```
package graphtraversals;

import graphs.Graph;
import graphs.Graph.AdjacentEdge;

import java.util.Queue;
import java.util.LinkedList;

public class BreadthFirst<V extends BreadthFirstVertex,E> {

    public enum Colour{ White, Grey, Black }

    public BreadthFirst() {
        super();
    }

    public void breadthFirstSearch( Graph<V,E> G, V source ) {
        /* Traverse all the vertices of the graph initializing them
         * to be unvisited (White), distance infinity
         * (approximately) from the source and to have no parent.
         */
        for( V u : G ) {
            u.colour = Colour.White;
            u.distance = Integer.MAX_VALUE;
            u.parent = null;
        }
        /* Start the breadth-first search from the source.
         * Colour it Grey to indicate it has been noticed,
         * but not yet searched. The source is distance 0 from
         * itself, and has no parent. */
        source.colour = Colour.Grey;
        source.distance = 0;
        source.parent = null;
        /* To perform a breadth-first search a first-in first-out
         * queue is used to store the vertices that have been
         * discovered, but not yet processed.
         * The queue is initialized to contain the source vertex.
         */
        Queue<V> Q = new LinkedList<V>();
        Q.add( source );
        while( ! Q.isEmpty() ) {
            /* Process the first vertex, u, in the queue. */
            V u = Q.remove();
            /* Traverse all adjacent vertices of u searching for
                    undiscovered (White) vertices. */
            for( AdjacentEdge<V, E> e : G.adjacent(u) ) {
                V v = e.target;
                if( v.colour == Colour.White ) {
                    /* For each undiscovered vertex, v,
                     * process it by:
                     * marking v as discovered (Grey),
                     * calculating its distance from the source,
                     * noting that v's parent is u, and
                     * enqueue v so that its adjacent
                     * vertices are searched.
                     */
                    v.colour = Colour.Grey;
                    v.distance = u.distance + 1;
                    v.parent = u;
                    Q.add(v);
                }
            }
            /* Having examined the vertices adjacent to u,
             * we mark u as being completely processed (Black).
             */
            u.colour = Colour.Black;
        }
    }
}
```

Aug 25, 14 12:58 **BreadthFirstTest.java** Page 1/1

```
package graphtraversals;

import graphs.DGraphAdj;
import graphs.Graph;
import graphtraversals.BreadthFirstVertex;

public class BreadthFirstTest {

    public static void main(String[] args) {
        BreadthFirst<BreadthFirstVertex,Object> bf =
                new BreadthFirst<BreadthFirstVertex,Object>();
        Graph<BreadthFirstVertex,Object> G =
                new DGraphAdj<BreadthFirstVertex,Object>();
        BreadthFirstVertex v0 = new BreadthFirstVertex(); G.addVertex( v0 );
        BreadthFirstVertex v1 = new BreadthFirstVertex(); G.addVertex( v1 );
        BreadthFirstVertex v2 = new BreadthFirstVertex(); G.addVertex( v2 );
        BreadthFirstVertex v3 = new BreadthFirstVertex(); G.addVertex( v3 );
        BreadthFirstVertex v4 = new BreadthFirstVertex(); G.addVertex( v4 );
        BreadthFirstVertex v5 = new BreadthFirstVertex(); G.addVertex( v5 );

        Object ob = new Object();
        G.addEdge( v0, v1, ob );
        G.addEdge( v0, v3, ob );
        G.addEdge( v1, v4, ob );
        G.addEdge( v3, v1, ob );
        G.addEdge( v4, v3, ob );
        G.addEdge( v2, v4, ob );
        G.addEdge( v2, v5, ob );
        G.addEdge( v5, v5, ob );

        bf.breadthFirstSearch( G, v0 );
        for( BreadthFirstVertex u : G ) {
            System.out.println( u );
        }
    }

}
```

Sep 04, 14 16:49 **BreadthFirstVertex.java** Page 1/1

```
package graphtraversals;

import graphs.Vertex;
import graphtraversals.BreadthFirst.Colour;

public class BreadthFirstVertex extends Vertex {
    public Colour colour;
    public int distance;
    public BreadthFirstVertex parent;

    public BreadthFirstVertex( ) {
        super();
        this.colour = Colour.White;
        this.distance = Integer.MAX_VALUE;
        this.parent = null;
    }

    public String toString() {
        return super.toString() + " distance " +
                (distance == Integer.MAX_VALUE ? "infinity" : distance) +
            " parent " + (parent == null ? "null" : parent );
    }
}
```

```
package graphtraversals;

import graphs.UGraph;
import graphs.Vertex;
import graphs.Graph.AdjacentEdge;

class Components {

    private enum Colour{ White, Grey, Black }

    public class V extends Vertex {
        private Colour colour;
        public int component;

        public V() {
            colour = Colour.White;
        }

        public String toString() {
            return super.toString() + " component " + component;
        }
    }

    public Components() {
    }

    private int comp;

    /** Find the connected components of an undirected graph.
     * The array component maps each vertex to its component
     * number. Two vertices are in the same component if and
     * only if they have the same component number. */

    public void components( UGraph<V,Object> G ) {
        /* Mark all vertices of G as unvisited (White). */
        for( V u : G ) {
            u.colour = Colour.White;
        }
        comp = 0;
        /* Traverse through the vertices and for each vertex u
         * that has not been visited start a new component and
         * mark all vertices connected to u as being in that
         * component. */
        for( V u : G ) {
            if( u.colour == Colour.White ) {
                comp++;
                visit(G,u);
            }
        }
    }

    private void visit( UGraph<V,Object> G, V u ) {
        /** Visit all vertices connected to u and mark as in
         * the current component.
         */
        u.colour = Colour.Grey;
        u.component = comp;
        for( AdjacentEdge<V,Object> e : G.adjacent(u) ) {
            if( e.target.colour == Colour.White ) {
                visit( G, e.target );
            }
        }
```



```
        u.colour = Colour.Black;
    }
}
```

Aug 25, 14 12:59 **ComponentsTest.java** Page 1/1

```
package graphtraversals;

import graphs.UGraph;
import graphs.UGraphAdj;
import graphtraversals.Components.V;

public class ComponentsTest {

    public static void main(String[] args) {
        Components components = new Components();
        UGraph<V,Object> G = new UGraphAdj<V,Object>();
        V v0 = components.new V(); G.addVertex( v0 );
        V v1 = components.new V(); G.addVertex( v1 );
        V v2 = components.new V(); G.addVertex( v2 );
        V v3 = components.new V(); G.addVertex( v3 );
        V v4 = components.new V(); G.addVertex( v4 );
        V v5 = components.new V(); G.addVertex( v5 );

        Object ob = new Object();
        G.addEdge( v0, v1, ob );
        G.addEdge( v0, v3, ob );
        G.addEdge( v1, v4, ob );
        G.addEdge( v4, v3, ob );
        G.addEdge( v2, v5, ob );

        components.components( G );
        for( V u : G ) {
            System.out.println( u );
        }
    }

}
```

Aug 25, 14 12:58 **DepthFirst.java** Page 1/1

```
package graphtraversals;

import graphs.Graph;
import graphs.Graph.AdjacentEdge;

public class DepthFirst<V extends DepthFirstVertex,E> {

    public enum Colour{ White, Grey, Black }

    public DepthFirst( ) {
        super();
    }

    private int time;

    public void DepthFirstSearch( Graph<V,E> G ) {
        /* Initialize all the vertices of the graph to be
                undiscovered and have no parent. */
        for( V u : G ) {
            u.colour = Colour.White;
            u.parent = null;
        }
        /* Search through all vertices of the graph and visit
                any that have not yet been discovered. */
        time = 0;
        for( V u : G ) {
            if( u.colour == Colour.White ) {
                visit( G, u );
            }
        }
    }
    private void visit( Graph<V,E> G, V u ) {
        u.colour = Colour.Grey;
        time++;
        u.discovery = time;
        /* Visit all vertices adjacent to u that have not been
         * discovered (still White).
         */
        for( AdjacentEdge<V,E> e : G.adjacent(u) ) {
            V v = e.target;
            if( v.colour == Colour.White ) {
                v.parent = u;
                visit( G, v );
            }
            v.colour = Colour.Black;
            time++;
            u.finish = time;
        }
    }
}
```

Aug 25, 14 12:58 **DepthFirstTest.java** Page 1/1

```
package graphtraversals;

import graphs.DGraphAdj;
import graphs.Graph;

public class DepthFirstTest {

    public static void main(String[] args) {
        DepthFirst<DepthFirstVertex,Object> df =
                new DepthFirst<DepthFirstVertex,Object>();
        Graph<DepthFirstVertex,Object> G =
                new DGraphAdj<DepthFirstVertex,Object>();
        DepthFirstVertex v0 = new DepthFirstVertex(); G.addVertex( v0 );
        DepthFirstVertex v1 = new DepthFirstVertex(); G.addVertex( v1 );
        DepthFirstVertex v2 = new DepthFirstVertex(); G.addVertex( v2 );
        DepthFirstVertex v3 = new DepthFirstVertex(); G.addVertex( v3 );
        DepthFirstVertex v4 = new DepthFirstVertex(); G.addVertex( v4 );
        DepthFirstVertex v5 = new DepthFirstVertex(); G.addVertex( v5 );

        Object ob = new Object();
        G.addEdge( v0, v1, ob );
        G.addEdge( v0, v3, ob );
        G.addEdge( v1, v4, ob );
        G.addEdge( v3, v1, ob );
        G.addEdge( v4, v3, ob );
        G.addEdge( v2, v4, ob );
        G.addEdge( v2, v5, ob );
        G.addEdge( v5, v5, ob );

        df.DepthFirstSearch( G );
        for( DepthFirstVertex u : G ) {
            System.out.println( u );
        }
    }

}
```

Aug 25, 14 13:13 **OutDegree.java** Page 1/1

```
package graphtraversals;

import java.util.HashMap;
import java.util.Map;

import graphs.Graph;
import graphs.Vertex;
import graphs.Graph.AdjacentEdge;

public class OutDegree<V extends Vertex, E> {

    Map<V,Integer> outDegree(Graph<V,E> G) {
        Map<V,Integer> OD = new HashMap<V,Integer>(G.size());
        for (V u : G) {
            for( AdjacentEdge<V,E> v : G.adjacent(u) ) {
                OD.put(u, OD.get(u)+1);
            }
        }
        return OD;
    }


}
```

```
package graphtraversals;

import graphs.DGraph;
import graphs.Vertex;
import graphs.Graph.AdjacentEdge;

import java.util.List;
import java.util.LinkedList;

public class Topological {

    private enum Colour{White, Grey, Black};

    private class V extends Vertex {
        Colour colour;
    }

    public List<V> ord;
    public boolean cyclic;

    public Topological() {
        super();
    }

    public void order( DGraph<V,Object> G ) {
        cyclic = false;
        ord = new LinkedList<V>();
        for( V u : G ) {
            u.colour = Colour.White;
        }
        for( V u : G ) {
            if( u.colour == Colour.White ) {
                visit( G, u );
            }
        }
    }

    private void visit( DGraph<V,Object> G, V u ) {
        /* for all w: G.V . w.colour == Colour.Grey  =>  (w,u) IN G.E+ */
        u.colour = Colour.Grey;
        /* for all w: G.V . w.colour == Colour.Grey  =>  (w,u) IN G.E* */
        for( AdjacentEdge<V,Object> e : G.adjacent(u) ) {
            V v = e.target;
            if( v.colour == Colour.White ) {
                /* for all w: G.V . w.colour == Grey  =>  (w,v) IN G.E+ */
                visit( G, v );
                /* !cyclic => respects(ord, G.E) &&
                 *            G.E*(| {v} |) subset rng(ord) */
            } else if( v.colour == Colour.Grey ) {
                /* (for all w: G.V . w.colour == Colour.Grey => (w,u) IN G.E* )
                   (v,u) IN G.E* && (u,v) IN G.E
                   (v,v) IN G.E+ */
                cyclic = true;
            }
        }
        /* !cyclic => respects(ord, G.E) &&
         *            (for all v:G.V . (u,v) IN G.E => G.E*(| {v} |) subset rng(
ord))
         */
        u.colour = Colour.Black;
        /* All vertices reachable from u have been visited and added to ord
         * so u can now be added to the front of ord.
         */
        ord.add(0, u);
        /* !cyclic => respects(ord, E) &&
         *            G.E*(| {u} |) subset rng(Ord)
         */
    }
}
```

```
package graphtraversals;

import graphs.DGraph;
import graphs.DGraphMatrix;
import graphs.Vertex;

import java.util.Iterator;

final class V extends Vertex {
    int vertexNumber;

    V( int n ) {
        super();
        vertexNumber = n;
    }
}

public class UniversalSink {

    /** Determine if graph G has a universal sink
     * @param G directed graph
     * @return sink vertex if it exists, else return null
     */
    public static V universalSink(
            DGraph<V,Object> G ) {
        Iterator<V> vertices = G.iterator();
        if( ! vertices.hasNext() ) {
            return null; // empty graph has no sink
        }
        // The initial candidate for a sink is the first vertex
        V s = vertices.next();
        /* In the following predicate vertices is interpreted as
         * the set of vertices yet to be traversed by the iterator.
         * for all w : G − vertices . w != s => !(w sink of G) */
        while( vertices.hasNext() ) {
            // v iterates through the remaining vertices
            V v = vertices.next();
            // for all w : G − (vertices + {v}) . w != s => !(w sink of G)
            if( G.hasEdge(s, v) ) {
                // (s, v) an edge implies s not a sink, but maybe v is,
                // so make v the new candidate to be a sink.
                s = v;
            } else {
                // (s, v) not an edge implies v not a sink, but s may still
                // be a sink, so leave s as the candidate.
            }
            // for all w : G − vertices . w != s => !(w sink of G)
            // System.out.println( " s " + s + " v = " + v );
        }
        // for all w : G . w != s => !(w sink of G)
        // The only possible sink is s but we need to check whether it is.
        for( V w : G ) {
            /* s cannot be a sink if either
             *   there is an edge with source s (and destination w) or
             *   w is a vertex other than s and there is no edge from w to s
             */
            if( G.hasEdge(s, w) || (w != s && !G.hasEdge(w, s)) ) {
                return null;
            }
        }
        // for all w : G . !G.hasEdge(s, w) && (w = s || G.hasEdge(w, s))
        // s is a sink of G
```



```
        return s;
    }

    public static void main(String[] args) {
        DGraph<V,Object> G =
                new DGraphMatrix<V,Object>(6);
        V v0 = new V(0); G.addVertex( v0 );
        V v1 = new V(1); G.addVertex( v1 );
        V v2 = new V(2); G.addVertex( v2 );
        V v3 = new V(3); G.addVertex( v3 );
        V v4 = new V(4); G.addVertex( v4 );
        V v5 = new V(5); G.addVertex( v5 );
        Object ob = new Object();
        G.addEdge( v0, v2, ob );
        G.addEdge( v1, v2, ob );
        G.addEdge( v3, v2, ob );
        G.addEdge( v4, v2, ob );
        G.addEdge( v5, v2, ob );
        G.addEdge( v0, v3, ob );
        G.addEdge( v1, v3, ob );
        G.addEdge( v4, v3, ob );
        V sink = universalSink( G );
        System.out.println( (sink == null ? "No sink for graph" :
                                            "Sink is " + sink) );
    }
}
```