GrALoG Ford-Fulkerson Algorithm Plugin

Design and Plan Document

---

1. Executive Summary

This document outlines the design and implementation plan for a GrALoG plugin that visualizes the Ford-Fulkerson algorithm for computing maximum flow in network graphs. The plugin will provide step-by-step interactive visualization of the algorithm's execution, including augmenting path discovery, residual graph updates, and flow calculations.

Target Platform: GrALoG (Graphs, Algorithms, Logic and Games)
Algorithm: Ford-Fulkerson Maximum Flow Algorithm
Implementation Language: Java
Primary Libraries: JGraphT (graph data structures), JGraph (visualization)

---

2. Background

2.1 GrALoG Architecture

GrALoG is a Java-based educational tool for visualizing graph algorithms. Key characteristics:

- **Core Library:** Provides GUI for displaying and editing graphs
- **Plugin Interface:** Allows extension through plugins that provide:
  - Additional graph structure types
  - Algorithm implementations
  - Custom generators
- **Graph Libraries:**
  - JGraphT for graph data structures
  - JGraph for visualization and layout
- **Build System:** Gradle-based build system
- **Target Audience:** Research and education in logic, games, and graph algorithms

2.2 Ford-Fulkerson Algorithm

The Ford-Fulkerson method computes the maximum flow from a source to a sink in a flow network:

Key Concepts:

- **Flow Network:** Directed graph with capacity constraints on edges
- **Augmenting Path:** Path from source to sink in residual graph with available capacity
- **Residual Graph:** Network showing remaining capacity on edges
- **Residual Capacity:** For edge (u,v): c(u,v) - f(u,v)
- **Bottleneck:** Minimum residual capacity along a path

Algorithm Steps:

1. Initialize all flows to zero
2. While there exists an augmenting path from source to sink:
   - Find an augmenting path P (using BFS or DFS)
   - Calculate bottleneck capacity (minimum residual capacity on path)
   - Augment flow along path P by bottleneck value
   - Update residual graph
3. Return total flow

Visualization Requirements:

- Show current flow vs. capacity on edges (e.g., "5/10")
- Highlight augmenting paths
- Display residual graph alongside flow network
- Animate flow updates
- Show bottleneck calculation
- Track total flow value

---

3. System Architecture

3.1 Plugin Components

```
gralog-fordfulkerson/
├── src/
│   └── main/
│       └── java/
│           └── gralog/
│               └── fordfulkerson/
│                   ├── structure/
│                   │   ├── FlowNetwork.java
│                   │   ├── FlowEdge.java
│                   │   └── FlowVertex.java
│                   ├── algorithm/
│                   │   ├── FordFulkersonAlgorithm.java
│                   │   ├── EdmondsKarpAlgorithm.java
│                   │   └── AugmentingPathFinder.java
│                   ├── visualization/
│                   │   ├── FlowNetworkRenderer.java
│                   │   ├── ResidualGraphRenderer.java
│                   │   ├── AlgorithmAnimator.java
│                   │   └── StepController.java
│                   ├── ui/
│                   │   ├── FordFulkersonDialog.java
│                   │   ├── ControlPanel.java
│                   │   └── StepVisualizationPanel.java
│                   └── FordFulkersonPlugin.java
├── resources/
│   ├── plugin.xml
│   └── icons/
└── build.gradle
```

3.2 Class Diagram

```
┌─────────────────────────────┐
│  GrALoG Core API            │
│  (Plugin Interface)         │
└──────────────┬──────────────┘
               │ implements
               ▼
┌─────────────────────────────┐
│ FordFulkersonPlugin         │
│ ─────────────────────────── │
│ + getName()                 │
│ + getDescription()          │
│ + createAlgorithm()         │
└──────────────┬──────────────┘
               │ creates
               ▼
┌─────────────────────────────┐     ┌─────────────────────────────┐
│ FlowNetwork                 │◄────│ FlowEdge                    │
│ ─────────────────────────── │uses │ ─────────────────────────── │
│ - vertices: Set             │     │ - source: Vertex            │
│ - edges: Set                │     │ - target: Vertex            │
│ - source: Vertex            │     │ - capacity: int             │
│ - sink: Vertex              │     │ - flow: int                 │
│ + addEdge()                 │     │ + getResidualCapacity()     │
│ + getResidualGraph()        │     │ + augmentFlow()             │
└──────────────┬──────────────┘     └─────────────────────────────┘
               │
               ▼
┌─────────────────────────────┐
│ FordFulkersonAlgorithm      │
│ ─────────────────────────── │
│ - network: FlowNetwork      │
│ - maxFlow: int              │
│ - steps: List<Step>         │
│ + execute()                 │
│ + nextStep()                │
│ + previousStep()            │
│ + reset()                   │
└──────────────┬──────────────┘
               │
               ▼
┌─────────────────────────────┐
│ AlgorithmAnimator           │
│ ─────────────────────────── │
│ + visualizeStep()           │
│ + highlightPath()           │
│ + updateFlowDisplay()       │
│ + renderResidualGraph()     │
└─────────────────────────────┘
```

4. Detailed Design

4.1 Data Structures

4.1.1 FlowNetwork

```java
public class FlowNetwork extends DirectedGraph<FlowVertex, FlowEdge> {
    private FlowVertex source;
    private FlowVertex sink;
    private Map<FlowEdge, Integer> capacity;
    private Map<FlowEdge, Integer> flow;

    public FlowNetwork createResidualGraph() {
        // Create residual graph with forward and backward edges
    }

    public int getTotalFlow() {
        // Sum flow leaving source
    }
}
```

4.1.2 FlowEdge

```java
public class FlowEdge extends DefaultEdge {
    private int capacity;
    private int flow;
    private boolean isResidualEdge; // For residual graph edges

    public int getResidualCapacity() {
        return capacity - flow;
    }

    public void augmentFlow(int deltaFlow) {
        flow += deltaFlow;
    }

    public String getDisplayLabel() {
        return flow + "/" + capacity;
    }
}
```

4.1.3 AlgorithmState

```java
public class AlgorithmState {
    public enum StepType {
        INITIALIZATION,
        PATH_SEARCH,
        PATH_FOUND,
        NO_PATH_FOUND,
        BOTTLENECK_CALCULATION,
        FLOW_AUGMENTATION,
        COMPLETED
    }

    private StepType stepType;
    private List<FlowEdge> currentPath;
    private int bottleneckValue;
    private int currentMaxFlow;
    private FlowNetwork residualGraph;
    private String description;
}
```

4.2 Algorithm Implementation

4.2.1 Main Algorithm Loop

java

```java
public class FordFulkersonAlgorithm implements AlgorithmInterface {

    private FlowNetwork network;
    private List<AlgorithmState> executionSteps;
    private int currentStepIndex;

    public void execute() {
        // Step 1: Initialize
        initializeFlow();
        recordStep(StepType.INITIALIZATION);

        while (true) {
            // Step 2: Find augmenting path
            List<FlowEdge> path = findAugmentingPath();

            if (path == null) {
                recordStep(StepType.NO_PATH_FOUND);
                recordStep(StepType.COMPLETED);
                break;
            }

            recordStep(StepType.PATH_FOUND, path);

            // Step 3: Calculate bottleneck
            int bottleneck = calculateBottleneck(path);
            recordStep(StepType.BOTTLENECK_CALCULATION, path, bottleneck);

            // Step 4: Augment flow
            augmentFlow(path, bottleneck);
            recordStep(StepType.FLOW_AUGMENTATION, path, bottleneck);
        }
    }

    private List<FlowEdge> findAugmentingPath() {
        // Use BFS (Edmonds-Karp) or DFS
        return bfsAugmentingPath(network.getSource(), network.getSink());
    }

    private List<FlowEdge> bfsAugmentingPath(FlowVertex source, FlowVertex sink) {
        Queue<FlowVertex> queue = new LinkedList<>();
        Map<FlowVertex, FlowEdge> parent = new HashMap<>();
        Set<FlowVertex> visited = new HashSet<>();

        queue.add(source);
        visited.add(source);

        while (!queue.isEmpty()) {
```

```java
            FlowVertex current = queue.poll();

            if (current.equals(sink)) {
                return reconstructPath(parent, source, sink);
            }

            for (FlowEdge edge : network.outgoingEdgesOf(current)) {
                if (edge.getResidualCapacity() > 0) {
                    FlowVertex target = edge.getTarget();
                    if (!visited.contains(target)) {
                        visited.add(target);
                        parent.put(target, edge);
                        queue.add(target);
                    }
                }
            }
        }

        return null; // No path found
    }
}
```

4.3 Visualization Components

4.3.1 Step-by-Step Display

```java
public class StepVisualizationPanel extends JPanel {

    private FlowNetworkRenderer networkRenderer;
    private ResidualGraphRenderer residualRenderer;
    private ControlPanel controlPanel;

    public void visualizeStep(AlgorithmState state) {
        switch (state.getStepType()) {
            case INITIALIZATION:
                showInitialNetwork();
                break;

            case PATH_FOUND:
                highlightAugmentingPath(state.getCurrentPath());
                updateResidualGraph(state.getResidualGraph());
                break;

            case BOTTLENECK_CALCULATION:
                showBottleneckCalculation(state);
                break;

            case FLOW_AUGMENTATION:
                animateFlowUpdate(state.getCurrentPath(),
                                 state.getBottleneckValue());
                break;

            case COMPLETED:
                showFinalResult(state.getCurrentMaxFlow());
                break;
        }
    }
}
```

4.3.2 Edge Rendering

```java
public class FlowEdgeRenderer extends DefaultEdgeRenderer {

    @Override
    public void render(Graphics2D g, FlowEdge edge) {
        // Draw edge with appropriate color
        Color edgeColor = getEdgeColor(edge);

        // Draw edge label showing flow/capacity
        String label = edge.getFlow() + "/" + edge.getCapacity();

        // Highlight if part of current augmenting path
        if (isInCurrentPath(edge)) {
            g.setStroke(new BasicStroke(3.0f));
            g.setColor(Color.GREEN);
        } else if (edge.getResidualCapacity() == 0) {
            g.setColor(Color.RED);
        } else {
            g.setColor(Color.BLACK);
        }

        // Render edge and label
        drawArrow(g, edge.getSource(), edge.getTarget());
        drawLabel(g, label, getMidpoint(edge));
    }

    private Color getEdgeColor(FlowEdge edge) {
        float saturation = (float) edge.getFlow() / edge.getCapacity();
        return Color.getHSBColor(0.33f, saturation, 1.0f);
    }
}
```

4.4 User Interface

4.4.1 Control Panel

```java
public class ControlPanel extends JPanel {

    private JButton nextButton;
    private JButton previousButton;
    private JButton playButton;
    private JButton resetButton;
    private JSlider speedSlider;
    private JLabel stepDescription;
    private JLabel flowValueLabel;

    public ControlPanel(AlgorithmController controller) {
        initializeComponents();
        setupEventHandlers(controller);
    }

    private void setupEventHandlers(AlgorithmController controller) {
        nextButton.addActionListener(e -> controller.nextStep());
        previousButton.addActionListener(e -> controller.previousStep());
        playButton.addActionListener(e -> controller.toggleAutoPlay());
        resetButton.addActionListener(e -> controller.reset());
    }
}
```

4.4.2 Configuration Dialog

```java
public class FordFulkersonDialog extends JDialog {

    private JComboBox<PathFindingStrategy> strategySelector;
    private JSpinner sourceSelector;
    private JSpinner sinkSelector;
    private JCheckBox showResidualGraphCheck;
    private JCheckBox animateCheck;

    public AlgorithmConfiguration getConfiguration() {
        return new AlgorithmConfiguration()
            .withStrategy(strategySelector.getSelectedItem())
            .withSource(sourceSelector.getValue())
            .withSink(sinkSelector.getValue())
            .withResidualGraphVisible(showResidualGraphCheck.isSelected())
            .withAnimation(animateCheck.isSelected());
    }
}
```

5. Algorithm Visualization States

5.1 State Transitions

```
[INITIALIZATION] → Initial graph with zero flow
        ↓
[PATH_SEARCH] → Searching for augmenting path (BFS/DFS animation)
        ↓
[PATH_FOUND] → Highlight discovered path
        ↓
[BOTTLENECK_CALCULATION] → Show minimum capacity along path
        ↓
[FLOW_AUGMENTATION] → Animate flow increase on path edges
        ↓
[UPDATE_RESIDUAL] → Update residual graph display
        ↓
[PATH_SEARCH] → (loop back) or → [NO_PATH_FOUND]
                                 ↓
                          [COMPLETED] → Show max flow result
```

5.2 Visual Elements by State

State	Network Display	Residual Graph	Info Panel
INITIALIZATION	All edges 0/capacity	Same as network	"Starting Ford-Fulkerson..."
PATH_SEARCH	Current flows	Updated residual	"Searching for augmenting path..."
PATH_FOUND	Path highlighted green	Path shown	"Found path: s→v1→v2→t"
BOTTLENECK_CALCULATION	Edges labeled with residual	Minimum edge highlighted	"Bottleneck = min(5,3,8) = 3"
FLOW_AUGMENTATION	Animated flow increase	-	"Augmenting flow by 3 units"
COMPLETED	Final flows shown	-	"Maximum flow = 23"

---

6. Implementation Plan

Phase 1: Core Data Structures (Week 1-2)

Deliverables:

- [ ] FlowNetwork class implementation
- [ ] FlowEdge class with flow/capacity tracking
- [ ] FlowVertex class
- [ ] Unit tests for data structures
- [ ] Integration with JGraphT

Tasks:

1. Create base graph structures
2. Implement residual graph generation
3. Add capacity and flow tracking
4. Write comprehensive unit tests
5. Document API

Phase 2: Algorithm Implementation (Week 3-4)

Deliverables:

- [ ] Ford-Fulkerson algorithm core logic
- [ ] BFS-based path finding (Edmonds-Karp)
- [ ] DFS-based path finding option
- [ ] State tracking for visualization
- [ ] Algorithm correctness tests

Tasks:

1. Implement augmenting path search
2. Implement flow augmentation logic
3. Add state recording for each step
4. Create algorithm test suite
5. Validate against known max-flow examples

Phase 3: Visualization Components (Week 5-6)

Deliverables:

- [ ] Custom edge renderer for flow/capacity display
- [ ] Path highlighting system
- [ ] Residual graph renderer
- [ ] Step controller with forward/backward navigation
- [ ] Animation system

Tasks:

1. Extend JGraph renderers
2. Implement edge coloring based on saturation
3. Create animation framework
4. Add transition effects
5. Implement step navigation

Phase 4: User Interface (Week 7-8)

Deliverables:

- [ ] Configuration dialog
- [ ] Control panel with playback controls
- [ ] Step information display
- [ ] Source/sink selection interface
- [ ] Settings panel

Tasks:

1. Design UI mockups
2. Implement Swing components
3. Wire up event handlers
4. Add keyboard shortcuts
5. Implement preferences persistence

Phase 5: Plugin Integration (Week 9)

Deliverables:

- [ ] Plugin descriptor (plugin.xml)
- [ ] GrALoG plugin interface implementation
- [ ] Menu integration
- [ ] Toolbar integration
- [ ] Build configuration

Tasks:

1. Study GrALoG plugin API
2. Implement plugin lifecycle methods
3. Register algorithm with GrALoG
4. Add to application menus
5. Configure Gradle build

Phase 6: Testing & Documentation (Week 10)

Deliverables:

- [ ] Comprehensive test suite
- [ ] User manual
- [ ] API documentation
- [ ] Example graphs
- [ ] Installation guide

Tasks:

1. Integration testing with GrALoG
2. User acceptance testing
3. Write user documentation
4. Create tutorial examples
5. Generate JavaDoc
6. Prepare release package

Phase 7: Polish & Release (Week 11)

Deliverables:

- [ ] Bug fixes
- [ ] Performance optimization
- [ ] UI refinement
- [ ] Release package
- [ ] GitHub repository

Tasks:

1. Address feedback from testing
2. Optimize rendering performance
3. Polish animations
4. Create release package
5. Publish on GitHub/SourceForge

---

7. Technical Specifications

7.1 Dependencies

```gradle
dependencies {
    // GrALoG Core
    implementation 'gralog:gralog-core:3.0+'

    // Graph Libraries
    implementation 'org.jgrapht:jgrapht-core:1.5.1'
    implementation 'org.jgrapht:jgrapht-io:1.5.1'

    // UI Components
    implementation 'com.jgraph:jgraph:5.13.0.0'

    // Testing
    testImplementation 'junit:junit:4.13.2'
    testImplementation 'org.mockito:mockito-core:4.6.1'

    // Logging
    implementation 'org.slf4j:slf4j-api:1.7.36'
}
```

7.2 Plugin Descriptor (plugin.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
    <name>Ford-Fulkerson Maximum Flow</name>
    <version>1.0.0</version>
    <description>
        Visualizes the Ford-Fulkerson algorithm for computing maximum flow in network graphs.
        Includes step-by-step execution, residual graph display, and path highlighting.
    </description>

    <author>Your Name</author>
    <requires>
        <gralog version="3.0+"/>
    </requires>

    <structure>
        <name>Flow Network</name>
        <class>gralog.fordfulkerson.structure.FlowNetwork</class>
    </structure>

    <algorithm>
        <name>Ford-Fulkerson (BFS)</name>
        <class>gralog.fordfulkerson.algorithm.EdmondsKarpAlgorithm</class>
        <structure>Flow Network</structure>
    </algorithm>

    <algorithm>
        <name>Ford-Fulkerson (DFS)</name>
        <class>gralog.fordfulkerson.algorithm.FordFulkersonAlgorithm</class>
        <structure>Flow Network</structure>
    </algorithm>
</plugin>
```

7.3 Build Configuration (build.gradle)

```gradle
plugins {
    id 'java'
    id 'application'
}

group = 'gralog.plugins'
version = '1.0.0'

sourceCompatibility = JavaVersion.VERSION_11
targetCompatibility = JavaVersion.VERSION_11

repositories {
    mavenCentral()
    maven {
        url = 'https://oss.sonatype.org/content/repositories/snapshots'
    }
}

dependencies {
    // Dependencies listed in 7.1
}

jar {
    manifest {
        attributes(
            'Plugin-Name': 'Ford-Fulkerson',
            'Plugin-Version': version,
            'Plugin-Class': 'gralog.fordfulkerson.FordFulkersonPlugin'
        )
    }

    from {
        configurations.runtimeClasspath.collect {
            it.isDirectory() ? it : zipTree(it)
        }
    }
}

task copyToGralog(type: Copy, dependsOn: jar) {
    from jar
    into "$System.env.GRALOG_HOME/plugins"
}
```

---

8. User Interface Mockups

8.1 Main Algorithm View

```
┌─────────────────────────────────────────────────────────────────┐
│ GrALoG - Ford-Fulkerson Algorithm                        [X] │
├─────────────────────────────────────────────────────────────────┤
│ File  Edit  View  Algorithm  Help                                │
├─────────────────────────────────────────────────────────────────┤
│ [◀ Prev]  [▶ Next]  [▶▶ Play]  [↻ Reset]    Speed: [░░░░░ ] │
├──────────────────────────────┬──────────────────────────────────┤
│                  │                          │
│  Flow Network        │      Residual Graph              │
│                  │                          │
│    (s)             │          (s)                     │
│   /  \            │         /  \                     │
│ 5/10  3/8          │       5     5                    │
│  /    \           │      /       \                   │
│ (v1)──2/5──(v2)      │    (v1)──3──(v2)                 │
│  \    /           │      \       /                   │
│ 4/6  7/10          │       2     3                    │
│   \  /            │        \   /                     │
│    (t)             │         (t)                      │
│                  │                          │
│ Current Flow: 12     │  Augmenting Path:                │
│ Max Flow: ???        │  s → v1 → v2 → t                 │
│                  │   Bottleneck: 2                  │
└──────────────────────────────┴──────────────────────────────────┘
│ Step 5 of 12: Augmenting flow along path s→v1→v2→t by 2    │
└─────────────────────────────────────────────────────────────────┘
```

8.2 Configuration Dialog

```
┌─────────────────────────────────────────┐
│ Ford-Fulkerson Configuration        [X] │
├─────────────────────────────────────────┤
│                                         │
│ Source Vertex:  [Dropdown: s        ▼] │
│ Sink Vertex:    [Dropdown: t        ▼] │
│                                         │
│ Path Finding Strategy:                  │
│  ○ BFS (Edmonds-Karp - guaranteed O(V·E²))│
│  ● DFS (may be slower)                  │
│                                         │
│ Visualization Options:                  │
│  ☑ Show residual graph                  │
│  ☑ Animate transitions                  │
│  ☑ Highlight augmenting paths           │
│  ☑ Show bottleneck calculation          │
│                                         │
│ Animation Speed: [▓▓▓▓▓▓▓▓░░░░░░░] Fast    │
│                                         │
│      [ Cancel ]    [  Start  ]          │
└─────────────────────────────────────────┘
```

---

9. Testing Strategy

9.1 Unit Tests

FlowNetwork Tests:

- Test edge addition/removal
- Test source/sink assignment
- Test residual graph generation
- Test flow conservation constraints

Algorithm Tests:

- Test on simple graphs (2-3 nodes)
- Test on graphs with multiple paths
- Test on graphs with cycles
- Test edge cases (no path, zero capacity)
- Validate against known solutions

Example Test Cases:

java

```java
@Test
public void testSimpleMaxFlow() {
    FlowNetwork network = createSimpleNetwork();
    // Graph: s --[10]--> v1 --[10]--> t
    //        s --[10]--> v2 --[10]--> t

    FordFulkersonAlgorithm algo = new FordFulkersonAlgorithm(network);
    int maxFlow = algo.execute();

    assertEquals(20, maxFlow);
}

@Test
public void testBottleneckCalculation() {
    FlowNetwork network = createBottleneckNetwork();
    // Graph: s --[10]--> v1 --[5]--> v2 --[10]--> t

    FordFulkersonAlgorithm algo = new FordFulkersonAlgorithm(network);
    int maxFlow = algo.execute();

    assertEquals(5, maxFlow); // Limited by middle edge
}
```

9.2 Integration Tests

- Test plugin loading in GrALoG
- Test UI component integration
- Test serialization/deserialization
- Test with various graph sizes (10, 100, 1000 nodes)
- Test memory usage with large graphs

9.3 Performance Benchmarks

Graph Size	Edges	Expected Time (BFS)	Expected Time (DFS)
Small (10 nodes)	20	< 100ms	< 100ms
Medium (100 nodes)	500	< 1s	< 5s
Large (1000 nodes)	5000	< 10s	< 60s

---

10. Example Usage Scenarios

10.1 Educational Scenario: Teaching Maximum Flow

Instructor Workflow:

1. Open GrALoG and create a new Flow Network
2. Add vertices and edges representing a supply network
3. Set capacities on edges
4. Select source and sink vertices
5. Run Ford-Fulkerson algorithm
6. Step through algorithm showing:
   - How augmenting paths are found
   - How bottleneck is calculated
   - How flow is updated
   - Why algorithm terminates
7. Show final maximum flow value
8. Demonstrate min-cut theorem by highlighting saturated edges

10.2 Research Scenario: Comparing Path-Finding Strategies

Researcher Workflow:

1. Load test graph from file
2. Configure algorithm with BFS strategy
3. Run and record execution steps and time
4. Reset algorithm
5. Configure with DFS strategy
6. Run and compare performance
7. Export results for analysis

10.3 Student Exercise: Hands-On Learning

Student Workflow:

1. Given a network graph problem
2. Manually predict maximum flow
3. Run algorithm step-by-step
4. Compare their predicted flow with algorithm result
5. Review each step to understand where their prediction differed
6. Modify graph and retry

---

11. Future Enhancements

11.1 Short-term (Post-v1.0)

- [ ] Support for Dinic's algorithm (blocking flows)
- [ ] Support for Push-Relabel algorithm
- [ ] Graph import/export in various formats (GraphML, DOT)
- [ ] Comparison mode: run multiple algorithms side-by-side
- [ ] Performance statistics and profiling

11.2 Medium-term

- [ ] 3D visualization option
- [ ] Interactive graph editing during algorithm execution
- [ ] Automated test case generation
- [ ] Min-cut visualization
- [ ] Support for multiple sources/sinks

11.3 Long-term

- [ ] Web-based version using GWT
- [ ] Real-world problem templates (network routing, matching)
- [ ] Integration with online judge systems
- [ ] Machine learning for optimal path selection heuristics
- [ ] Distributed algorithm execution for massive graphs

---

12. Risk Assessment

Risk	Probability	Impact	Mitigation Strategy
GrALoG API changes	Medium	High	Pin specific GrALoG version; maintain compatibility layer
Performance issues with large graphs	High	Medium	Implement progressive rendering; add graph size limits
Animation causing UI lag	Medium	Medium	Use off-screen rendering; optimize render loop
Complex residual graph confusing users	Medium	Low	Add toggle for residual graph; provide tutorial
Memory issues with history tracking	Low	High	Limit history size; implement lazy state reconstruction
Incompatibility with JGraph versions	Low	Medium	Test with multiple JGraph versions; use stable API only

---

13. Success Criteria

13.1 Functional Requirements

- ✓ Correctly computes maximum flow for all valid input graphs
- ✓ Accurately visualizes each step of the algorithm
- ✓ Provides forward/backward step navigation
- ✓ Displays both flow network and residual graph
- ✓ Highlights augmenting paths and bottlenecks
- ✓ Handles graphs up to 1000 nodes efficiently

13.2 Educational Requirements

- ✓ Clear visual representation aids understanding
- ✓ Step descriptions are pedagogically sound
- ✓ Supports self-paced learning
- ✓ Provides meaningful feedback at each step
- ✓ Includes tutorial examples

13.3 Technical Requirements

- ✓ Integrates seamlessly with GrALoG
- ✓ Follows GrALoG coding standards
- ✓ Comprehensive test coverage (>80%)
- ✓ Documented API with JavaDoc
- ✓ No memory leaks or performance regressions

---

14. Documentation Deliverables

14.1 User Documentation

- Quick start guide
- Feature walkthrough with screenshots
- Tutorial: Understanding Ford-Fulkerson
- FAQ and troubleshooting
- Keyboard shortcuts reference

14.2 Developer Documentation

- Architecture overview
- API reference (JavaDoc)
- Plugin development guide
- Contributing guidelines
- Build and deployment instructions

14.3 Educational Materials

- Sample problems with solutions
- Classroom exercises
- Assessment questions
- Video tutorials (optional)
- Presentation slides

---

15. Deployment and Distribution

15.1 Build Artifacts

- `gralog-fordfulkerson-1.0.0.jar` - Plugin JAR
- `gralog-fordfulkerson-1.0.0-sources.jar` - Source code
- `gralog-fordfulkerson-1.0.0-javadoc.jar` - API documentation

15.2 Distribution Channels

- GitHub Releases
- GrALoG plugin repository
- Maven Central (optional)

15.3 Installation Instructions

```bash
# Method 1: Manual Installation
1. Download gralog-fordfulkerson-1.0.0.jar
2. Copy to $GRALOG_HOME/plugins/
3. Restart GrALoG

# Method 2: Build from Source
git clone https://github.com/yourusername/gralog-fordfulkerson
cd gralog-fordfulkerson
./gradlew build copyToGralog

# Method 3: Gradle Task
./gradlew installPlugin
```

16. Contact and Support

Project Lead: [Your Name]
Email: [your.email@example.com]
GitHub: https://github.com/yourusername/gralog-fordfulkerson
Issue Tracker: https://github.com/yourusername/gralog-fordfulkerson/issues
Documentation: https://github.com/yourusername/gralog-fordfulkerson/wiki

Support Channels:

- GitHub Issues for bug reports
- GitHub Discussions for questions
- Email for private inquiries

---

17. References

1. Ford, L. R. & Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics.
2. Edmonds, J. & Karp, R. M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems".
3. Cormen, T. H., et al. (2009). "Introduction to Algorithms" (3rd ed.), Chapter 26: Maximum Flow.
4. GrALoG Documentation: http://www.cs.ox.ac.uk/stephan.kreutzer/gralog/
5. JGraphT Documentation: https://jgrapht.org/
6. JGraph User Guide: http://www.jgraph.com/

Appendices

Appendix A: Sample Input Graph Format

```json
{
  "vertices": [
    {"id": "s", "label": "Source"},
    {"id": "v1", "label": "V1"},
    {"id": "v2", "label": "V2"},
    {"id": "t", "label": "Sink"}
  ],
  "edges": [
    {"source": "s", "target": "v1", "capacity": 10},
    {"source": "s", "target": "v2", "capacity": 8},
    {"source": "v1", "target": "v2", "capacity": 5},
    {"source": "v1", "target": "t", "capacity": 6},
    {"source": "v2", "target": "t", "capacity": 10}
  ],
  "source": "s",
  "sink": "t"
}
```

Appendix B: Algorithm Pseudocode

```
FordFulkerson(Graph G, Node s, Node t):
  for each edge (u, v) in G.E:
    (u, v).flow = 0

  maxFlow = 0

  while exists path P from s to t in residual graph:
    bottleneck = min{residual_capacity(e) : e in P}

    for each edge (u, v) in P:
      if (u, v) is forward edge:
        (u, v).flow += bottleneck
      else:  // backward edge
        (v, u).flow -= bottleneck

    maxFlow += bottleneck

  return maxFlow
```

Appendix C: Complexity Analysis

Time Complexity:

- Ford-Fulkerson with DFS: $O(E \cdot |f^*|)$ where f^* is maximum flow
- Edmonds-Karp (BFS): $O(V \cdot E^2)$

Space Complexity:

- Graph storage: O(V + E)
- Residual graph: O(V + E)
- Path storage: O(V)
- History tracking: O(steps · E)

Document Version: 1.0
Last Updated: November 25, 2025
Status: Initial Design - Ready for Implementation