1. **Цель работы:**

Изучить методы имитационного моделирования и ускоренного имитационного моделирования для решения задачи о определении вероятности связности двух вершин в случайном графе.

2. **Исходный граф:**

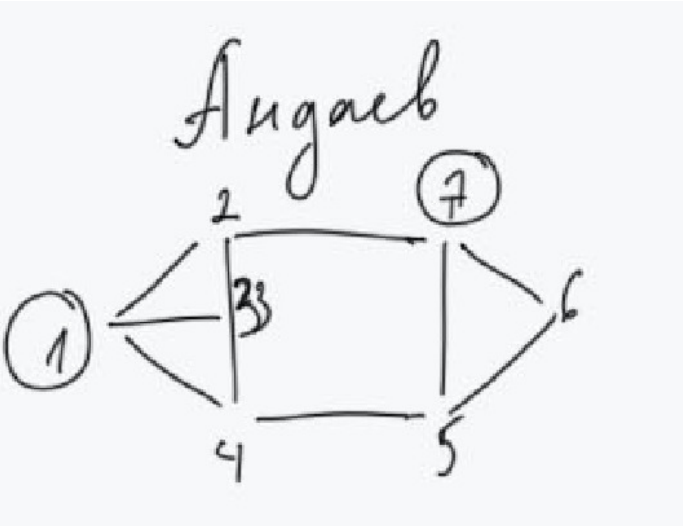


Рис. 1. Исходный случайный граф.

3. **Первый этап:**

В первом этапе лабораторной работы необходимо использовать метод имитационного моделирования для получения оценки вероятности пути между двумя вершинами, как функцию от вероятности ребра. Для этого воспользуемся алгоритмом моделирования (рис. 2).

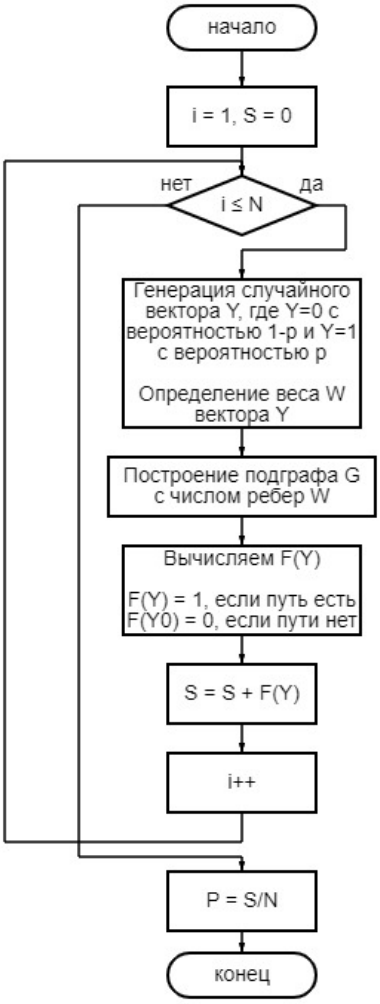


Рис. 2. Алгоритм имитационного моделирования.

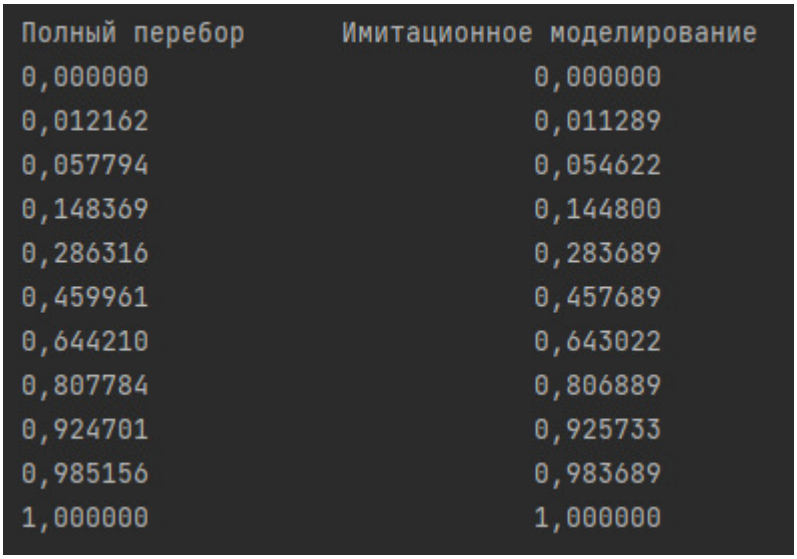
Для получения числа экспериментов N необходимо ввести такую переменную, как

доверительный интервал □. Зависимость переменной N от □: .

Рассмотрим результирующую оценку вероятности пути для □ = 0.1 и □ = 0.01 и сравним с результатами полного перебора.

Полный перебор	Имитационное моделирование
0,000000	0,000000
0,012162	0,022222
0,057794	0,088889
0,148369	0,133333
0,286316	0,288889
0,459961	0,444444
0,644210	0,626667
0,807784	0,831111
0,924701	0,964444
0,985156	0,991111
1,000000	1,000000

Рис. 3. Результат имитационного моделирования для □ = 0.1.



Полный перебор	Имитационное моделирование
0,000000	0,000000
0,012162	0,011289
0,057794	0,054622
0,148369	0,144800
0,286316	0,283689
0,459961	0,457689
0,644210	0,643022
0,807784	0,806889
0,924701	0,925733
0,985156	0,983689
1,000000	1,000000

Рис. 4. Результат имитационного моделирования для □ = 0.01.

Исходя из полученных данных, можно сделать вывод о том, что имитационное моделирование было проведено верно, так как оценка вероятности существования пути должна отличаться от вероятности полного перебора не более чем на ±□.

4. **Второй этап:**

Во втором этапе лабораторной работы необходимо использовать один из методов ускоренного имитационного моделирования для получения оценки вероятности пути между двумя вершинами, как функцию от вероятности ребра. Для этого воспользуемся алгоритмом ускорения за счет снижения временных затрат на проведение некоторых экспериментов (рис. 5).

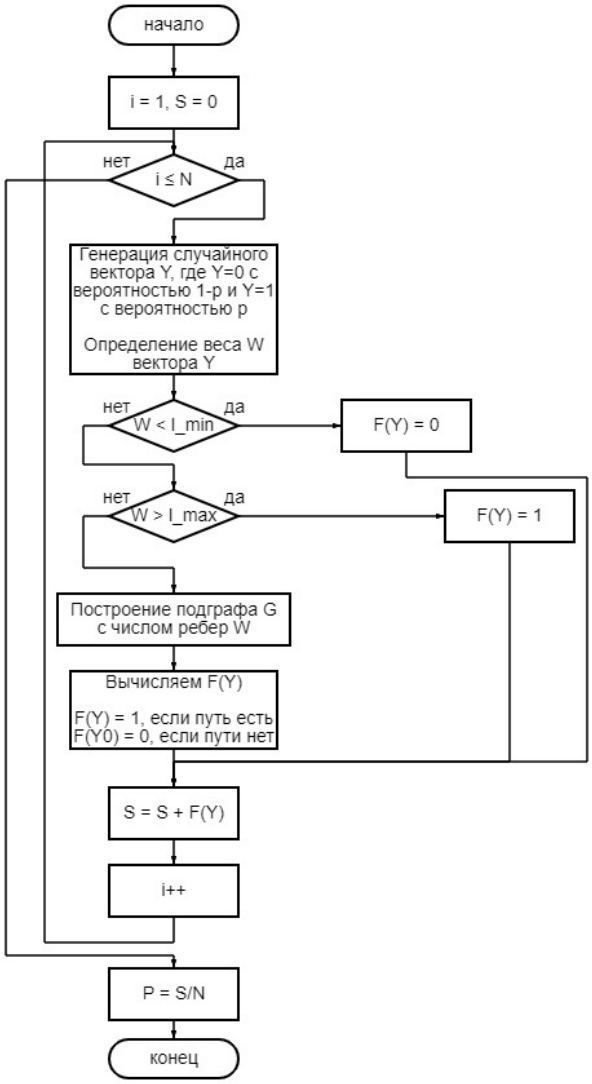


Рис. 5. Алгоритм ускоренного имитационного моделирования.

Для получения числа экспериментов N необходимо ввести такую переменную, как

доверительный интервал □. Зависимость переменной N от □: .

Для исходного графа , так как при количестве ребер < 2 подграф не будет иметь пути между двумя необходимыми вершинами, а , так как при числе ребер > 8 подграф всегда будет иметь путь между двумя рассматриваемыми вершинами.

Рассмотрим результирующую оценку вероятности пути для □ = 0.1 и □ = 0.01 и сравним с результатами полного перебора.

Полный перебор	Имитационное моделирование	Ускоренное имитационное моделирование
0,000000	0,000000	0,000000
0,012162	0,004444	0,000000
0,057794	0,053333	0,053333
0,148369	0,160000	0,151111
0,286316	0,342222	0,280000
0,459961	0,448889	0,506667
0,644210	0,640000	0,666667
0,807784	0,808889	0,800000
0,924701	0,946667	0,955556
0,985156	0,986667	0,986667
1,000000	1,000000	1,000000

Рис. 6. Результат ускоренного имитационного моделирования для □ = 0.1.

Полный перебор	Имитационное моделирование	Ускоренное имитационное моделирование
0,000000	0,000000	0,000000
0,012162	0,011289	0,013556
0,057794	0,054622	0,060400
0,148369	0,144800	0,147156
0,286316	0,283689	0,283778
0,459961	0,457689	0,457778
0,644210	0,643022	0,644444
0,807784	0,806889	0,809111
0,924701	0,925733	0,924133
0,985156	0,983689	0,984800
1,000000	1,000000	1,000000

Рис. 7. Результат ускоренного имитационного моделирования для □ = 0.01.

Исходя из полученных данных, можно сделать вывод о том, что ускоренное имитационное моделирование было проведено верно, так как оценка вероятности существования пути должна отличаться от вероятности полного перебора не более чем на ±□.

5. Построение графиков:

Рассмотрим графики зависимости вероятности пути и оценки вероятности пути для □ = 0.1 и □ = 0.01 от вероятности существования ребра.

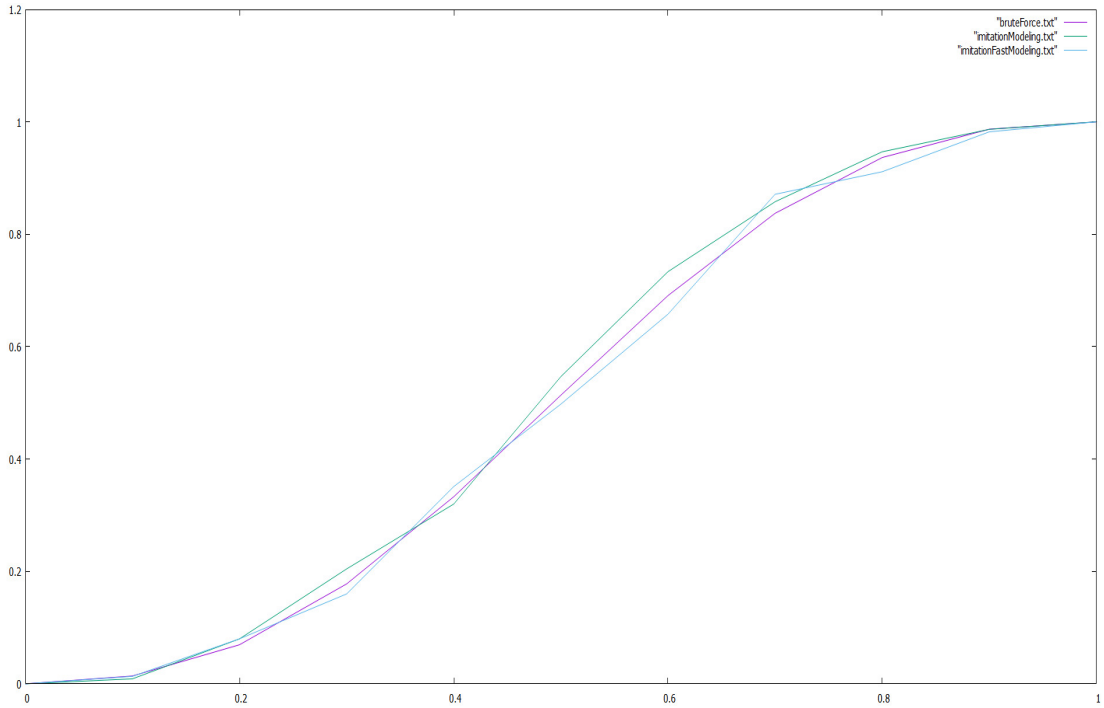


Рис. 8. Графики зависимости для □ = 0.1.

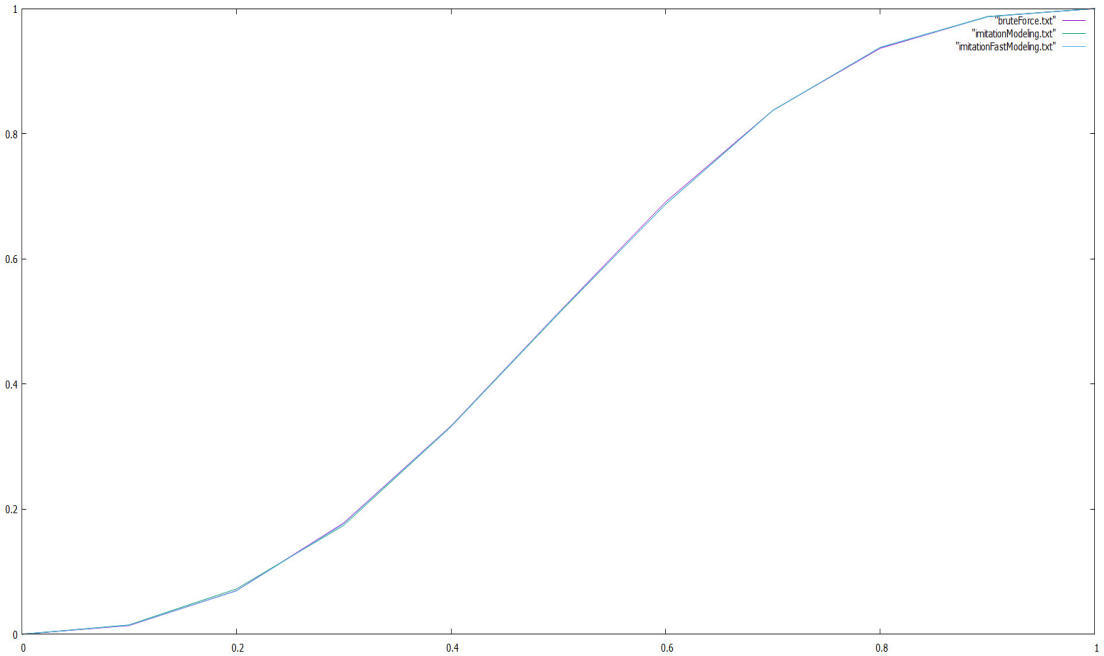


Рис. 9. Графики зависимости для □ = 0.01.

Оценка вероятности при □ = *0.1* имеет большее отклонение от точных значений, чем оценка вероятности при □ = *0.01*. Кроме того, из графиков видно, что оценка вероятности может отличаться от вероятности полного перебора как на -□, так и на +□.

Построим график зависимости отношения от вероятности появления ребра, где N – число экспериментов имитационного моделирования, N’ – число экспериментов ускоренного имитационного моделирования.

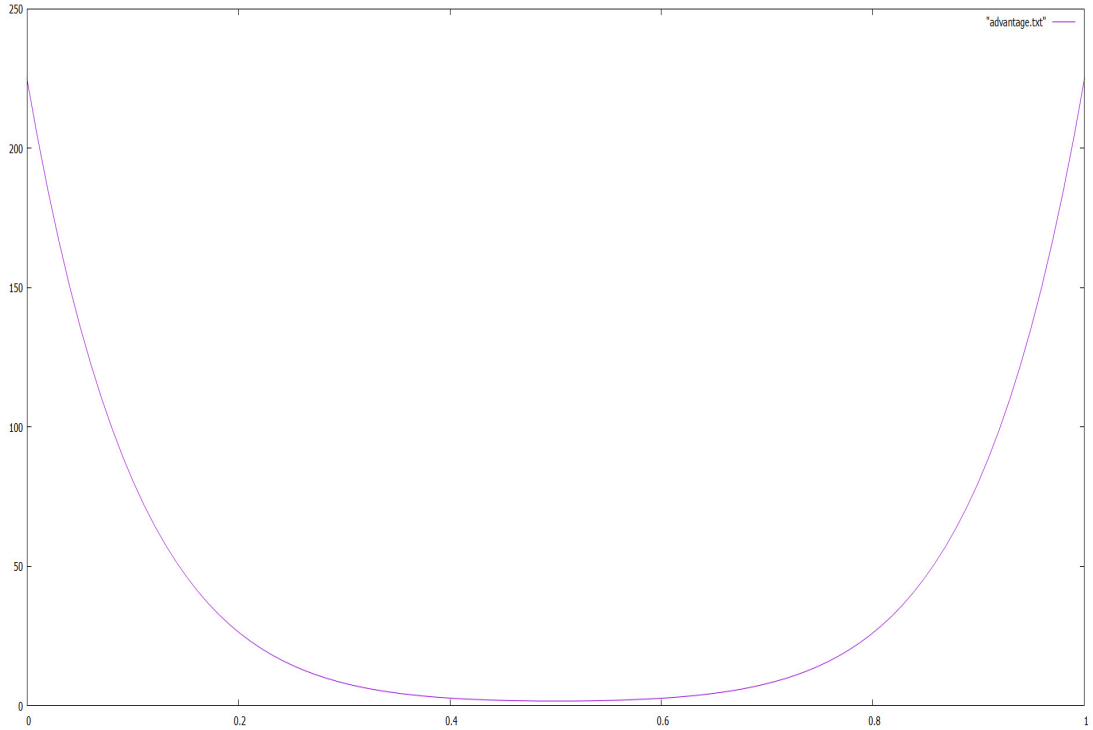


Рис. 10. График выигрыша ускоренного имитационного моделирования.

Из графика видно, что наибольший выигрыш ускоренное имитационное моделирование дает при вероятности существования ребра близкой либо к 0, либо к 1. По мере приближения от границ к вероятности равной ½ выигрыш будет заметно уменьшаться.

6. Вывод

В ходе выполнения работы, изучили методы имитационного моделирования и ускоренного имитационного моделирования. Определили число испытаний, необходимое для заданного доверительного интервала. Выяснили, что ускоренное имитационное моделирование дает больший выигрыш при вероятности существования ребра близкой к границам (то есть близкой к 0 или 1), так как при вероятности ребра p, близкой к 0, число ребер у подграфов чаще будет меньше, чем , а при вероятности ребра p, близкой к 1, число ребер у подграфов чаще будет больше, чем .

7. Приложение с кодом:

```
8. import java.io.FileWriter;
   import java.io.IOException;
   import java.util.*;

   import static java.lang.Integer.min;
   import static java.util.Arrays.fill;

   public class Main {

       public static void main(String[] args) {
           int MAX = Integer.MAX_VALUE / 2;
           int vNum = 7;
           int[][] graph = {{1, 1, 1, 1, MAX, MAX, MAX},
                   {1, 1, 1, MAX, MAX, MAX, 1},
                   {1, 1, 1, 1, MAX, MAX, MAX},
```

```
                {1, MAX, 1, 1, 1, MAX, MAX},
                {MAX, MAX, MAX, 1, 1, 1, 1},
                {MAX, MAX, MAX, MAX, 1, 1, 1},
                {MAX, 1, MAX, MAX, 1, 1, 1}};
        int[][] subgraph = new int[vNum][vNum];
        int eNum = 0;
        int[][] edges = new int[2][vNum * vNum];

        System.out.format("%1s%32s%40s", "Полный перебор", "Имитационное
моделирование", "Ускоренное имитационное моделирование");
        System.out.println();
        ArrayList<Double> bruteForce = new ArrayList<>();
        for (double p = 0; p <= 1; p += 0.1) {
            eNum = 0;
            double prob2 = 0;
            int k = 0;
            for (int i = 0; i < vNum; i++) {
                for (int j = 0; j < vNum; j++) {
                    if (i == j) {
                        break;
                    }
                    if (graph[i][j] == 1) {
                        eNum++;
                        edges[0][k] = i;
                        edges[1][k] = j;
                        k++;
                    }
                }
            }
            int[] check = new int[eNum];
            int clearCount = 0;
            for (int i = 0; i < eNum; i++) {
                check[i] = 1;
            }
            clearSubgraph(subgraph, vNum, MAX);
            for (int i = 0; i < eNum; i++) {
                if (check[i] == 2) {
                    clearCount++;
                    subgraph[edges[0][i]][edges[1][i]] = 1;
                    subgraph[edges[1][i]][edges[0][i]] = 1;
                }
            }
            if (dijkstra(0, vNum, subgraph, MAX) < MAX) {
                prob2 += Math.pow(p, clearCount) * Math.pow(1 - p, eNum -
clearCount);
            }
            while (enumeration(check, 2, eNum)) {
                clearSubgraph(subgraph, vNum, MAX);
                clearCount = 0;
                for (int i = 0; i < eNum; i++) {
                    if (check[i] == 2) {
                        clearCount++;
```

```
                        subgraph[edges[0][i]][edges[1][i]] = 1;
                        subgraph[edges[1][i]][edges[0][i]] = 1;
                    }
                }
                if (dijkstra(0, vNum, subgraph, MAX) < MAX) {
                    prob2 += Math.pow(p, clearCount) * Math.pow(1 - p, eNum -
clearCount);
                }
            }
            bruteForce.add(prob2);
        }
        save(bruteForce, "bruteForce.txt");

        double e = 0.01;
        double N = 9 / (e * e * 4);
        ArrayList<Double> imitationModeling = new ArrayList<>();
        for (double p = 0; p <= 1; p += 0.1) {
            double s = 0;
            for (int i = 0; i < N; i++) {
                int[] y = new int[eNum];
                int W = 0;
                for (int j = 0; j < eNum; j++) {
                    y[j] = randomY(p);
                    W += y[j];
                }
                clearSubgraph(subgraph, vNum, MAX);
                for (int j = 0; j < W; j++) {
                    int r = randomEdges(eNum);
                    if (subgraph[edges[0][r]][edges[1][r]] != 1) {
                        subgraph[edges[0][r]][edges[1][r]] = 1;
                        subgraph[edges[1][r]][edges[0][r]] = 1;
                    } else {
                        j--;
                    }
                }
                if (dijkstra(0, vNum, subgraph, MAX) < MAX) {
                    s += 1;
                }
            }
            double pIm = s / N;
            imitationModeling.add(pIm);
        }
        save(imitationModeling, "imitationModeling.txt");

        int lMin = 2;
        int lMax = eNum - 2;
        ArrayList<Double> imitationFastModeling = new ArrayList<>();
        ArrayList<Double> advantage = new ArrayList<>();
        for (double p = 0; p <= 1; p += 0.1) {
            double NFast = 0;
            double s = 0;
```

```
            for (int i = 0; i < N; i++) {
                int[] y = new int[eNum];
                int W = 0;
                for (int j = 0; j < eNum; j++) {
                    y[j] = randomY(p);
                    W += y[j];
                }
                if (W < lMin) {
                    continue;
                }
                if (W > lMax) {
                    s += 1;
                    continue;
                }
                NFast++;
                clearSubgraph(subgraph, vNum, MAX);
                for (int j = 0; j < W; j++) {
                    int r = randomEdges(eNum);
                    if (subgraph[edges[0][r]][edges[1][r]] != 1) {
                        subgraph[edges[0][r]][edges[1][r]] = 1;
                        subgraph[edges[1][r]][edges[0][r]] = 1;
                    } else {
                        j--;
                    }
                }
                if (dijkstra(0, vNum, subgraph, MAX) < MAX) {
                    s += 1;
                }
            }
            double pIm = s / N;
            imitationFastModeling.add(pIm);
            if (NFast != 0) {
                advantage.add(N / NFast);
            } else {
                advantage.add(N / (NFast + 1));
            }
        }
        save(imitationFastModeling, "imitationFastModeling.txt");
        save(advantage, "advantage.txt");
        for (int i = 0; i < 11; i ++) {
            System.out.format("%1f%32f%32f", bruteForce.get(i),
imitationModeling.get(i), imitationFastModeling.get(i));
            System.out.println();
        }
    }

    public static int dijkstra(int start, int vNum, int[][] graph, int MAX) {
        boolean[] used = new boolean[vNum];
        int[] dist = new int[vNum];

        fill(dist, MAX);
```

```
        dist[start] = 0;

        for (; ; ) {
            int v = -1;
            for (int nv = 0; nv < vNum; nv++)
                if (!used[nv] && dist[nv] < MAX && (v == -1 || dist[v] >
dist[nv]))
                    v = nv;
            if (v == -1) break;
            used[v] = true;
            for (int nv = 0; nv < vNum; nv++)
                if (!used[nv] && graph[v][nv] < MAX)
                    dist[nv] = min(dist[nv], dist[v] + graph[v][nv]);
        }

        return dist[6];
    }

    public static boolean enumeration(int[] check, int edge, int eNum) {
        int j = eNum - 1;
        while (j >= 0 && check[j] == edge) {
            j--;
        }
        if (j < 0) {
            return false;
        }
        if (check[j] >= edge) {
            j--;
        }
        check[j]++;
        if (j == eNum - 1) {
            return true;
        }
        for (int k = j + 1; k < eNum; k++) {
            check[k] = 1;
        }
        return true;
    }

    public static void clearSubgraph(int[][] subgraph, int vNum, int MAX) {
        for (int i = 0; i < vNum; i++) {
            for (int j = 0; j < vNum; j++) {
                if (i == j) {
                    subgraph[i][j] = 1;
                } else {
                    subgraph[i][j] = MAX;
                }
            }
        }
    }

    public static int randomY(double p) {
```

```
        return Math.random() < p ? 1 : 0;
    }

    public static int randomEdges(int edges) {
        return (int) (Math.random() * edges);
    }

    public static void save(List<Double> list, String filepath) {
        try {
            FileWriter fileWriter = new FileWriter(filepath);
            for (int i = 0; i < list.size(); ++i) {
                fileWriter.write(i * 0.1 + "    " + list.get(i) +"\n");
                fileWriter.flush();
            }
        } catch (IOException exception) {
            exception.printStackTrace();
        }
    }
}
```