors) {
        List<Vertex> path = new ArrayList<>();
        Vertex current = target;
        path.add(current);
        while (predecessors.get(current) != null) {
            Vertex pred = predecessors.get(current);
            path.add(pred);
            current = pred;
        }
        Collections.reverse(path);
        return path;
    }

/*
choose next vertex from not visited yet with the smallest value of heuristic f
 */
    private static Vertex getNextVertex(List<Vertex> notSptSet) {
        return notSptSet.stream()
                .min(Comparator.comparingDouble(Vertex::getF))
                .get();
    }

/*
```

```
get a Vertex object from list of all vertices by square's position (i,j)
 */
    private static Optional<Vertex> getVertexByij(int row, int col) {
        return allVertices.stream()
                .filter(v -> v.getRow() == row && v.getColumn() == col)
                .filter(v -> Objects.nonNull(v.getCost()))
                .findFirst();
    }

/*
function that picks random available vertex, which is not barrier
 */
    private static Vertex pickRandomVertex() {
        List<Vertex> vertices = allVertices.stream()
                .filter(vertex -> Objects.nonNull(vertex.getCost()))
                .collect(Collectors.toList());
        return vertices.get(rnd.nextInt(vertices.size()));
    }

/*
function to get a list of all existing vertices
 */
    private static List<Vertex> getAllVertices() {
        List<Vertex> allVertices = new ArrayList<>();
        for (int i = 0; i < N_CELLS; i++) {
            for (int j = 0; j < N_CELLS; j++) {
                allVertices.add(new Vertex(i, j, adjMatrix[i][j]));
            }
        }
        return allVertices;
    }

/*
function that generates a grid of a given size and fill it with ones and nulls(with are
considered as barriers)
 */
    private static Integer[][] genRandomGrid() {
        Integer[][] adjMatrix = new Integer[N_CELLS][N_CELLS];
        List<Integer> allCells = new ArrayList<>(Collections.nCopies(N_CELLS * N_CELLS,
1));
        for (int i = 0; i < N_BARRIERS; i++) {
            allCells.set(i, null); // fill with barriers (barrier here is null-cell)
        }
        for (int i = allCells.size() - 1; i > 0; i--) { // generate a random permutation
            Collections.swap(allCells, i, rnd.nextInt(i));
        }
        int currentIndex = 0;
        for (int i = 0; i < N_CELLS; i++) {
            for (int j = 0; j < N_CELLS; j++) {
                adjMatrix[i][j] = allCells.get(currentIndex);
                currentIndex++;
            }
        }
        return adjMatrix;
    }
/*
function for printing out found path visually understandable
```

```
 */
    public static void printGrid(List<Vertex> path){
        for (int i = 0; i < N_CELLS; i++) {
            System.out.print("|");
            for (int j = 0; j < N_CELLS; j++) {
                if (adjMatrix[i][j] == null){
                    System.out.printf("%s|", adjMatrix[i][j]);
                }else if (path.contains(getVertexByij(i, j).get())){
                    System.out.print("  * |");
                }else{
                    System.out.print("    |");
                }
            }
            System.out.println();
        }
    }
}
```

***Вспомогательный код*:**

Класс Вершина:

```
@Getter
public class Vertex {
    private int row; //horizontal position on grid
    private int column; //vertical position on grid
    private Integer cost; // cost to reach the vertex

    @Setter
    private int g; //distance form source to this vertex
    @Setter
    private double f; // heuristic value of function "distance + cost"

    public Vertex (int row, int column, Integer cost){
        this.row = row;
        this.column = column;
        this.cost = cost;
    }
}
```

Главный класс:

```
public class Main {
       @SneakyThrows
       public static void main(String[] args) {
           org.openjdk.jmh.Main.main(args);
           List<Integer> shortestPathDA = Graph.dijkstraAlgorithm();
           System.out.printf("shortest path between %d and %d by DA is %s\n", source,
       target, shortestPathDA.toString());
           List<Edge> allEdges = Graph.getAllEdges();
           Visualiser.drawGraph(Graph.adjMatrix, allEdges);
           List<Integer> shortestPathBFA = Graph.bellmanFordAlgorithm();
           System.out.printf("shortest path between %d and %d by BFA is %s", source,
       target, shortestPathBFA.toString());
           Optional<List<Vertex>> shortestPathAstar = GraphII.aStarAlgorithm();
           shortestPathAstar.ifPresent(result -> {
               List<String> stringList = result.stream()
                       .map(vertex -> String.format("(%d, %d)", vertex.getRow(),
```

```
        vertex.getColumn()))
                        .collect(Collectors.toList());
                System.out.printf("shortest path between (%d, %d) and (%d, %d) by A* is
        %s\n",
                        GraphII.source.getRow(), GraphII.source.getColumn(),
                        GraphII.target.getRow(), GraphII.target.getColumn(),
        stringList.toString());
                GraphII.printGrid(result);
            });
}
```