Tallinna Tehnikaülikool

Algoritmid ja andmestruktuurid

Individuaaltöö aines "Algoritmid ja andmestruktuurid"

Koostaja: Nikita Viira

Juhendaja: Jaanus Pöial

Tallinn 2020

Table of contents

Description of the problem

Binary relation is considered to be transitive, if for all elements *a, b, c* in *X*, whenever *R* relates *a* to *b* and *b* to *c*, then *R* also relates *a* to *c*. In short, if there is a pair *(a, b)* and there is a pair *(b, c)*, the relation also needs to have a pair *(a, c)* to be considered transitive.

A transitive closure of a relation *R* on a set *X* is the smallest relation on *X* that contains *R* and is transitive. In short, a transitive closure is the minimum number of pairs, which have to be added to a relation in order for it to become transitive. If the relation is already transitive, then the transitive closure of it is equal to the relation itself.

If we are talking about the theory of graphs, the problem can be worded as the following: “Can I get from node A to node B by going straight from A to B?”. For example, I have a graph with nodes A, B, C and paths, which go from A to B and from B to C. In this example, we can see that A can’t go to C without having to go through B, so the logical step would be to add a path from A to C. That would make the graph transitive.

To find the transitive closure of a graph, I decided to use Floyd-Warshall's algorithm, which is usually used to find the shortest paths in a weighted graph. In my case, I will be using it to find the transitive closure of an unweighted directed graph.

Description of the solution

For finding the transitive closure of a graph, I used Floyd-Warshall's algorithm. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. Algorithm's performance is O(N^3), as it has 3 nested loops in it.

The logic behind the algorithm is the following:

- Create an adjacency matrix of a graph. Number of rows in a matrix is equal to the number of vertices in a graph. Every '1' in a graph indicates that there is a pair between the two vertices in a graph.
- Start iterating through all the possible rows and columns of a matrix (the ***i*** and ***j*** for loops).
- Pick a different vertex ***k***.
- If ***k*** ends up being an intermediate vertex to ***i*** and ***j*** such that the distance from ***i*** to ***j*** via ***k*** is smaller than the initial distance recorded, then we update the Distance between ***i*** and ***j***.

To go from the tree representation of graph, to the adjacency matrix representation, I used the method, which was provided by the teacher, which transformed the Graph class into a 2D array of integers.

Originally, the algorithm does the sum between the distance from ***i*** to ***k*** and from ***k*** to ***j***. But this is needed only in case of a weighted graph. Since my matrix consists of binary numbers (0 and 1), I can assume that if atleast one number is a '1', then the sum is gonna be '1', because 0 + 1 is 1 in logical arithmetics. By making a small change to the original algorithm, here is the implementation which I came up with:

```
public void transitiveClosureFloydWarshall(int[][] matrix) {
  for(int k = 0; k < matrix.length; k++) {
     for(int i = 0; i < matrix.length; i++) {
        for(int j = 0; j < matrix.length; j++) {
           if (matrix[i][k] == 1 && matrix[k][j] == 1) {
              matrix[i][j] = 1;
           }
        }
     }
  }
}
```

This implementation was extremely slow for a graph with more than 1000 vertices, so I started searching the Internet to find the ways of optimizing the algorithm. In one of the articles, I have stumbled upon an interesting solution. The only thing that I had to change in my first implementation was to add a check, that the path between ***i*** and ***k*** exists, before actually iterating for a third time. This reduced the complexity of the algorithm from O(N^3)

to O(N^2) because the third loop is now optional. That is how the final implementation looks like:

```
public void transitiveClosureFloydWarshall(int[][] matrix) {
  for (int k = 0; k < matrix.length; k++) {
     for (int i = 0; i < matrix.length; i++) {
        if (matrix[i][k] == 1) {
           for (int j = 0; j < matrix.length; j++) {
              if (matrix[k][j] == 1) {
                 matrix[i][j] = 1;
              }
           }
        }
     }
  }
}
```

In this implementation, you can see that the `matrix[i][k] == 1` check is now outside of the ***j*** loop, which makes the algorithm a lot faster than it originally was.

Since the task required the use of the Graph, Vertex and Arc classes, I had to make a method, which would go from the adjacency matrix representation of a graph, back to the tree structure of Graph class. For that, I created a method, which takes an adjacency matrix as the parameter, and recreates the current instance of Graph class, according to the matrix.

At first, I set the starting point of the graph (**`first`** variable) to null, because I am creating a new graph from scratch. Then, I initialize an array of vertices, the length of which is equal to the length of the matrix. Then, I iterate over each row of the matrix, and add a new vertex using the *createVertex()* method.

Now that I have all the vertices in the array, I start iterating through each row and column of the matrix in search of the '**1**' values. If I find the '**1**', it means that there is a pair between the two vertices ***i*** and ***j***, so I use the *createArc()* method. Since the graph has to be constructed the "other way around", I use the `matrix.length - 1 - i` instead of simply putting `i`.

User guidelines

Inside the *run()* method, you can insert the adjacency matrix of the graph you want using the text editor.
Execute the following commands in your terminal in order to run the application:

- **`javac -cp src src/GraphTask.java`**
- **`java -cp src GraphTask`**

Example:

```
/// ⬇️ ⬇️ Your test matrix goes here ⬇️ ⬇️ \\\
int[][] testMatrix = {
  {1, 0, 1, 0, 0},
  {1, 1, 0, 0, 0},
  {0, 1, 1, 0, 0},
  {0, 0, 1, 1, 1},
  {0, 0, 0, 0, 1}
};
```

Terminal output:

```
Your input graph:
G
0 -->  (0->2)  (0->0)
1 -->  (1->1)  (1->0)
2 -->  (2->2)  (2->1)
3 -->  (3->4)  (3->3)  (3->2)
4 -->  (4->4)

Your output graph:
G
0 -->  (0->2)  (0->1)  (0->0)
1 -->  (1->2)  (1->1)  (1->0)
2 -->  (2->2)  (2->1)  (2->0)
3 -->  (3->4)  (3->3)  (3->2)  (3->1)  (3->0)
4 -->  (4->4)
```

Testing plan

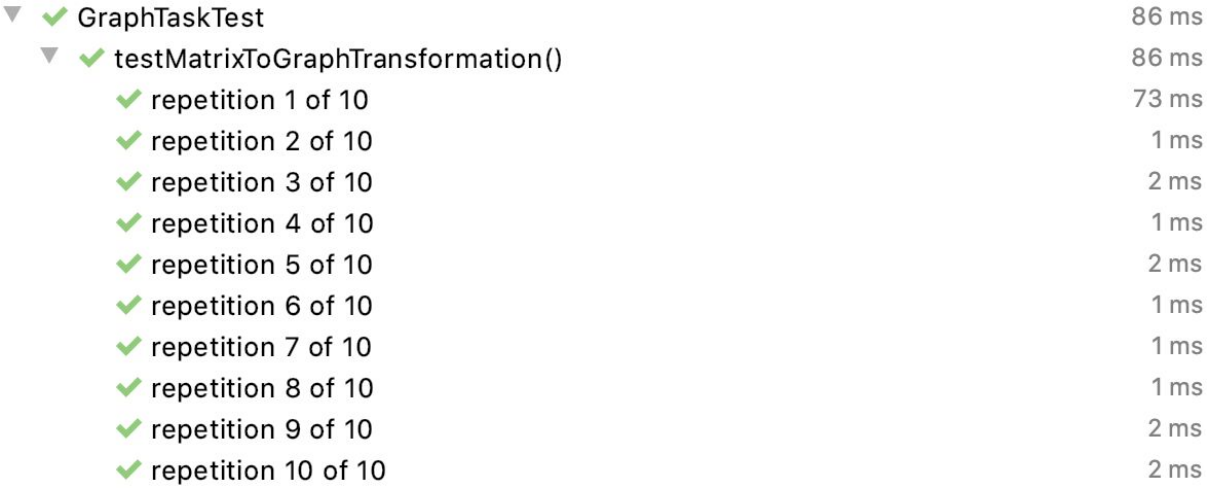
Test #1 (Matrix to graph transformation)

Description: Create a random graph and its matrix, then transform that matrix back to graph and create the matrix of the transformed graph. Check that the original matrix is equal to the matrix of the transformed graph. The test is repeated 10 times.

JUNIT test:

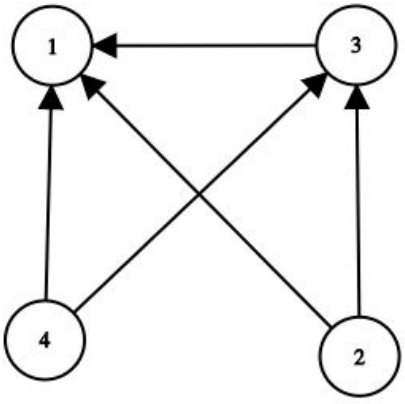
```
@RepeatedTest(10)
public void testMatrixToGraphTransformation() {
    graph.createRandomSimpleGraph(10, 10);
    int[][] originalMatrix = graph.createAdjMatrix();

    graph.createGraphFromAdjMatrix(originalMatrix);
    int[][] transformedGraphMatrix = graph.createAdjMatrix();

    assertThat(originalMatrix).isEqualTo(transformedGraphMatrix);
}
```

JUNIT test result:

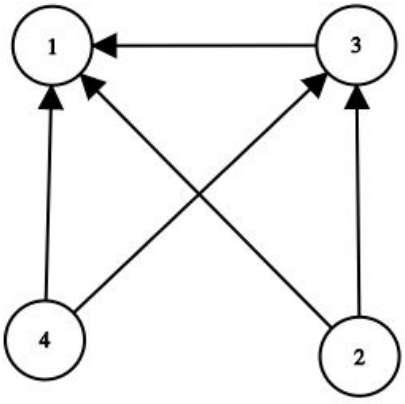


Test #2 (Floyd Warshall's algorithm)

Input graph:



Graph of transitive closure:



Matrix of this graph:

1	**1**	0	**1**
0	0	0	0
0	**1**	0	**1**
0	**1**	0	**1**

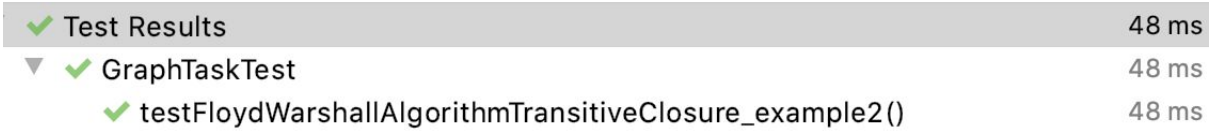
Transitive closure of this graph:

1	**1**	0	**1**
0	0	0	0
0	**1**	0	**1**
0	**1**	0	**1**

JUNIT test:

```
@Test
public void testFloydWarshallAlgorithmTransitiveClosure_example2() {
    graph.createGraphFromAdjMatrix(new int[][]{
      {1,1,0,1},{0,0,0,0},{0,1,0,1},{0,1,0,1}
    });

    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);

    assertThat(adjMatrix).isEqualTo(new int[][]{
      {1,1,0,1},{0,0,0,0},{0,1,0,1},{0,1,0,1}
    });
}
```

JUNIT test result:



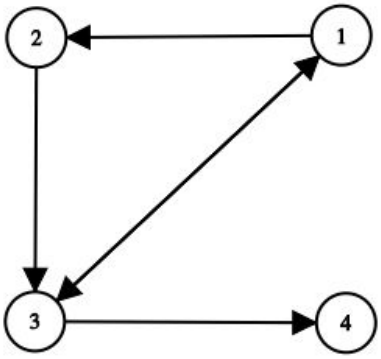
Manual control:

A					A					A^2			
1	**1**	0	**1**	**X**	**1**	**1**	0	**1**	=	**1**	**1**	0	**1**
0	0	0	0		0	0	0	0		0	0	0	0
0	**1**	0	**1**		0	**1**	0	**1**		0	**1**	0	**1**
0	**1**	0	**1**		0	**1**	0	**1**		0	**1**	0	**1**

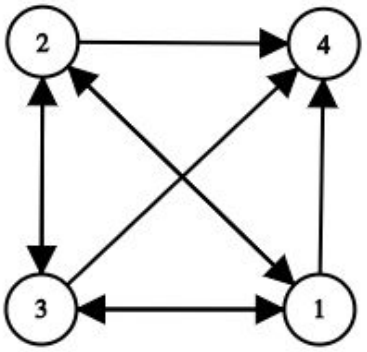
After multiplying the matrix by itself, we received the same matrix, which means that this relation was already transitive. If the relation is transitive, it is equal to its transitive closure.

Test #3 (Floyd Warshall's algorithm)

Input graph:



Graph of transitive closure:



Matrix of this graph:

0	**1**	**1**	0
0	0	**1**	0
1	0	0	**1**
0	0	0	0

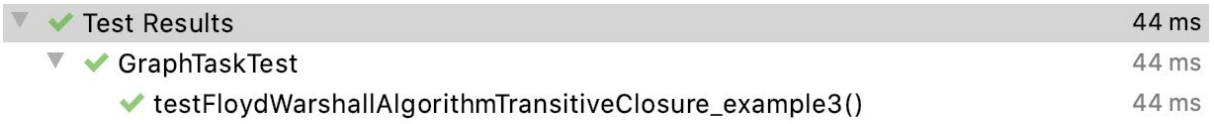
Transitive closure matrix of this graph:

1	**1**	**1**	**1**
1	**1**	**1**	**1**
1	**1**	**1**	**1**
0	0	0	0

JUNIT test:

```
@Test
public void testFloydWarshallAlgorithmTransitiveClosure_example3() {
    graph.createGraphFromAdjMatrix(new int[][]{
      {0,1,1,0},{0,0,1,0},{1,0,0,1},{0,0,0,0}
    });

    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);

    assertThat(adjMatrix).isEqualTo(new int[][]{
      {1,1,1,1},{1,1,1,1},{1,1,1,1},{0,0,0,0}
    });
}
```

JUNIT test result:



Manual control:

A | A | A^2

0	**1**	**1**	0		0	**1**	**1**	0		**1**	0	**1**	**1**
0	0	**1**	0	**X**	0	0	**1**	0	=	**1**	0	**0**	**1**
1	0	0	**1**		**1**	0	0	**1**		**0**	**1**	**1**	**0**
0	0	0	0		0	0	0	0		0	0	0	0

A^2 | A | A^3

1	0	**1**	**1**		0	**1**	**1**	0		**1**	**1**	**1**	**1**
1	0	0	**1**	**X**	0	0	**1**	0	=	0	**1**	**1**	0
0	**1**	**1**	0		**1**	0	0	**1**		**1**	0	**1**	**1**
0	0	0	0		0	0	0	0		0	0	0	0

A^3 | A | A^4

1	**1**	**1**	**1**		0	**1**	**1**	0		**1**	**1**	**1**	**1**
0	**1**	**1**	0	**X**	0	0	**1**	0	=	**1**	0	**1**	**1**
1	0	**1**	**1**		**1**	0	0	**1**		**1**	**1**	**1**	**1**
0	0	0	0		0	0	0	0		0	0	0	0

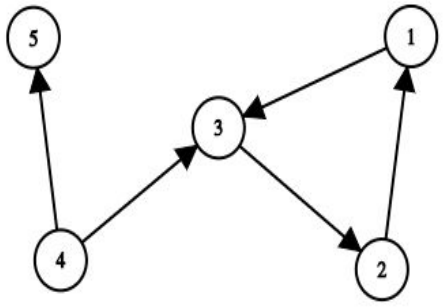
A^4 | A | A^5

1	**1**	**1**	**1**		0	**1**	**1**	0		**1**	**1**	**1**	**1**
1	0	**1**	**1**	**X**	0	0	**1**	0	=	**1**	**1**	**1**	**1**
1	**1**	**1**	**1**		**1**	0	0	**1**		**1**	**1**	**1**	**1**
0	0	0	0		0	0	0	0		0	0	0	0

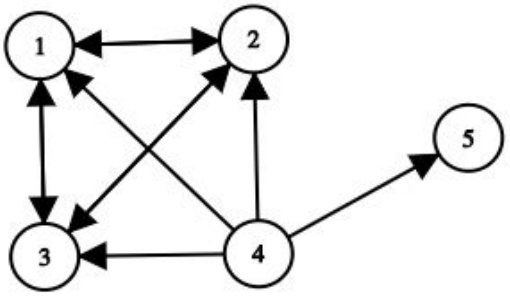
The next multiplication gives the same result, which means that the power of 5 is the matrix we need. By the formula of transitive closure, we do $A + A^5$, which results in A^5. So A^5 is the transitive closure of this graph.

Test #4 (Floyd Warshall's algorithm)

Input graph:



Graph of transitive closure:



Matrix of this graph:

1	0	**1**	0	0
1	**1**	0	0	0
0	**1**	**1**	0	0
0	0	**1**	**1**	**1**
0	0	0	0	**1**

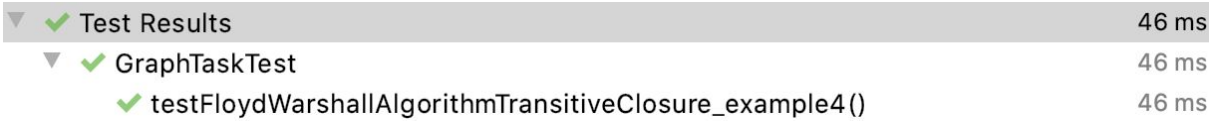
Transitive closure of this graph:

1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	**1**	**1**
0	0	0	0	**1**

JUNIT test:

```
@Test
public void testFloydWarshallAlgorithmTransitiveClosure_example4() {
    graph.createGraphFromAdjMatrix(new int[][]{
      {1,0,1,0,0},{1,1,0,0,0},{0,1,1,0,0},{0,0,1,1,1},{0,0,0,0,1}
    });

    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);

    assertThat(adjMatrix).isEqualTo(new int[][]{
      {1,1,1,0,0},{1,1,1,0,0},{1,1,1,0,0},{1,1,1,1,1},{0,0,0,0,1}
    });
}
```

JUNIT test result:



Manual control:

A X A = A^2

1	0	**1**	0	0
1	**1**	0	0	0
0	**1**	**1**	0	0
0	0	**1**	**1**	**1**
0	0	0	0	**1**

X

1	0	**1**	0	0
1	**1**	0	0	0
0	**1**	**1**	0	0
0	0	**1**	**1**	**1**
0	0	0	0	**1**

=

1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	0	0
0	**1**	**1**	**1**	**1**
0	0	0	0	**1**

A^2 X A = A^3

1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	0	0
0	**1**	**1**	**1**	**1**
0	0	0	0	**1**

X

1	0	**1**	0	0
1	**1**	0	0	0
0	**1**	**1**	0	0
0	0	**1**	**1**	**1**
0	0	0	0	**1**

=

1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	0	0
1	**1**	**1**	**1**	**1**
0	0	0	0	**1**

The next multiplication gives the same result, which means that the power of 3 is the matrix we need. By the formula of transitive closure, we do $A + A^3$, which results in A^3. So A^3 is the transitive closure of this graph.

Test #5 (Timeout test)

Description: Create a random graph with 5000 vertices and 5000 arcs. Create an adjacency matrix of this graph. Use the algorithm to find the transitive closure of the matrix. Transform matrix back to the graph. The test is repeated 25 times and checks that the execution time stays less than 2 seconds.

JUNIT test:

```
@RepeatedTest(25)
public void testCase5000Vertices() {
    graph.createRandomSimpleGraph(5000, 5000);
    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);
    graph.createGraphFromAdjMatrix(adjMatrix);

    assertThat(currentTimeMillis() - startingPoint).isLessThan(2000L);
}
```

JUNIT test result:

✔ GraphTaskTest	34 s 457 ms
✔ testCase5000Vertices()	34 s 457 ms
✔ repetition 1 of 25	1 s 873 ms
✔ repetition 2 of 25	1 s 331 ms
✔ repetition 3 of 25	1 s 266 ms
✔ repetition 4 of 25	1 s 625 ms
✔ repetition 5 of 25	1 s 761 ms
✔ repetition 6 of 25	1 s 404 ms
✔ repetition 7 of 25	1 s 666 ms
✔ repetition 8 of 25	1 s 562 ms
✔ repetition 9 of 25	1 s 143 ms
✔ repetition 10 of 25	1 s 784 ms
✔ repetition 11 of 25	1 s 268 ms
✔ repetition 12 of 25	1 s 81 ms
✔ repetition 13 of 25	1 s 152 ms
✔ repetition 14 of 25	1 s 380 ms
✔ repetition 15 of 25	1 s 440 ms
✔ repetition 16 of 25	1 s 228 ms
✔ repetition 17 of 25	1 s 176 ms
✔ repetition 18 of 25	1 s 180 ms
✔ repetition 19 of 25	1 s 379 ms
✔ repetition 20 of 25	1 s 255 ms
✔ repetition 21 of 25	1 s 551 ms
✔ repetition 22 of 25	1 s 360 ms
✔ repetition 23 of 25	1 s 230 ms
✔ repetition 24 of 25	1 s 112 ms
✔ repetition 25 of 25	1 s 250 ms

Median execution time is around 1,3 seconds.

Speed analysis

Description: Random graphs with 500, 1000, 2000, 4000, 8000 vertices and arcs are created. Measure the execution time of each of the following steps:

- Graph creation
- Matrix creation
- Algorithm execution
- Transforming matrix back to graph

JUNIT test:

```
@ParameterizedTest
@ValueSource(ints = {500, 1000, 2000, 4000, 8000})
public void speedTest(int n) {
    graph.createRandomSimpleGraph(n, n);
    System.out.println("Graph creation took "
      + (currentTimeMillis() - startingPoint) + " milliseconds");

    long matrixCreationStart = currentTimeMillis();
    int[][] adjMatrix = graph.createAdjMatrix();
    System.out.println("Matrix creation took "
      + (currentTimeMillis() - matrixCreationStart) + " milliseconds");

    long algorithmStart = currentTimeMillis();
    graph.transitiveClosureFloydWarshall(adjMatrix);
    System.out.println("Algorithm execution took "
      + (currentTimeMillis() - algorithmStart) + " milliseconds");

    long transformationStart = currentTimeMillis();
    graph.createGraphFromAdjMatrix(adjMatrix);
    System.out.println("Graph transformation took "
      + (currentTimeMillis() - transformationStart) + " milliseconds");

    System.out.println("Overall execution time is "
      + (currentTimeMillis() - startingPoint) + " milliseconds");
}
```

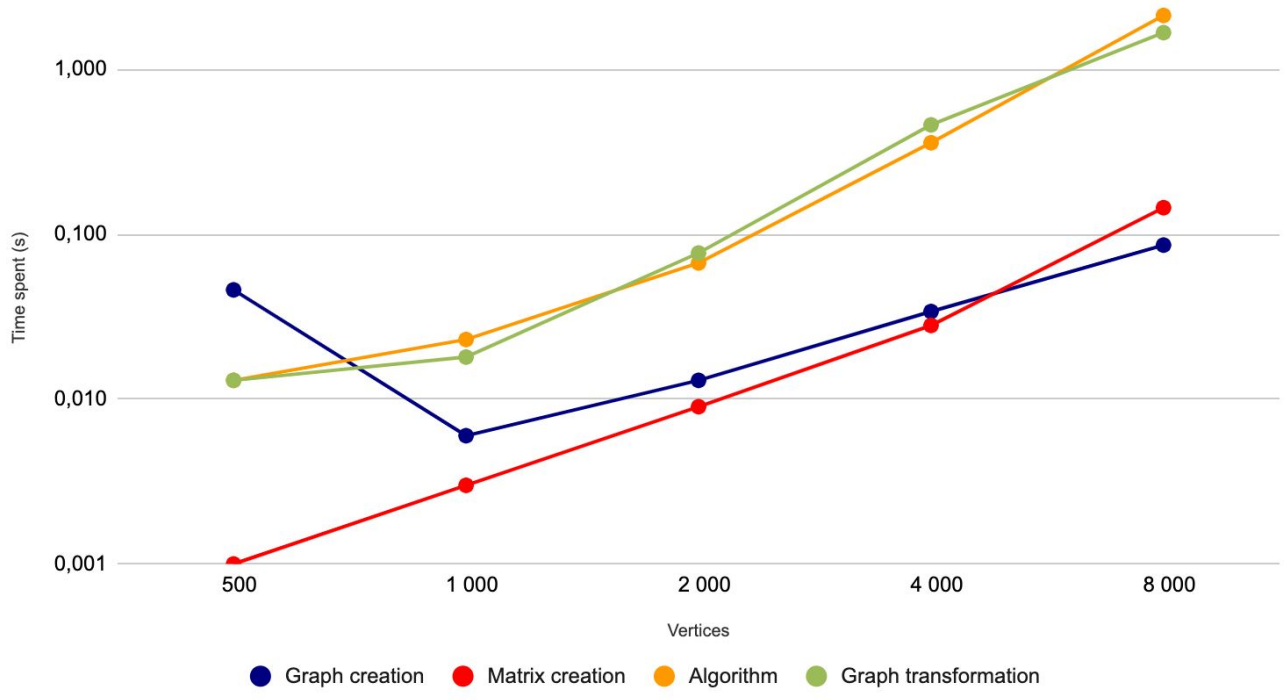
Console output:

- **500 vertices**
  Graph creation took 46 milliseconds
  Matrix creation took 1 milliseconds
  Algorithm execution took 13 milliseconds
  Graph transformation took 13 milliseconds

Overall execution time is 80 milliseconds

- **1000 vertices**
  Graph creation took 6 milliseconds
  Matrix creation took 3 milliseconds
  Algorithm execution took 23 milliseconds
  Graph transformation took 18 milliseconds
  Overall execution time is 47 milliseconds
- **2000 vertices**
  Graph creation took 13 milliseconds
  Matrix creation took 9 milliseconds
  Algorithm execution took 67 milliseconds
  Graph transformation took 77 milliseconds
  Overall execution time is 167 milliseconds
- **4000 vertices**
  Graph creation took 34 milliseconds
  Matrix creation took 28 milliseconds
  Algorithm execution took 359 milliseconds
  Graph transformation took 461 milliseconds
  Overall execution time is 883 milliseconds
- **8000 vertices**
  Graph creation took 86 milliseconds
  Matrix creation took 145 milliseconds
  Algorithm execution took 2126 milliseconds
  Graph transformation took 1672 milliseconds
  Overall execution time is 4029 milliseconds

Speed chart:



Used literature

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- https://en.wikipedia.org/wiki/Transitive_closure
- https://csacademy.com/app/graph_editor/
- https://studfile.net/preview/4229244/page:3/
- https://siddharths2710.wordpress.com/2017/05/12/optimizing-the-floyd-warshall-algorithm/

Full text of the programm

```
/** Container class to different classes, that makes the whole
* set of classes one class formally.
*/
public class GraphTask {

  /** Main method. */
  public static void main (String[] args) {
     GraphTask a = new GraphTask();
     a.run();
  }

  public void run() {
     Graph g = new Graph("G");

     /// ⬇ ⬇ Your test matrix goes here ⬇ ⬇ \\\
     int[][] testMatrix = {
       {1, 0, 1, 0, 0},
       {1, 1, 0, 0, 0},
       {0, 1, 1, 0, 0},
       {0, 0, 1, 1, 1},
       {0, 0, 0, 0, 1}
     };
     g.createGraphFromAdjMatrix(testMatrix);
     System.out.print("Your input graph: ");
     System.out.println(g.toString());

     int[][] adjMatrix = g.createAdjMatrix();
     g.transitiveClosureFloydWarshall(adjMatrix);
     g.createGraphFromAdjMatrix(adjMatrix);

     System.out.print("Your output graph: ");
     System.out.println(g.toString());
  }

  static class Vertex {
     private final String id;
     private Vertex next;
     private Arc first;
     private int info = 0;

     public Vertex(String s, Vertex v, Arc e) {
        this.id = s;
        this.next = v;
        this.first = e;
     }

     public Vertex(String s) {
        this(s, null, null);
     }
```

```
    @Override
    public String toString() {
        return id;
    }
}


/** Arc represents one arrow in the graph. Two-directional edges are
 * represented by two Arc objects (for both directions).
 */
static class Arc {
    private final String id;
    private Vertex target;
    private Arc next;

    public Arc(String s, Vertex v, Arc a) {
        this.id = s;
        this.target = v;
        this.next = a;
    }

    public Arc(String s) {
        this(s, null, null);
    }

    @Override
    public String toString() {
        return id;
    }
}


public static class Graph {
    private final String id;
    private Vertex first;

    public Graph(String s, Vertex v) {
        this.id = s;
        this.first = v;
    }

    public Graph(String s) {
        this(s, null);
    }

    @Override
    public String toString() {
        String nl = System.getProperty ("line.separator");
        StringBuilder sb = new StringBuilder(nl);
        sb.append (id);
        sb.append (nl);
        Vertex v = first;
        while (v != null) {
```

```
            sb.append (v.toString());
            sb.append (" -->");
            Arc a = v.first;
            while (a != null) {
               sb.append (" ");
               sb.append (a.toString());
               sb.append (" (");
               sb.append (v.toString());
               sb.append ("->");
               sb.append (a.target.toString());
               sb.append (")");
               a = a.next;
            }
            sb.append (nl);
            v = v.next;
         }
         return sb.toString();
      }

      public Vertex createVertex(String vid) {
         Vertex res = new Vertex (vid);
         res.next = first;
         first = res;
         return res;
      }

      public void createArc(String aid, Vertex from, Vertex to) {
         Arc res = new Arc (aid);
         res.next = from.first;
         from.first = res;
         res.target = to;
      }

      /**
       * Create a connected undirected random tree with n vertices.
       * Each new vertex is connected to some random existing vertex.
       * @param n number of vertices added to this graph
       */
      public void createRandomTree (int n) {
         if (n <= 0) return;
         Vertex[] varray = new Vertex [n];
         for (int i = 0; i < n; i++) {
            varray [i] = createVertex ("v" + (n - i));
            if (i > 0) {
               int vnr = (int)(Math.random()*i);
               createArc ("a" + varray [vnr].toString() + "_"
                  + varray [i].toString(), varray [vnr], varray [i]);
               createArc ("a" + varray [i].toString() + "_"
                  + varray [vnr].toString(), varray [i], varray [vnr]);
            }
         }
      }
```

```
/**
 * Create an adjacency matrix of this graph.
 * Side effect: corrupts info fields in the graph
 * @return adjacency matrix
 */
public int[][] createAdjMatrix() {
   int info = 0;
   Vertex v = first;
   while (v != null) {
      v.info = info++;
      v = v.next;
   }
   int[][] res = new int [info][info];
   v = first;
   while (v != null) {
      int i = v.info;
      Arc a = v.first;
      while (a != null) {
         int j = a.target.info;
         res [i][j]++;
         a = a.next;
      }
      v = v.next;
   }
   return res;
}

/**
 * Create a connected simple (undirected, no loops, no multiple
 * arcs) random graph with n vertices and m edges.
 * @param n number of vertices
 * @param m number of edges
 */
public void createRandomSimpleGraph (int n, int m) {
   if (n <= 0)
      return;
   if (m < n-1 || m > n*(n-1)/2)
      throw new IllegalArgumentException
        ("Impossible number of edges: " + m);
   first = null;
   createRandomTree (n);       // n-1 edges created here
   Vertex[] vert = new Vertex [n];
   Vertex v = first;
   int c = 0;
   while (v != null) {
      vert[c++] = v;
      v = v.next;
   }
   int[][] connected = createAdjMatrix();
   int edgeCount = m - n + 1;  // remaining edges
   while (edgeCount > 0) {
      int i = (int)(Math.random()*n);  // random source
      int j = (int)(Math.random()*n);  // random target
```

```
            if (i==j)
               continue;  // no loops
            if (connected [i][j] != 0 || connected [j][i] != 0)
               continue;  // no multiple edges
            Vertex vi = vert [i];
            Vertex vj = vert [j];
            createArc ("a" + vi.toString() + "_" + vj.toString(), vi, vj);
            connected [i][j] = 1;
            createArc ("a" + vj.toString() + "_" + vi.toString(), vj, vi);
            connected [j][i] = 1;
            edgeCount--;  // a new edge happily created
         }
      }

      /**
       * Given an adjacency matrix of an unweighted directed graph,
       * find the shortest distances between every pair of vertices
       * by using the Floyd Warshall's algorithm.
       * Result of this function is a transitive closure of an input matrix.
       * @param matrix an adjacency matrix of a graph
       *               (Every "1" in a matrix indicates a pair between two
vertices)
       */
      public void transitiveClosureFloydWarshall(int[][] matrix) {
         for (int k = 0; k < matrix.length; k++) {
            for (int i = 0; i < matrix.length; i++) {
               if (matrix[i][k] == 1) {
                  for (int j = 0; j < matrix.length; j++) {
                     if (matrix[k][j] == 1) {
                        matrix[i][j] = 1;
                     }
                  }
               }
            }
         }
      }

      /**
       * Take an adjacency matrix and override the existing graph according
       * to the structure of the input matrix.
       * @param matrix an adjacency matrix of a graph
       *               (Every "1" in a matrix indicates a pair between two
vertices)
       */
      public void createGraphFromAdjMatrix(int[][] matrix) {
         first = null;
         Vertex[] vertices = new Vertex[matrix.length];

         for (int i = 0; i < matrix.length; i++) {
            vertices[i] = createVertex(String.valueOf(matrix.length - 1 - i));
         }

         for (int i = 0; i < matrix.length; i++) {
```

```
            for (int j = 0; j < matrix.length; j++) {
               if (matrix[i][j] == 1) {
                  createArc("", vertices[matrix.length - 1 - i],
vertices[matrix.length - 1 - j]);
               }
            }
         }
      }
   }
}
```

Full text of the JUNIT test

Used testing frameworks:

- JUNIT 5.0
- ASSERTJ

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static java.lang.System.currentTimeMillis;
import static org.assertj.core.api.Assertions.assertThat;

/** Testklass.
* @author jaanus
*/
public class GraphTaskTest {
  private static  Long startingPoint;
  private static GraphTask.Graph graph;

  @BeforeEach
  public void setUp() {
     graph = new GraphTask.Graph("G");
     startingPoint = currentTimeMillis();
  }

  @Test
  @DisplayName(
    "The matrix of the graph is following: "
      + "0, 1, 1"
      + "0, 0, 1"
      + "1, 0, 0"
  )
  public void testFloydWarshallAlgorithmTransitiveClosure_example1() {
     graph.createGraphFromAdjMatrix(new int[][]{
       {0,1,1},{0,0,1},{1,0,0}
     });

     int[][] adjMatrix = graph.createAdjMatrix();
     graph.transitiveClosureFloydWarshall(adjMatrix);

     assertThat(adjMatrix).isEqualTo(new int[][]{
       {1,1,1},{1,1,1},{1,1,1}
     });
  }

  @Test
```

```
@DisplayName(
  "The matrix of the graph is following: "
    + "1, 1, 0, 1"
    + "0, 0, 0, 0"
    + "0, 1, 0, 1"
    + "0, 1, 0, 1"
)
public void testFloydWarshallAlgorithmTransitiveClosure_example2() {
   graph.createGraphFromAdjMatrix(new int[][]{
     {1,1,0,1},{0,0,0,0},{0,1,0,1},{0,1,0,1}
   });

   int[][] adjMatrix = graph.createAdjMatrix();
   graph.transitiveClosureFloydWarshall(adjMatrix);

   assertThat(adjMatrix).isEqualTo(new int[][]{
     {1,1,0,1},{0,0,0,0},{0,1,0,1},{0,1,0,1}
   });
}

@Test
@DisplayName(
  "The matrix of the graph is following: "
    + "0, 1, 1, 0"
    + "0, 0, 1, 0"
    + "1, 0, 0, 1"
    + "0, 0, 0, 0"
)
public void testFloydWarshallAlgorithmTransitiveClosure_example3() {
   graph.createGraphFromAdjMatrix(new int[][]{
     {0,1,1,0},{0,0,1,0},{1,0,0,1},{0,0,0,0}
   });

   int[][] adjMatrix = graph.createAdjMatrix();
   graph.transitiveClosureFloydWarshall(adjMatrix);

   assertThat(adjMatrix).isEqualTo(new int[][]{
     {1,1,1,1},{1,1,1,1},{1,1,1,1},{0,0,0,0}
   });
}

@Test
@DisplayName(
  "The matrix of the graph is following: "
    + "1, 0, 1, 0, 0"
    + "1, 1, 0, 0, 0"
    + "0, 1, 1, 0, 0"
    + "0, 0, 1, 1, 1"
    + "0, 0, 0, 0, 1"
)
public void testFloydWarshallAlgorithmTransitiveClosure_example4() {
   graph.createGraphFromAdjMatrix(new int[][]{
     {1,0,1,0,0},{1,1,0,0,0},{0,1,1,0,0},{0,0,1,1,1},{0,0,0,0,1}
```

```
    });

    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);

    assertThat(adjMatrix).isEqualTo(new int[][]{
      {1,1,1,0,0},{1,1,1,0,0},{1,1,1,0,0},{1,1,1,1,1},{0,0,0,0,1}
    });
}

@RepeatedTest(10)
public void testMatrixToGraphTransformation() {
    graph.createRandomSimpleGraph(10, 10);
    int[][] originalMatrix = graph.createAdjMatrix();

    graph.createGraphFromAdjMatrix(originalMatrix);
    int[][] transformedGraphMatrix = graph.createAdjMatrix();

    assertThat(originalMatrix).isEqualTo(transformedGraphMatrix);
}

@RepeatedTest(25)
public void testCase5000Vertices() {
    graph.createRandomSimpleGraph(5000, 5000);
    int[][] adjMatrix = graph.createAdjMatrix();
    graph.transitiveClosureFloydWarshall(adjMatrix);
    graph.createGraphFromAdjMatrix(adjMatrix);

    assertThat(currentTimeMillis() - startingPoint).isLessThan(2000L);
}

@ParameterizedTest
@ValueSource(ints = {500, 1000, 2000, 4000, 8000})
public void speedTest(int n) {
    graph.createRandomSimpleGraph(n, n);
    System.out.println("Graph creation took "
      + (currentTimeMillis() - startingPoint) + " milliseconds");

    long matrixCreationStart = currentTimeMillis();
    int[][] adjMatrix = graph.createAdjMatrix();
    System.out.println("Matrix creation took "
      + (currentTimeMillis() - matrixCreationStart) + " milliseconds");

    long algorithmStart = currentTimeMillis();
    graph.transitiveClosureFloydWarshall(adjMatrix);
    System.out.println("Algorithm execution took "
      + (currentTimeMillis() - algorithmStart) + " milliseconds");

    long transformationStart = currentTimeMillis();
    graph.createGraphFromAdjMatrix(adjMatrix);
    System.out.println("Graph transformation took "
      + (currentTimeMillis() - transformationStart) + " milliseconds");
```

```
        System.out.println("Overall execution time is "
          + (currentTimeMillis() - startingPoint) + " milliseconds");
    }
}
```