Network Flow Algorithm Analysis Toolkit

MadFlow Group

User Guide

This document provides a comprehensive guide to using the scripts in this repository for generating graphs, computing maximum flow, benchmarking algorithms, and visualizing results.

Note: Python 3 Required

All scripts in this repository require Python 3. Depending on your system, the Python 3 executable may be named `python` or `python3`.

To check which command to use:

1. Run `python --version` - if this shows Python 3.x, you can use `python` instead of `python3` in all examples
2. If that shows Python 2.x or doesn't exist, run `which python3` to locate Python 3
3. If Python 3 is not installed, install it first, then use `python3` to run the scripts

All examples in this document use `python3`, but you may substitute `python` if that's your Python 3 command.

1. Repository Layout

The repository contains the following main components:

1.1. Source Files

`graph.py` — Core graph data structure and utilities. Defines the `Graph` class that represents a directed graph using an adjacency list and provides methods to load graphs from files, add edges, and perform BFS for finding augmenting paths.

`ford_fulkerson.py` — Standard Ford-Fulkerson algorithm implementation using BFS to find augmenting paths.

`scaling_ford_fulkerson.py` — Capacity scaling variant of Ford-Fulkerson that uses delta-scaling to improve performance on graphs with large capacities.

`preflow_push.py` — Preflow-Push (push-relabel) algorithm implementation using height labels and excess flow.

`mad-flow.py` — Unified command-line interface for running max flow algorithms. Supports all three algorithms and can output results in human-readable or JSON format.

`generate_graphs.py` — Graph generation script for creating test datasets. Generates four types of graphs: Bipartite, FixedDegree, Mesh, and Random. Compiles and runs Java graph generators from the `graphGenerationCode/` directory.

`benchmark.py` — Performance benchmarking tool that runs max flow algorithms multiple times on generated graphs, collects timing statistics, and uses multiprocessing for parallel execution.

`plot_results.py` — Visualization tool for benchmark results. Generates bar charts, comparison line charts, and ratio comparison plots. Requires matplotlib to be installed.

1.2. Directory Structure

- `GeneratedGraphs/` — Graphs used for Phase 1 analysis (organized by type: Bipartite/, FixedDegree/, Mesh/, Random/)
- `GeneratedGraphs2/` — Graphs used for Phase 2 analysis
- `GeneratedGraphs3/` — Graphs used for Phase 3 analysis
- `BenchmarkResultsData/` — Benchmark results from Phase 1 analysis (contains CSV and JSON files organized by algorithm and graph type)
- `BenchmarkResultsData2/` — Benchmark results from Phase 2 analysis (contains CSV and JSON files)
- `BenchmarkResultsData3/` — Benchmark results from Phase 3 analysis (contains CSV and JSON files)
- `BenchmarkResultsPlots/` — Default output directory for generated plots (organized by algorithm and graph type, plus Comparisons/)
- `graphGenerationCode/` — Java source code for graph generators (compiled on-the-fly by generate_graphs.py)

2. Running Flow Algorithms on a Single Graph

The `mad-flow.py` script provides a unified interface for running any of the three max flow algorithms on a single graph file.

```
# Run Ford-Fulkerson on a graph (default algorithm)
python3 mad-flow.py -g graphs/Mesh/g1.txt

# Use Scaling Ford-Fulkerson algorithm
python3 mad-flow.py -g graphs/Mesh/g1.txt -a scaling_ford_fulkerson

# Use Preflow-Push algorithm
python3 mad-flow.py -g graphs/Mesh/g1.txt -a preflow_push

# Get JSON output for scripting
python3 mad-flow.py -g graphs/Mesh/g1.txt -a scaling_ford_fulkerson --json

# Specify custom source and sink nodes
python3 mad-flow.py -g graph.txt -s start -t end -a preflow_push
```

3. Generating Graphs for Benchmarking

The `generate_graphs.py` script creates test graphs for benchmarking experiments. It supports four graph types: Bipartite, FixedDegree, Mesh, and Random.

```
# Generate all default graphs
python3 generate_graphs.py

# Generate first 5 graphs of each type
python3 generate_graphs.py -n 5

# Generate only random graphs with custom density
python3 generate_graphs.py --types random --random-density 10
```

```
# Generate graphs to a custom output directory
python3 generate_graphs.py -o GeneratedGraphs2 --types bipartite,mesh
```

Graphs are saved to the output directory (default: `GeneratedGraphs/`) organized by type. Each graph file is named descriptively based on its parameters.

4. Benchmarking Flow Algorithms

The `benchmark.py` script runs max flow algorithms multiple times on generated graphs and collects performance statistics.

```
# Benchmark all algorithms on all graph types (10 runs per graph)
# Clean the output directory before starting
python3 benchmark.py -i GeneratedGraphs -r 10 --clean

# Benchmark specific algorithm only
python3 benchmark.py -i GeneratedGraphs -a ford_fulkerson -r 10 --clean

# Benchmark multiple specific algorithms
python3 benchmark.py -i GeneratedGraphs -a ford_fulkerson,preflow_push -r 10 --
clean

# Benchmark specific graph types only
python3 benchmark.py -i GeneratedGraphs -t bipartite,mesh -r 10 --clean

# Limit to 4 parallel processes (default: number of CPU cores)
python3 benchmark.py -i GeneratedGraphs -r 10 -p 4 --clean
```

Note: The benchmark script automatically runs multiple graph analyses in parallel, using one process per CPU core by default. This significantly speeds up benchmarking on multicore systems. You can adjust the parallelism using the `-p` or `--processes` argument.

Results are saved in the output directory organized as:

```
BenchmarkResultsData/
  <algorithm>/
    <graph_type>/
      results.json    # Detailed results with all statistics
      results.csv     # Tabular format for spreadsheet analysis
```

5. Plotting Benchmark Results

The `plot_results.py` script generates visualizations from benchmark results. **Note: matplotlib must be installed** (`pip3 install matplotlib`).

```
# Generate all plots including comparisons
python3 plot_results.py --clean

# Generate plots with log scale versions
python3 plot_results.py --clean --log-scale
```

```
# Generate only comparison plots
python3 plot_results.py --clean --comparison-only

# Plot specific algorithms and graph types
python3 plot_results.py -a ford_fulkerson,preflow_push -t bipartite,mesh --clean
```

Plots are saved in the output directory organized as:

```
BenchmarkResultsPlots/
  <algorithm>/
    <graph_type>/
      mean_runtime.png    # Bar chart: mean runtime vs input size
      max_runtime.png     # Bar chart: max runtime vs input size
      mean_runtime.svg    # SVG versions of above
      max_runtime.svg
  Comparisons/
    <graph_type>/
      mean_runtime_comparison.png      # Line chart comparing all algorithms
      max_runtime_comparison.png
      ratio_comparison.png             # Ratio comparison plot
      [.svg versions of all above]
```

6. Algorithm Implementations

This section describes the key routines implemented in the core algorithm files.

6.1. graph.py

`Graph.__init__(file_path)` — Constructor that initializes a new graph and loads it from the specified file path.

`Graph.load_graph(file_path)` — Reads a graph file and constructs the adjacency list representation by parsing edges line by line.

`Graph.add_edge(u, v, w)` — Adds a directed edge from vertex u to vertex v with capacity w to the graph's adjacency list.

`Graph.BFS(s, t, parent, capacity_threshold=1)` — Performs breadth-first search to find an augmenting path from source s to sink t in the residual graph, considering only edges with capacity at or above the threshold.

`Graph.get_num_vertices()` — Returns the total number of vertices in the graph.

`Graph.get_num_edges()` — Returns the total number of edges in the graph.

6.2. ford_fulkerson.py

`ford_fulkerson(graph, source, sink)` — Implements the Ford-Fulkerson algorithm using BFS to repeatedly find augmenting paths and push flow until no more paths exist, returning the maximum flow value.

6.3. scaling_ford_fulkerson.py

`scaling_max_flow(graph, source, sink)` — Implements the capacity scaling variant of Ford-Fulkerson that uses delta-scaling (starting with the largest power of 2) to find augmenting paths only through edges with sufficient capacity, improving performance on graphs with large capacity values.

6.4. preflow_push.py

`preflow_push(graph, source, sink)` — Wrapper function that converts a Graph object into a capacity dictionary and calls the preflow-push algorithm.

`preflow_push_max_flow(capacity, source, sink)` — Implements the preflow-push (push-relabel) algorithm that maintains a preflow and height labels, using local push and relabel operations to move excess flow toward the sink.

`push(u, v)` — Helper function that pushes as much excess flow as possible from vertex u to vertex v along an admissible edge.

`relabel(u)` — Helper function that increases the height of vertex u to enable new push operations when no valid pushes are currently available.

`discharge(u)` — Helper function that repeatedly pushes flow from vertex u until all excess is removed or a relabel is needed.

7. Test Results and Output

The output from running the algorithms on all test input files can be found in the benchmark results directories:

- `BenchmarkResultsData/` — Contains CSV and JSON files with detailed results from Phase 1 analysis
- `BenchmarkResultsData2/` — Contains CSV and JSON files with detailed results from Phase 2 analysis
- `BenchmarkResultsData3/` — Contains CSV and JSON files with detailed results from Phase 3 analysis

Each results file includes timing statistics (min, max, mean, median, standard deviation), graph metadata (vertices, edges), and computed maximum flow values for each test graph.

8. Development and Testing Environment

This toolkit was developed and tested by the MadFlow Group using the following system configuration:

- **Python Version:** 3.14.1
- **Hardware:** Apple M3 Pro Mac
- **CPU Cores:** 11
- **RAM:** 18 GiB

The benchmark results and performance characteristics documented in this project were obtained using this configuration. Results may vary on different hardware and software environments.

Happy benchmarking! May your flows be maximum and your runtimes minimal.
