GI-Pipe User's Guide

Ales Shvaibovich

May 19, 2025

Abstract

This document serves as the user guide for the software package *GI-Pipe*, a modular pipeline developed for analyzing the performance of graph isomorphism algorithms, with a focus on the nauty toolset. The source code is available on GitHub. This guide, along with the accompanying technical documentation titled *"GI-Pipe Documentation"* [Shv25b], forms part of my Bachelor's Thesis titled *"Determining Graph Isomorphism with the Help of Advanced Tools"* [Shv25a]. The document explains how to use the pipeline's main components, including graph generation, isomorphism testing, visualization, and complexity estimation.

Contents

1 Introduction **3**

2 Prerequisites **3**

3 Pipeline **3**
- 3.1 Main `pipeline.sh` Script . . . 4
- 3.2 Generation . . . 5
  - 3.2.1 Dataset Specification . . . 5
  - 3.2.2 Pipeline Usage . . . 5
  - 3.2.3 Separate Usage . . . 5
  - 3.2.4 Example . . . 6
- 3.3 Processing . . . 6
  - 3.3.1 Pipeline Usage . . . 6
  - 3.3.2 Separate Usage . . . 6
  - 3.3.3 Examples . . . 6
- 3.4 Visualization . . . 7
  - 3.4.1 Pipeline Usage . . . 7
  - 3.4.2 Separate Usage . . . 7
  - 3.4.3 Examples . . . 7
- 3.5 Estimation . . . 8
  - 3.5.1 Pipeline Usage . . . 8
  - 3.5.2 Separate Usage . . . 8
  - 3.5.3 Examples . . . 8

4 Advanced Pipeline Usage **9**
- 4.1 Additional Scripts . . . 9
  - 4.1.1 Adjacency Matrix to graph6 Converter . . . 9
  - 4.1.2 Edge List to graph6 Converter . . . 10
  - 4.1.3 Isomorphic Graph Set Generator . . . 10
  - 4.1.4 SRG Graph Generator from Parameters . . . 10
  - 4.1.5 Planar Graph Generation . . . 11
- 4.2 Adding a New Graph Type . . . 11
  - 4.2.1 Option 1: Custom Graph Generator . . . 11
  - 4.2.2 Option 2: Prepared Dataset . . . 11

1 Introduction

This guide introduces *GI-Pipe* (short for *Graph Isomorphism Pipeline*), a software framework for evaluating graph isomorphism algorithms in a controlled and extensible environment. The primary objective of *GI-Pipe* is to provide empirical insights into the time complexity of the nauty algorithm when applied to different graph types and sizes.

GI-Pipe consists of several interconnected modules that handle tasks such as dataset generation, graph isomorphism checking, performance logging, result visualization, and complexity estimation. Each module can be used independently or as part of a larger automated pipeline, depending on the user's needs.

This guide provides comprehensive usage instructions for the pipeline and its components, including script descriptions, command-line interfaces, and example workflows. It also offers guidance for extending the pipeline with additional graph types or datasets.

2 Prerequisites

GI-Pipe is guaranteed to run on macOS. It may also be compatible with Linux distributions, but no official support is provided for Windows at this time. The following steps outline the setup instructions for macOS:

1. Clone the project repository from GitHub.
2. Download and install SageMath for macOS. For Linux users, installation instructions and binaries are available at SageMath's official website.
3. Install CMake version 3.30 or newer.
4. Download and compile the `nauty` library from the Nauty and Traces homepage. After successful compilation, move the resulting `nauty` directory (e.g., `nauty_2_8_9`) into the root directory of the cloned repository.
5. Install Python 3.10 or newer.
6. Install required Python modules by running:

   ```
   pip install pandas matplotlib scipy scikit-learn
   ```

To test the installation, run the following command:

```
./pipeline.sh --type random
```

If everything is installed correctly, this script will:

- Generate a small dataset of random graphs with default parameters.
- Compile the C source file into an executable named `process.exe`.
- Process the generated dataset and save results to the `processed` directory.
- Visualize the results and save the figures to the `pictures` directory.
- Estimate time complexity, and save the computed metrics and plots to the `pictures` directory.

3 Pipeline

It is strongly recommended to run GI-Pipe exclusively using the `pipeline.sh` script rather than executing individual stages separately. The main pipeline script provides options to isolate specific stages (e.g., `--drop_gen` to skip the generation stage and use a preexisting dataset). This section documents all capabilities of the script. First, the help message of `pipeline.sh` is provided, followed by detailed descriptions of individual stages.

3.1 Main `pipeline.sh` Script

To distinguish between multiple runs, the pipeline uses timestamps. All directories and files created during a run are labeled with a unique timestamp, which is printed to the console at the beginning of execution.

Help text of the `pipeline.sh` script:

```
Usage: ./pipeline.sh [OPTIONS]

This script runs a complete pipeline for generating, processing, and visualizing graph datasets.
It supports multiple graph types including trees, random graphs, regular graphs, and more.

GENERAL OPTIONS:
  -h, --help, ?                 Show this help message and exit.

GRAPH TYPE (required if generation stage is not dropped):
  --type <graph_type>           The type of graph to generate or process.

  Supported graph types:
    path
    regular_bipartite
    complete_bipartite
    bipartite
    tree
    random
    regular
    cactus
    cycle
    complete
    srg
    planar

FLOW OPTIONS:
  --drop_gen <dataset_path>     Skip generation stage and use an existing dataset at the given path.
  --drop_proc <processed_file>  Skip both generation and processing stages.
                                Provide path to an existing processed .csv file.
  --drop_vis <processed_file>   Skip generation, processing and visualisation stages.
                                Provide path to an existing processed .csv file.
  --only_gen                    Run only generation stage.
  --only_proc <dataset_path>    Run only processing stage with dataset at the given path.
  --only_vis <processed_file>   Run only visualisation stage with provided processed .csv file.
  --only_est <processed_file>   Run only estimation stage with provided processed .csv file.

GENERATION OPTIONS (used if generation is not dropped):
  --start <int>                 Starting graph size (default: 10).
  --end <int>                   Ending graph size (default: 500).
  --step <int>                  Step size between graph sizes (default: 10).
  --set_num <int>               Number of graphs to generate per size (default: 3).
  --oi                          Only generate isomorphic graphs.

TYPE-SPECIFIC GENERATION OPTIONS:
  For 'random' and 'bipartite':
    --density <float>           Density of random graphs (default: 0.5).

  For 'regular' and 'regular_bipartite':
    --degree <int>              Degree of each vertex (default: 3).

PROCESSING OPTIONS:
```

```
  --opt_tree                    Run processing stage with optimization for trees.

NOTES:
  - For 'srg' and 'planar' types, generation is skipped
            automatically and pre-prepared datasets are used.
  - For 'regular', 'regular_bipartite', 'cycle', and
            'complete' types, only isomorphic graphs are generated by default.

EXAMPLES:
  Generate and process tree graphs with default settings:
    ./pipeline.sh --type tree

  Generate random graphs of size 20 to 100 with density 0.7:
    ./pipeline.sh --start 20 --end 100 --type random --density 0.7

  Use existing dataset and skip generation:
    ./pipeline.sh --drop_gen generated_dataset/random/123456/

  Use existing processed file and skip gen/proc:
    ./pipeline.sh --drop_proc processed/regular/123456.csv
```

3.2 Generation

The generation stage uses SageMath to create a dataset of graphs in `graph6` format. The graph type (e.g., `--type tree`) and size range (`--start`, `--end`, `--step`) can be configured, along with parameters like density or degree, depending on the type.

For each graph size, two sets of graphs are created:

- An **isomorphic** set
- A **non-isomorphic** set

These are saved to subdirectories `isomorphic/` and `non_isomorphic/` respectively. To generate only isomorphic graphs, use the `--oi` flag.

3.2.1 Dataset Specification

A dataset directory may contain:

- Two subdirectories: `isomorphic/` and `non_isomorphic/`
- One subdirectory: `isomorphic/`
- No subdirectories (implies only isomorphic data)

Each dataset file (with `.g6` extension) is named according to the graph size it contains (e.g., `10.g6` holds graphs of size 10).

3.2.2 Pipeline Usage

Refer to `GENERATION OPTIONS`, `GRAPH TYPE`, and `TYPE-SPECIFIC OPTIONS` in section 3.1.

3.2.3 Separate Usage

Although not recommended, the generation stage can be run independently.

Help text for `generation.py`:

```
usage: sage -python generation.py [-h] --type TYPE [--density DENSITY] [--degree DEGREE]
    --start START --end END --step STEP --set_size SET_SIZE --output_dir OUTPUT_DIR [--oi]

Generate graphs and save them in a specified directory.
```

```
options:
  -h, --help            show this help message and exit
  --type TYPE           Type of graphs to generate: tree, random, regular, etc.
  --density DENSITY     Density for some graph types. Default is 0.5.
  --degree DEGREE       Degree for some graph types. Default is 3.
  --start START         Starting number of nodes in the graphs.
  --end END             Ending number of nodes in the graphs.
  --step STEP           Step size for the number of nodes.
  --set_size SET_SIZE   Number of graphs to generate for each size.
  --output_dir OUTPUT_DIR
                        Output directory for saving the graphs.
  --oi                  Only generate isomorphic graphs if set.
```

3.2.4 Example

```
./pipeline.sh --type tree --start 10 --step 5 --end 20 --set_size 3 --oi
```

Or separately:

```
sage -python generation.py --type tree --start 10 --step 5 --end 20 --set_size 3 --oi
```

This generates an `isomorphic/` directory containing `10.g6`, `15.g6`, and `20.g6` files, each with 3 isomorphic tree graphs of the specified sizes.

3.3 Processing

This stage runs the graph isomorphism detection algorithm on the dataset and records execution times into a CSV file.

3.3.1 Pipeline Usage

See `PROCESSING OPTIONS` in section 3.1.

3.3.2 Separate Usage

Executable syntax:

```
./process.exe <dataset_path> <result_file> [--opt_tree]
```

Arguments:

- `<dataset_path>`: Path to dataset directory (trailing slash required)
- `<result_file>`: Output CSV filename
- `--opt_tree`: Enable tree-specific optimizations

3.3.3 Examples

Example 1. Using the processing stage separately:

```
./process.exe generated_dataset/tree/12345/ result.csv --opt_tree
```

In this example, it is assumed that the directory `generated_dataset/tree/12345/` contains the graph dataset. The dataset will be processed, and the execution time will be stored in `result.csv`. Additionally, tree optimization will be applied during processing.

Example 2. Using the entire pipeline:

```
./pipeline.sh --type tree --opt_tree
```

In this example, the processing stage uses the dataset generated in the previous (generation) stage, and the results are stored in an automatically named file. Tree optimization is also applied.

Example 3. Structure of the processed CSV file:

node_count	average_time	is_isomorphic
10	0.000001	true
20	0.000002	true
30	0.000006	true

- `node_count` — number of vertices in graphs within the set.
- `average_time` — average processing time per graph pair.
- `is_isomorphic` — boolean flag indicating whether graphs in the set were isomorphic.

3.4 Visualization

The visualization stage provides graphical insight into the execution times of the isomorphism detection algorithm. It transforms the CSV output from the processing stage into plots that illustrate the relationship between graph size and runtime.

Three types of plots are generated:

- **data.png** — standard linear scale for both axes.
- **data_scaled_y.png** — logarithmic scale on the y-axis.
- **data_scaled_xy.png** — logarithmic scale on both axes.

3.4.1 Pipeline Usage

When using the full pipeline script, the visualization stage runs automatically (unless explicitly dropped with `--drop_vis`). File paths and names are managed automatically.

3.4.2 Separate Usage

The visualization script can be executed independently of the pipeline:

```
usage: draw.py [-h] --data_file DATA_FILE --output_dir OUTPUT_DIR

Visualize isomorphism algorithm performance.

options:
  -h, --help                Show this help message and exit.
  --data_file DATA_FILE     File with data to visualize.
  --output_dir OUTPUT_DIR   Relative path of output directory.
```

3.4.3 Examples

Example 1. Standalone usage that requires explicit specification of the data file and output directory:

```
python draw.py --data_file processed/12345.csv --output_dir pictures/12345/
```

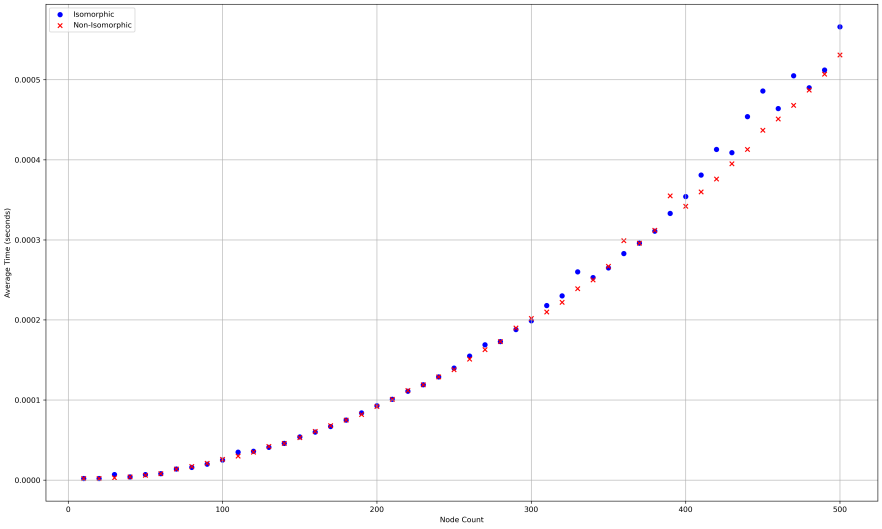
In this example, the algorithm's execution time for the number of nodes from the file `12345.csv` will be visualized and saved in the directory `pictures/12345/`. The generated files are `data.png`, `data_scaled_y.png`, and `data_scaled_xy.png`.

Example 2. Using the complete pipeline:

```
./pipeline.sh --type random
```

In this example, there is no need to specify the data file name or output directory, as they are automatically generated by the pipeline.

Example 3. Visualization appearance:



3.5 Estimation

The estimation stage performs curve fitting to determine which mathematical function best approximates the growth of execution time with respect to graph size. This helps evaluate the practical time complexity of the isomorphism algorithm on different graph types.

The script produces two outputs:

- **metrics.csv** — parameters of the best fitting functions.
- **estimation.png** — visualization of the fitted curve(s) against empirical data.

3.5.1 Pipeline Usage

When run as part of the full pipeline, the estimation step is performed automatically and output files are saved in the corresponding results directory.

3.5.2 Separate Usage

You can run the estimation script independently with the following syntax:

```
usage: estimate.py [-h] --data_file DATA_FILE --output_dir OUTPUT_DIR

Estimate best fitting function and visualize results.

options:
  -h, --help                Show this help message and exit.
  --data_file DATA_FILE     Relative path to CSV file with data for estimation.
  --output_dir OUTPUT_DIR   Relative path of output directory for picture and CSV metric result.
```

3.5.3 Examples

Example 1. Separate usage of the estimation script:

```
python estimate.py --data_file processed/123.csv --output_dir pictures/
```

In this example, metrics are estimated based on the processed file `processed/123.csv`, saved to `pictures/metrics.csv`, and visualized in `pictures/estimation.png`.

Example 2. Usage as part of the pipeline:

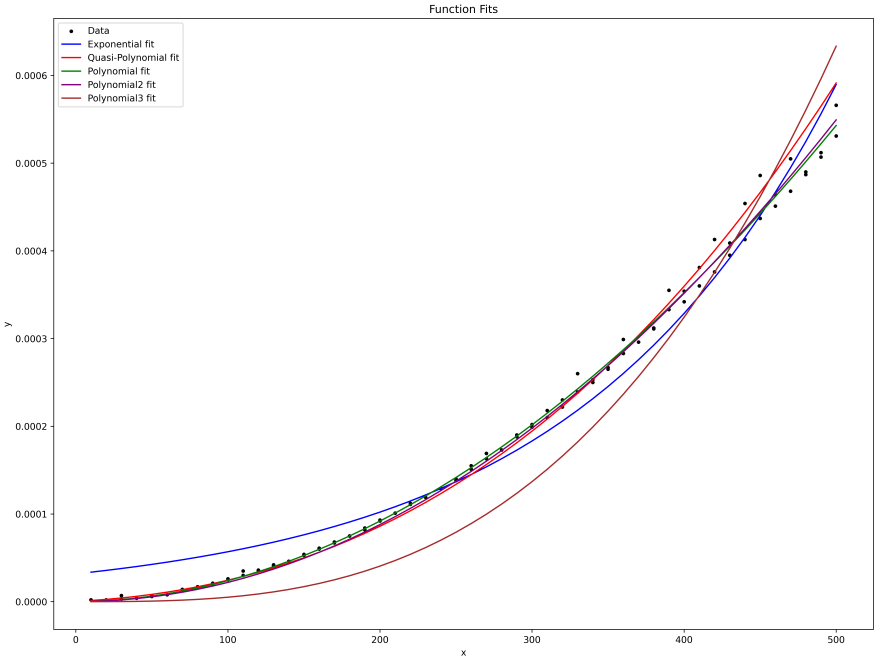
```
./pipeline.sh --type path
```

In this example, there is no need to provide the data file name or output directory, as they are automatically named by the pipeline.

Example 3. Metrics file structure preview:

functionName	RSS	r2	parameter1	parameter2	parameter3
exponential	6.6125363e-08	0.97570707	3.171485e-05	0.99295172	-0.8263916
quasi_polynomial	2.7496541e-08	0.9898984	-14.5252974	0.27017917	1.98925802
polynomial	7.8788181e-09	0.9971055	3.17845e-09	1.93870048	
polynomial2	8.852834e-09	0.99674767	2.19718e-09		
polynomial3	2.10018655e-07	0.92284401	5.0665888e-12		

Example 4. Visualization preview:



4 Advanced Pipeline Usage

This section describes how to use supporting scripts that enable generation of datasets for additional graph types. These tools are useful for extending the pipeline and customizing it to your needs.

4.1 Additional Scripts

This subsection introduces standalone utilities used for transforming or generating graph datasets in various formats.

4.1.1 Adjacency Matrix to graph6 Converter

Usage:

```
usage: sage -python adj_to_graph6.py <input_directory> <output_directory>

Converts adjacency matrix graph representation to graph6 format.

options:
    <input_directory>       Directory with txt files where adjacency matrix is stored.
    <output_directory>      Directory where output g6 files will be stores.
```

Converts adjacency matrix representations (stored as .txt files) into graph6 format. All .txt files in the input directory are processed and converted; resulting .g6 files are saved in the output directory with matching base names.

4.1.2 Edge List to graph6 Converter

Usage:

```
usage: sage -python edges_to_graph6.py <input_file> <output_dir>

Converts edges of list generated by Boltzmann generator graph representation to graph6 format.

options:
    <input_file>        File with txt files where adjacency matrix is stored.
    <output_dir>        Directory where output g6 file will be stores.
```

Converts a single edge list (in .txt format, as generated by a Boltzmann sampler [Fus09]) into a graph6 file. The script skips the first and last lines of the file, computes the number of nodes, and names the resulting graph accordingly.

4.1.3 Isomorphic Graph Set Generator

Usage:

```
usage: sage -python graph6_iso_dup.py <input_directory> <output_directory>

Generates set of isomorphic graphs for each graph in <input_directory>.

options:
    <input_directory>       Directory with g6 files where one graph is stored.
    <output_directory>      Directory where output g6 files will be stores.
```

Generates multiple isomorphic variants of each input graph. Input files must each contain a single graph in graph6 format. Output files are saved with the same names to the specified output directory.

4.1.4 SRG Graph Generator from Parameters

Usage:

```
usage: sage -python generate_srg_from_csv.py <csv_file> <output_directory>

Reads all parameters from <csv_file> and generates SRG graphs for them.

options:
    <csv_file>              Files with parameters for SRG graphs.
    <output_directory>      Directory where output g6 files will be stores.
```

Reads a CSV file containing parameters of strongly regular graphs (SRGs) and generates the corresponding graphs in graph6 format. Output filenames are based on the number of nodes.

Input CSV format:

v	k	lambda	mu
5	2	0	1
9	4	1	2
10	3	0	1
13	6	2	3
15	6	1	3

4.1.5 Planar Graph Generation

Usage:

`./planar_generator.sh`

This script uses a Boltzmann generator [Fus09] to produce planar graphs and internally invokes the edges_to_graph6.py script (see section 4.1.2) to convert the generated edge lists to graph6 format. The output files are stored in the directory `graph6_single/planar/`.

4.2 Adding a New Graph Type

To extend the pipeline with a new graph type, follow one of the procedures below depending on whether you are generating graphs programmatically or using a prepared dataset.

4.2.1 Option 1: Custom Graph Generator

If you have your own algorithm to generate graphs:

1. If the graph type ensures only one isomorphism class per node count (e.g., complete graphs), set `ONLY_ISOMORPHIC="true"` in `pipeline.sh` for your type.
2. Implement the generator function as `generate_yourType` in `generation.py`.
3. Add a corresponding case to the `generate_graph` function in `generation.py`.

Refer to the documentation for details on how to enforce pipeline parameters.

4.2.2 Option 2: Prepared Dataset

If you already have a dataset:

1. Ensure your dataset matches the format described in section 3.2.1.
2. Place it in the directory `prepared_dataset/yourType/`.
3. In `pipeline.sh`, set `RUN_GEN="false"` and `DATASET_DIR="prepared_dataset/yourType/"` to use your dataset instead of generating new graphs.

Refer to the documentation [Shv25b] for details on how to enforce pipeline parameters.

References

[Fus09] Éric Fusy. Uniform random sampling of planar graphs in linear time. *Random Structures and Algorithms*, 35(4):464–522, 6 2009.

[Shv25a] Ales Shvaibovich. Determining graph isomorphism with the help of advanced tools, 2025.

[Shv25b] Ales Shvaibovich. Gi-pipe documentation, 2025.