GrALoG Ford-Fulkerson Algorithm Visualization Plugin

Design and Implementation Plan

Document Version: 1.0
Date: November 26, 2025
Author: Technical Design Document

Executive Summary

This document outlines the design and implementation plan for a GrALoG plugin that provides step-by-step visualization of the Ford-Fulkerson algorithm for computing maximum flow in network flow graphs. The plugin will extend GrALoG's existing graph visualization capabilities to support flow networks with capacities, flow values, and residual graphs, while providing an interactive, educational experience for understanding how the algorithm finds augmenting paths and incrementally builds toward maximum flow.

Key Features

- Interactive flow network graph creation and editing
- Step-by-step visualization of the Ford-Fulkerson algorithm
- Display of residual graph alongside original network
- Augmenting path highlighting and flow updates
- Visual representation of flow/capacity ratios on edges
- Support for both manual stepping and automatic animation
- Min-cut visualization upon algorithm completion

---

1. Project Overview

1.1 Purpose

Create an educational tool integrated into GrALoG that helps students and practitioners understand the Ford-Fulkerson maximum flow algorithm through interactive visualization.

1.2 Target Users

- Computer Science students learning graph algorithms
- Algorithm instructors demonstrating maximum flow concepts
- Researchers analyzing network flow problems
- Anyone studying optimization and network algorithms

1.3 Integration with GrALoG

GrALoG is a Java-based, plugin-extensible framework for visualizing graph structures and algorithms. Our plugin will:

- Define a custom `FlowNetwork` structure extending GrALoG's graph model
- Implement the `FordFulkersonAlgorithm` class that integrates with GrALoG's algorithm execution framework
- Provide custom rendering for flow/capacity visualization
- Support GrALoG's standard interaction patterns

---

2. Algorithm Background

2.1 Ford-Fulkerson Method Overview

The Ford-Fulkerson method is an approach for computing maximum flow in a flow network by:

1. Starting with zero flow on all edges
2. Finding augmenting paths from source to sink in the residual graph
3. Computing bottleneck capacity (minimum residual capacity along the path)
4. Augmenting flow along the path by the bottleneck amount
5. Updating the residual graph
6. Repeating until no augmenting path exists

2.2 Key Concepts for Visualization

Flow Network

- Directed graph $G = (V, E)$
- Source vertex s and sink vertex t
- Capacity function $c(u,v) \geq 0$ for each edge (u,v)
- Flow function $f(u,v)$ satisfying:
  - Capacity constraint: $f(u,v) \leq c(u,v)$
  - Flow conservation: $\Sigma f(u,v) = \Sigma f(v,w)$ for all $v \neq s,t$

Residual Graph

- Contains forward edges with residual capacity $c(u,v) - f(u,v)$
- Contains backward edges with capacity $f(u,v)$
- Used to find augmenting paths

Augmenting Path

- Simple path from s to t in residual graph
- All edges have positive residual capacity
- Bottleneck = minimum residual capacity along path

2.3 Visualization Requirements

- Display original capacities and current flow on edges (e.g., "3/5" = 3 flow, 5 capacity)
- Highlight the current augmenting path being processed
- Show residual graph structure (optional toggle)
- Display backward edges when flow exists
- Update visual state incrementally at each step
- Show final maximum flow value and min-cut

---

3. Technical Architecture

3.1 Technology Stack

- **Language:** Java (JDK 11 or later)
- **Build System:** Gradle
- **Framework:** GrALoG plugin API
- **UI:** Swing/AWT (GrALoG's native UI framework)
- **Testing:** JUnit 4.13+

3.2 Plugin Structure

```
gralog-fordfulkerson-plugin/
├── build.gradle
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── gralog/
│   │   │       └── fordfulkerson/
│   │   │           ├── structure/
│   │   │           │   ├── FlowNetwork.java
│   │   │           │   ├── FlowEdge.java
│   │   │           │   ├── FlowVertex.java
│   │   │           │   └── FlowNetworkRenderer.java
│   │   │           ├── algorithm/
│   │   │           │   ├── FordFulkersonAlgorithm.java
│   │   │           │   ├── AugmentingPathFinder.java
│   │   │           │   ├── ResidualGraph.java
│   │   │           │   └── AlgorithmState.java
│   │   │           ├── ui/
│   │   │           │   ├── FlowNetworkEditor.java
│   │   │           │   ├── AlgorithmControlPanel.java
│   │   │           │   └── FlowStatisticsPanel.java
│   │   │           └── FordFulkersonPlugin.java
│   │   └── resources/
│   │       └── META-INF/
│   │           └── services/
│   │               └── gralog.plugins.Plugin
│   └── test/
│       └── java/
│           └── gralog/
│               └── fordfulkerson/
│                   ├── FordFulkersonAlgorithmTest.java
│                   ├── ResidualGraphTest.java
│                   └── FlowNetworkTest.java
└── README.md
```

3.3 Core Components

3.3.1 FlowNetwork (Structure)

```java
public class FlowNetwork extends Structure {
    private FlowVertex source;
    private FlowVertex sink;
    private Map<FlowEdge, Integer> capacities;
    private Map<FlowEdge, Integer> flows;

    // Methods for network manipulation
    public void setSource(FlowVertex s);
    public void setSink(FlowVertex t);
    public int getCapacity(FlowEdge e);
    public int getFlow(FlowEdge e);
    public int getResidualCapacity(FlowEdge e);
}
```

3.3.2 FlowEdge

```java
public class FlowEdge extends Edge {
    private int capacity;
    private int flow;
    private boolean isBackEdge; // For residual graph representation

    public void setCapacity(int c);
    public void setFlow(int f);
    public int getResidualCapacity();
    public FlowEdge getReverseEdge();
}
```

3.3.3 FordFulkersonAlgorithm

java

```java
public class FordFulkersonAlgorithm extends Algorithm {
    private FlowNetwork network;
    private ResidualGraph residualGraph;
    private AlgorithmState state;
    private List<AlgorithmStep> steps;

    @Override
    public AlgorithmParameters getParameters() {
        // Define algorithm parameters (path-finding method, etc.)
    }

    @Override
    public Object run(Structure s, AlgorithmParameters p) {
        // Main algorithm execution
    }

    public void executeStep() {
        // Execute single step for visualization
    }

    private List<FlowVertex> findAugmentingPath() {
        // BFS/DFS to find augmenting path
    }

    private int computeBottleneck(List<FlowVertex> path) {
        // Find minimum residual capacity
    }

    private void augmentPath(List<FlowVertex> path, int flow) {
        // Update flows along path
    }
}
```

3.3.4 AlgorithmState

```java
public class AlgorithmState {
    public enum Phase {
        INITIALIZATION,
        FINDING_PATH,
        PATH_FOUND,
        AUGMENTING_FLOW,
        UPDATING_RESIDUAL,
        COMPLETED,
        NO_PATH_FOUND
    }

    private Phase currentPhase;
    private List<FlowVertex> currentPath;
    private int currentBottleneck;
    private int totalFlow;
    private Set<FlowVertex> minCutSet;

    // State access and manipulation methods
}
```

3.3.5 ResidualGraph

```java
public class ResidualGraph {
    private FlowNetwork originalNetwork;
    private Map<FlowVertex, List<ResidualEdge>> adjacencyList;

    public void update(FlowNetwork network) {
        // Rebuild residual graph from current flows
    }

    public List<FlowVertex> findPathBFS(FlowVertex s, FlowVertex t) {
        // BFS for augmenting path
    }

    public List<FlowVertex> findPathDFS(FlowVertex s, FlowVertex t) {
        // DFS for augmenting path (alternative)
    }
}
```

3.4 GrALoG Plugin Integration

3.4.1 Plugin Registration

Create `FordFulkersonPlugin.java`:

```java
public class FordFulkersonPlugin extends Plugin {
    @Override
    public void register() {
        // Register FlowNetwork structure
        Structure.registerStructure(FlowNetwork.class);

        // Register Ford-Fulkerson algorithm
        Algorithm.registerAlgorithm(FordFulkersonAlgorithm.class, FlowNetwork.class);

        // Register custom renderer
        Renderer.registerRenderer(FlowNetworkRenderer.class, FlowNetwork.class);
    }
}
```

3.4.2 Service Provider Interface

Create `src/main/resources/META-INF/services/gralog.plugins.Plugin`:

```
gralog.fordfulkerson.FordFulkersonPlugin
```

---

4. Visualization Design

4.1 Graph Rendering

4.1.1 Edge Display

- **Format:** "flow/capacity" label on each edge
  - Example: "3/7" means 3 units of flow on edge with capacity 7
- **Color Coding:**
  - Black: Normal edges with available capacity
  - Blue: Edges at full capacity (flow = capacity)
  - Green: Edges in current augmenting path
  - Red: Backward/residual edges (when showing residual graph)
  - Gray: Edges with zero residual capacity

4.1.2 Vertex Display

- **Source vertex (s):** Double circle with "S" label
- **Sink vertex (t):** Double circle with "T" label
- **Intermediate vertices:** Standard circles
- **Highlight:** Yellow background for vertices in current augmenting path

4.1.3 Layout Options

- Manual positioning (drag and drop)
- Automatic layout using force-directed algorithm
- Hierarchical layout (source at left, sink at right)

4.2 Step-by-Step Visualization

Phase 1: Initialization

```
Display: "Initializing Ford-Fulkerson Algorithm"
State:
- All edge flows set to 0
- Residual graph = original graph
- Maximum flow = 0
```

Phase 2: Finding Augmenting Path

```
Display: "Searching for augmenting path from S to T..."
Visualization:
- Animate BFS/DFS traversal
- Highlight visited vertices in yellow
- Highlight current frontier in orange
- Show "path found" or "no path exists" message
```

Phase 3: Path Found

```
Display: "Augmenting path: S → A → B → T"
Visualization:
- Highlight path edges in green
- Show bottleneck calculation: min(5, 3, 6) = 3
- Display: "Bottleneck capacity: 3"
```

Phase 4: Augmenting Flow

```
Display: "Adding 3 units of flow along path"
Visualization:
- Animate flow increase on each edge in path
- Update edge labels (e.g., "0/5" → "3/5")
- Update total flow counter
```

Phase 5: Updating Residual Graph

```
Display: "Updating residual graph"
Visualization:
- Show decreased forward capacities
- Show increased backward capacities (optional view)
- Prepare for next iteration
```

Phase 6: Completion

```
Display: "Maximum flow found: 15"
Visualization:
- Highlight all edges with flow > 0
- Display min-cut edges
- Show reachable vertices from source in residual graph
```

4.3 User Interface Components

4.3.1 Main Visualization Panel

- Central canvas displaying the flow network
- Zoom and pan controls
- Toggle between original and residual graph view

4.3.2 Algorithm Control Panel

```
[◄ Previous Step] [▶ Next Step] [▶▶ Run to End] [⏸ Pause]
[■ Reset]

Animation Speed: [▁▂▃▄▅▆▇█] (slider)
```

4.3.3 Statistics Panel

```
┌─────────────────────────────────────┐
│ Algorithm Statistics                │
├─────────────────────────────────────┤
│ Current Flow:      12 / 18          │
│ Iterations:        5                │
│ Current Phase:     Finding Path     │
│ Path Length:       3 edges          │
│ Bottleneck:        4 units          │
└─────────────────────────────────────┘
```

4.3.4 Path History Panel

```
Iteration 1: S→A→B→T (flow: +3)
Iteration 2: S→C→D→T (flow: +5)
Iteration 3: S→A→D→T (flow: +2)
...
```

4.3.5 Settings Dialog

- Path finding method: BFS (Edmonds-Karp) or DFS
- Show residual graph: Yes/No
- Show backward edges: Yes/No
- Highlight min-cut: Yes/No
- Animation duration: 500ms - 5000ms

---

5. Implementation Plan

5.1 Phase 1: Core Structure (Week 1-2)

Objective: Establish basic flow network structure and rendering

Tasks:

1. Set up Gradle project and plugin structure
    - Configure build.gradle with GrALoG dependencies
    - Create directory structure
    - Set up testing framework
2. Implement FlowNetwork classes
    - FlowVertex class extending GrALoG's Vertex
    - FlowEdge class with capacity and flow properties
    - FlowNetwork class extending GrALoG's Structure
    - Source/sink designation methods
3. Implement FlowNetworkRenderer
    - Custom edge rendering with flow/capacity labels
    - Color coding for different edge states
    - Vertex highlighting for source/sink
4. Create basic editor functionality
    - Add/remove vertices
    - Add/remove edges with capacity specification
    - Designate source and sink
    - Manual layout adjustment

Deliverables:

- Working FlowNetwork structure
- Editable flow networks in GrALoG
- Basic visualization (no algorithm yet)

5.2 Phase 2: Algorithm Core (Week 3-4)

Objective: Implement Ford-Fulkerson algorithm logic

Tasks:

1. Implement ResidualGraph class
    - Build residual graph from flow network
    - Update method after flow augmentation
    - Support for backward edges
2. Implement AugmentingPathFinder
    - BFS implementation (Edmonds-Karp)
    - DFS implementation (alternative)
    - Path reconstruction from parent pointers
3. Implement FordFulkersonAlgorithm
    - Main algorithm loop
    - Bottleneck calculation
    - Flow augmentation logic
    - Termination detection
4. Implement AlgorithmState
    - Track current phase
    - Store current path and bottleneck
    - Maintain total flow
    - Store algorithm history

Deliverables:

- Working Ford-Fulkerson algorithm
- Correct maximum flow computation
- Unit tests for algorithm correctness

5.3 Phase 3: Step-by-Step Visualization (Week 5-6)

Objective: Add interactive stepping and animation

Tasks:

1. Break algorithm into discrete steps
    - Define step boundaries (state transitions)
    - Create step objects with visualization data
    - Implement step-forward and step-backward
2. Implement AlgorithmControlPanel
    - Step buttons (previous, next, run, reset)
    - Animation speed slider
    - Pause/resume functionality
3. Add visual highlighting
    - Animate path search (BFS/DFS traversal)
    - Highlight current augmenting path
    - Animate flow updates
    - Show phase transitions
4. Implement FlowStatisticsPanel
    - Display current flow value
    - Show iteration count
    - Display current phase
    - Show path and bottleneck info

Deliverables:

- Interactive step-by-step execution
- Visual animations for each phase
- Control panel for user interaction

5.4 Phase 4: Advanced Features (Week 7-8)

Objective: Add residual graph view and min-cut visualization

Tasks:

1. Implement residual graph visualization
    - Toggle between original and residual view
    - Display backward edges
    - Show residual capacities
2. Implement min-cut detection
    - Identify reachable vertices from source
    - Highlight min-cut edges
    - Display cut capacity
3. Add path history panel
    - Track all augmenting paths found
    - Display path sequence and flow added
    - Allow clicking to replay specific iteration
4. Enhanced settings
    - Path-finding method selection
    - Display option toggles
    - Animation customization

Deliverables:

- Residual graph visualization
- Min-cut highlighting
- Path history tracking
- Complete settings dialog

5.5 Phase 5: Polish and Testing (Week 9-10)

Objective: Comprehensive testing and user experience refinement

Tasks:

1. Comprehensive testing
    - Unit tests for all components
    - Integration tests with GrALoG
    - Test various network topologies
    - Edge cases (zero capacity, multiple paths, etc.)
2. Performance optimization
    - Efficient residual graph updates
    - Rendering optimization for large graphs
    - Memory management
3. User experience improvements
    - Intuitive controls
    - Clear status messages
    - Helpful error messages
    - Example networks (presets)
4. Documentation
    - Code documentation (Javadoc)
    - User manual
    - Tutorial examples
    - API documentation for extension

Deliverables:

- Complete test suite
- Polished user interface
- Comprehensive documentation
- Example networks

---

6. Testing Strategy

6.1 Unit Tests

6.1.1 FlowNetwork Tests

```java
@Test
public void testFlowConservation() {
    // Verify flow conservation at intermediate vertices
}

@Test
public void testCapacityConstraint() {
    // Verify flow never exceeds capacity
}

@Test
public void testSourceSinkDesignation() {
    // Test source/sink setting and validation
}
```

6.1.2 ResidualGraph Tests

```java
@Test
public void testResidualGraphConstruction() {
    // Verify residual capacities are correct
}

@Test
public void testBackwardEdgeCreation() {
    // Verify backward edges appear when flow > 0
}

@Test
public void testResidualGraphUpdate() {
    // Test update after flow augmentation
}
```

6.1.3 Algorithm Tests

```java
@Test
public void testSimpleMaxFlow() {
    // Test on simple 4-vertex network
}

@Test
public void testMultipleAugmentingPaths() {
    // Network requiring multiple iterations
}

@Test
public void testNoFlow() {
    // Disconnected source and sink
}

@Test
public void testBottleneckCalculation() {
    // Verify correct bottleneck identification
}
```

6.2 Integration Tests

6.2.1 GrALoG Plugin Tests

```java
@Test
public void testPluginRegistration() {
    // Verify plugin loads correctly
}

@Test
public void testStructureCreation() {
    // Test FlowNetwork creation through GrALoG
}

@Test
public void testAlgorithmExecution() {
    // Run algorithm through GrALoG interface
}
```

6.3 Test Networks

Network 1: Simple 4-Vertex

```
S --5--> A --3--> T
|              ^
+------6-------+
```

Expected max flow: 5

Network 2: Standard Example

```
   3     2
 A ---> B -+
 ^      | |
 |      v v
S|  1  C  T
 |      ^ |
 v      | v
 D ---> E -+
   4     5
```

Expected max flow: varies by capacities

Network 3: Zero Capacity

```
S --0--> T
```

Expected max flow: 0

Network 4: Complex Network

Use standard examples from literature (e.g., CLRS textbook examples)

6.4 Performance Tests

- Test with networks of 100, 500, 1000 vertices
- Measure rendering performance
- Measure algorithm execution time
- Memory profiling

---

7. Build Configuration

7.1 build.gradle

groovy

```groovy
plugins {
    id 'java'
}

group = 'gralog.fordfulkerson'
version = '1.0.0'

sourceCompatibility = 11
targetCompatibility = 11

repositories {
    mavenCentral()
    // Add GrALoG repository if available
}

dependencies {
    // GrALoG core (adjust path as needed)
    implementation project(':gralog-core')

    // Testing
    testImplementation 'junit:junit:4.13.2'
    testImplementation 'org.mockito:mockito-core:4.11.0'
}

test {
    useJUnit()
    testLogging {
        events "passed", "skipped", "failed"
        exceptionFormat "full"
    }
}

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

7.2 settings.gradle

groovy

```groovy
rootProject.name = 'gralog-fordfulkerson-plugin'
```

8. User Documentation

8.1 Quick Start Guide

Creating a Flow Network

1. Launch GrALoG and create a new graph
2. Select "Flow Network" from structure types
3. Add vertices by clicking on the canvas
4. Add edges by dragging from source to target vertex
5. Set edge capacity by double-clicking the edge
6. Right-click a vertex and select "Set as Source" or "Set as Sink"

Running Ford-Fulkerson

1. Select "Algorithms" → "Ford-Fulkerson Maximum Flow"
2. Choose path-finding method (BFS recommended)
3. Click "Start" to begin visualization
4. Use step controls to move through algorithm execution
5. View statistics panel for current flow information

Understanding the Visualization

- **Green edges**: Current augmenting path
- **Blue edges**: Full capacity (saturated)
- **Edge labels**: "flow/capacity" (e.g., "3/5")
- **Yellow vertices**: Vertices in current path
- **Red edges**: Min-cut edges (after completion)

8.2 Tutorial Examples

Example 1: Two-Path Network

```
Create a network:
S → A → T (capacity 5)
S → B → T (capacity 3)

Expected result:
- Path 1: S→A→T adds 5 flow
- Path 2: S→B→T adds 3 flow
- Maximum flow: 8
```

Example 2: Bottleneck Example

```
Create a network:
S → A (capacity 10)
A → T (capacity 3)

Expected result:
- Only 3 flow possible (bottleneck at A→T)
- Demonstrates bottleneck concept
```

Example 3: Backward Edge Example

```
Create a network demonstrating why backward edges matter:
S → A → B → T
S → C → B (competing for B→T capacity)

Show how algorithm may need to "undo" previous flow
```

---

9. Advanced Features (Future Enhancements)

9.1 Additional Algorithm Variants

- Capacity Scaling Ford-Fulkerson
- Dinic's Algorithm with blocking flows
- Push-Relabel (Goldberg-Tarjan) algorithm
- Comparison mode (run multiple algorithms side-by-side)

9.2 Enhanced Visualizations

- 3D network rendering for large graphs
- Animation of flow particles moving through network
- Heatmap showing edge utilization
- Interactive residual graph editing

9.3 Educational Features

- Built-in tutorial mode with guided examples
- Quiz mode: user must predict next augmenting path
- Algorithm performance comparison charts
- Export visualization as video/GIF

9.4 Analysis Tools

- Network statistics (diameter, connectivity, etc.)
- Sensitivity analysis (what if capacity changes?)
- Multiple source/sink support
- Min-cost max-flow extension

---

10. Deployment

10.1 Building the Plugin

```bash
# Clone GrALoG repository
git clone https://github.com/gralog/gralog.git
cd gralog

# Add plugin as submodule or subproject
# Create plugin directory
mkdir -p gralog-fordfulkerson-plugin
cd gralog-fordfulkerson-plugin

# Initialize project
gradle init

# Build plugin
./gradlew build

# Output: build/libs/gralog-fordfulkerson-plugin-1.0.0.jar
```

10.2 Installation

1. Locate GrALoG's plugins directory
2. Copy JAR file to plugins directory
3. Restart GrALoG
4. Verify plugin appears in plugins menu

10.3 Distribution

- Package as JAR file
- Include README.md and LICENSE
- Provide example networks (XML/JSON format)
- Host on GitHub or GrALoG plugins repository

---

11. Risk Assessment

11.1 Technical Risks

Risk	Probability	Impact	Mitigation
GrALoG API changes	Low	High	Use stable GrALoG version, document API dependencies
Performance issues with large graphs	Medium	Medium	Optimize rendering, implement level-of-detail
Algorithm bugs (correctness)	Low	High	Comprehensive testing, use proven algorithm implementations
Memory leaks with animations	Medium	Low	Proper resource cleanup, profiling

11.2 Project Risks

Risk	Probability	Impact	Mitigation
Scope creep	Medium	Medium	Strict feature prioritization, phased development
Integration complexity	Low	Medium	Early integration testing, consult GrALoG documentation
Timeline overrun	Medium	Low	Buffer time in schedule, incremental delivery

---

12. Success Criteria

12.1 Functional Requirements

✓ Plugin successfully loads in GrALoG
✓ Users can create and edit flow networks
✓ Algorithm correctly computes maximum flow
✓ Step-by-step visualization works smoothly
✓ All phases of algorithm are clearly visualized
✓ Min-cut is correctly identified and displayed

12.2 Performance Requirements

✓ Handles networks with up to 100 vertices without lag
✓ Step transitions occur within 500ms
✓ Memory usage remains reasonable (< 100MB for typical networks)

12.3 Usability Requirements

✓ Intuitive interface requiring no training
✓ Clear visual feedback for all actions
✓ Helpful error messages
✓ Comprehensive documentation
✓ At least 5 example networks included

---

13. Timeline Summary

Phase	Duration	Key Deliverables
Phase 1: Core Structure	2 weeks	Flow network structure, basic editing
Phase 2: Algorithm Core	2 weeks	Working Ford-Fulkerson implementation
Phase 3: Visualization	2 weeks	Step-by-step animation, controls
Phase 4: Advanced Features	2 weeks	Residual graph, min-cut, history
Phase 5: Polish & Testing	2 weeks	Complete testing, documentation
Total	**10 weeks**	**Production-ready plugin**

---

14. References

14.1 Algorithm Resources

- Cormen, Leiserson, Rivest, Stein. "Introduction to Algorithms" (3rd Edition), Chapter 26
- Ford, L.R., Fulkerson, D.R. (1956). "Maximal flow through a network"
- Edmonds, J., Karp, R.M. (1972). "Theoretical improvements in algorithmic efficiency"

14.2 GrALoG Resources

- GrALoG GitHub: https://github.com/gralog/gralog
- GrALoG Documentation: https://gralog.sourceforge.net/doc/website/index.html
- GrALoG Plugin Development Guide (if available)

14.3 Visualization References

- VisuAlgo: https://visualgo.net/en/maxflow
- iFlow interactive tool: https://arxiv.org/abs/2411.10484

14.4 Implementation Examples

- GeeksforGeeks Ford-Fulkerson implementations
- CP-Algorithms: https://cp-algorithms.com/graph/edmonds_karp.html
- Various GitHub repositories with Java implementations

Appendices

Appendix A: Sample Code Snippets

A.1 Basic Plugin Class

```java
package gralog.fordfulkerson;

import gralog.plugins.*;

public class FordFulkersonPlugin extends Plugin {

    @Override
    public void register() {
        // Register custom structures
        Structure.registerStructure(FlowNetwork.class);

        // Register algorithm
        Algorithm.registerAlgorithm(
            FordFulkersonAlgorithm.class,
            FlowNetwork.class,
            "Ford-Fulkerson Maximum Flow",
            "Computes maximum flow from source to sink"
        );

        // Register renderer
        Renderer.registerRenderer(
            FlowNetworkRenderer.class,
            FlowNetwork.class
        );
    }

    @Override
    public String getName() {
        return "Ford-Fulkerson Maximum Flow Plugin";
    }

    @Override
    public String getVersion() {
        return "1.0.0";
    }
}
```

A.2 Flow Edge Class

```java
package gralog.fordfulkerson.structure;

import gralog.structure.*;

public class FlowEdge extends Edge {
    private int capacity;
    private int flow;

    public FlowEdge() {
        this.capacity = 1;
        this.flow = 0;
    }

    public FlowEdge(int capacity) {
        this.capacity = capacity;
        this.flow = 0;
    }

    public int getCapacity() {
        return capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = Math.max(0, capacity);
    }

    public int getFlow() {
        return flow;
    }

    public void setFlow(int flow) {
        this.flow = Math.max(0, Math.min(flow, capacity));
    }

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

    @Override
    public String getLabel() {
        return flow + "/" + capacity;
    }
}
```

Appendix B: Example Networks (XML Format)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<flownetwork>
  <vertices>
    <vertex id="S" x="50" y="150" type="source"/>
    <vertex id="A" x="200" y="100"/>
    <vertex id="B" x="200" y="200"/>
    <vertex id="T" x="350" y="150" type="sink"/>
  </vertices>
  <edges>
    <edge from="S" to="A" capacity="5"/>
    <edge from="S" to="B" capacity="3"/>
    <edge from="A" to="T" capacity="5"/>
    <edge from="B" to="T" capacity="3"/>
  </edges>
</flownetwork>
```

Appendix C: Algorithm Pseudocode

```
FORD-FULKERSON(G, s, t):
    Initialize flow f to 0 on all edges

    while there exists an augmenting path p from s to t in residual graph Gf:
        bottleneck = minimum residual capacity along p

        for each edge (u,v) in p:
            if (u,v) is a forward edge:
                increase f(u,v) by bottleneck
            else: // (u,v) is a backward edge
                decrease f(v,u) by bottleneck

        update residual graph Gf

    return f (maximum flow)

FIND-AUGMENTING-PATH-BFS(Gf, s, t):
    Initialize queue Q with source s
    Mark s as visited
    parent[s] = null

    while Q is not empty:
        u = Q.dequeue()

        for each edge (u,v) with residual capacity > 0:
            if v is not visited:
                Mark v as visited
                parent[v] = u
                Q.enqueue(v)

                if v == t:
                    return RECONSTRUCT-PATH(parent, s, t)

    return null // no path found

RECONSTRUCT-PATH(parent, s, t):
    path = []
    current = t

    while current != null:
        path.prepend(current)
        current = parent[current]

    return path
```

Conclusion

This design document provides a comprehensive blueprint for developing a Ford-Fulkerson algorithm visualization plugin for GrALoG. The phased approach ensures systematic development with clear milestones, while the modular architecture allows for future extensions and enhancements.

The plugin will serve as an valuable educational tool, helping users understand one of the most important algorithms in graph theory through interactive, step-by-step visualization. By integrating seamlessly with GrALoG's existing framework, it extends the platform's capabilities while maintaining consistency with the user experience.

Next Steps:

1. Review and approve design document
2. Set up development environment
3. Begin Phase 1 implementation
4. Schedule regular progress reviews

Document Status: Complete
Approval Required: Yes
Estimated Development Time: 10 weeks
Target Completion: February 2026