GI-Pipe Documentation

Ales Shvaibovich

May 19, 2025

Abstract

This document provides detailed documentation for the GI-Pipe software, a multi-stage pipeline developed for evaluating the performance of graph isomorphism detection algorithms, particularly the nauty library. GI-Pipe automates dataset generation, algorithm execution, data visualization, and runtime estimation. The source code is available at GitHub, and this document complements the *GI-Pipe User's Guide* [Shv25b]. Together, they form the practical component of the author's Bachelor's Thesis on the topic *"Determining graph isomorphism with the help of advanced tools"* [Shv25a].

Contents

1 Introduction

Graph isomorphism is a classical problem in computer science and mathematics, with applications in chemistry, pattern recognition, and network theory. Determining whether two graphs are structurally identical remains a computationally challenging task, and several algorithms have been developed to tackle it — among them, the nauty algorithm stands out for its efficiency and practical utility.

GI-Pipe is a modular pipeline designed to analyze the behavior and performance of the nauty algorithm under varying input conditions. It consists of four stages: graph dataset generation, algorithm execution, data visualization, and computational complexity estimation. Throughout the pipeline, several tools — including SageMath, nauty, and Python libraries — are integrated to automate and streamline the evaluation process.

This document provides an in-depth overview of the architecture and implementation of GI-Pipe, focusing particularly on the more complex components of the system. Usage and setup instructions are documented separately in the accompanying manual titled *GI-Pipe User's Guide* [Shv25b]. Readers are encouraged to consult that guide for operational details and prerequisites.

2 Main Pipeline Script

The main pipeline script is implemented in Bash. It orchestrates the execution of individual scripts sequentially, passing the output of each stage as input to the subsequent stage.

2.1 Command-Line Options

The script accepts the following command-line options:

1. `--drop_gen <dataset_path>`: Skips the generation stage and uses an existing dataset located at the specified path.
2. `--drop_proc <processed_file>`: Skips both the generation and processing stages. Accepts the path to an existing `.csv` file containing processed data.
3. `--drop_vis <processed_file>`: Skips the generation, processing, and visualization stages. Accepts a path to a processed `.csv` file.
4. `--only_gen`: Runs only the graph generation stage.
5. `--only_proc <dataset_path>`: Runs only the processing stage, using the dataset located at the specified path.
6. `--only_vis <processed_file>`: Runs only the visualization stage, using the specified processed `.csv` file.
7. `--only_est <processed_file>`: Runs only the estimation stage, using the specified processed `.csv` file.
8. `-h, --help, ?`: Displays the help message and exits.
9. `--start <int>`: Specifies the starting graph size (default: 10).
10. `--end <int>`: Specifies the ending graph size (default: 500).
11. `--step <int>`: Sets the step size between graph sizes (default: 10).
12. `--set_num <int>`: Specifies the number of graphs to generate per size (default: 3).
13. `--oi`: Generates only isomorphic graph pairs.
14. `--type <graph_type>`: Specifies the type of graph to generate or process.
15. `--density <float>`: Sets the density for random graphs (default: 0.5).
16. `--degree <int>`: Sets the degree of each vertex (default: 3).

17. `--opt_tree`: Applies optimizations specific to tree processing.

The script uses boolean flags (`RUN_GEN`, `RUN_PROC`, `RUN_VIS`, `RUN_EST`) to control the execution of each pipeline stage. These flags are enabled by default and can be modified using options 1–7.

Options 9–16 are forwarded to the generation stage and fall back to default values if not specified. Option `--type` is also used to name output directories.

Option 17 is passed to the processing stage.

2.2 Forced Parameter Adjustments

Certain graph types require special handling, as shown in the following code:

```
case "$GRAPH_TYPE" in
    srg|planar)
        RUN_GEN="false"
        DATASET_DIR="prepared_dataset/${GRAPH_TYPE}/"
        ;;
    cycle|complete)
        ONLY_ISOMORPHIC="true"
        ;;
esac
```

Strongly regular (SRG) and planar graphs are difficult to generate dynamically. Therefore, pre-prepared datasets are used for these types (the dataset preparation methodology is described in the Bachelor's thesis [Shv25a]). Consequently, `RUN_GEN` is disabled, and `DATASET_DIR` is set to the appropriate dataset path.

For cycles and complete graphs, generating non-isomorphic graphs of the same size is impossible; hence, the `ONLY_ISOMORPHIC` flag is always set to true for these types.

2.3 Directory Naming Scheme

Directory names are generated dynamically to reduce the need for manual specification:

```
if [ "$RUN_GEN" = "true" ]; then
    DATASET_DIR="generated_dataset/${GRAPH_TYPE}/${TIMESTAMP}/"
fi
if [ "$RUN_PROC" = "true" ]; then
    PROCESSED_FILENAME="processed/${GRAPH_TYPE}/${TIMESTAMP}.csv"
fi
PICTURE_DIR="pictures/${GRAPH_TYPE}/${TIMESTAMP}/"
```

To distinguish between different executions of the pipeline, a timestamp is used. All generated files and directories are labeled with this timestamp, which is printed at the beginning of the script execution.

Note that when using `--drop_gen` or `--drop_proc`, the `--type` argument is not required. In these cases, `GRAPH_TYPE` defaults to `"default"`, which is acceptable as long as the generation stage is not executed.

2.4 Compiling `process.exe`

If the script is executed for the first time, it compiles `process.exe` using CMake:

```
if [ ! -f ./process.exe ]; then
    mkdir -p build
    cd build
    cmake ..
    make
    cp ./process.exe ../process.exe
    cd ..
fi
```

The compiled executable is not distributed due to platform-specific dependencies on the nauty library. Nauty must be compiled for the specific operating system and processor architecture in use. The compilation process is described in the prerequisites section of the *GI-Pipe User's Guide* [Shv25b], and the executable is linked via `CMakeLists.txt`.

3 Generation script

The generation script is written in Python using SageMath. It generates datasets of graphs with customizable parameters. Input parameters correspond to options 4–11 described in section 2.1. The script relies on SageMath methods, and only the core logic will be described in detail: isomorphic generation, non-isomorphic generation, and generation of non-trivial graph types.

3.1 Graph set generation

A shared function is used for both isomorphic and non-isomorphic graph set generation:

```
def generate_graphs(is_isomorphic, start, end, step, set_size, output_dir,
                                                              graph_type, density, degree):
    output_dir = os.path.join(output_dir, "isomorphic" if is_isomorphic else "non_isomorphic")
    os.makedirs(output_dir, exist_ok=True)

    for n in range(start, end + 1, step):
        # If exception is thrown during generation, skip current n generation
        try:
            # Generate graph set
            if is_isomorphic:
                graph_set = generate_isomorphic_set(n, set_size, graph_type, density, degree)
            else:
                graph_set = generate_non_isomorphic_set(n, set_size, graph_type, density, degree)

            # Save graph set to file in subdirectory
            file_path = os.path.join(output_dir, f"{n}.g6")
            with open(file_path, "w") as file:
                for g in graph_set:
                    file.write(g.graph6_string() + "\n")

        except Exception as e:
            print(f"Exception during graph generation: {e}")

    print(f"Dataset generated and saved in '{output_dir}'.")
```

3.1.1 Isomorphic generation

Functions for generating isomorphic graph sets:

```
def generate_isomorphic_set(n, set_size, graph_type, density, degree):
    # Generate first graph, then just shuffle its labeling
    original_graph = generate_graph(graph_type, n, density, degree)
    shuffled_graphs = [shuffle_labels(original_graph) for _ in range(set_size - 1)]
    return [original_graph] + shuffled_graphs

def shuffle_labels(graph):
    permutation = list(graph.vertices())
    random.shuffle(permutation)
    return graph.relabel(permutation, inplace=False)
```

For each node count n, the following steps are performed:

1. Generate a graph of the specified type.
2. Shuffle the vertex labels to create multiple isomorphic versions of the graph.
3. Save all generated graphs in graph6 format to a single file named after the current node count n.

3.1.2 Non-isomorphic generation

Functions for generating non-isomorphic graph sets:

```
def generate_non_isomorphic_set(n, set_size, graph_type, density, degree):
    non_isomorphic_graphs = []
    seen_canonical_labels = set()
    attempt = 0

    # Try to generate non-isomorphic graphs until set is filled or MAX_NON_ISO_ATTEMPTS is reached
    while len(non_isomorphic_graphs) < set_size and attempt < MAX_NON_ISO_ATTEMPTS:
        # Generate graph and take its canonical labeling
        graph = generate_graph(graph_type, n, density, degree)
        canonical_label = graph.canonical_label().copy(immutable=True)

        # If graph has unique canonical labeling, then it is not isomorphic to any preceding graph
        if canonical_label not in seen_canonical_labels:
            non_isomorphic_graphs.append(graph)
            seen_canonical_labels.add(canonical_label)

        # Increment attempt counter
        attempt += 1

    # If no non-isomorphic graph was generated, throw an error
    if len(non_isomorphic_graphs) >= 2:
        return non_isomorphic_graphs
    else:
        raise RuntimeError(f"Cannot generate non-isomorphic graphs.")
```

For each node count n, the generation proceeds as follows:

1. Initialize `seen_canonical_labels` as an empty set.
2. Generate a graph using the specified parameters.
3. Compute the canonical labeling of the graph.
4. If the canonical label is not in `seen_canonical_labels`, add it and keep the corresponding graph.
5. Repeat steps 2–4 until the required number of graphs is reached or the maximum number of attempts is exceeded.
6. Save all generated graphs in graph6 format to a file named after the current node count n. Raise an error if no graphs are generated.

3.2 Graph type generation

Most graph types are generated using built-in SageMath functions (e.g., `graphs.PathGraph(n)`). For unsupported types, custom generation algorithms are implemented.

3.2.1 Cactus

Custom cactus graph generator:

```
def generate_cactus(n):
    if n < 1:
        raise ValueError("Number of vertices must be positive")

    G = Graph()
    G.add_vertex(0)
    node_counter = 1 # count of existing nodes = index of next node

    while node_counter < n:
        # Choose a node already in the graph to attach something to
        base = random.choice(G.vertices())

        remaining = n - node_counter

        # Randomly decide to add:
        # - a single edge
        # - a cycle
        action = random.choice(['edge', 'cycle']) if remaining >= 2 else 'edge'

        if action == 'edge':
            G.add_edge(base, node_counter)
            node_counter += 1
        else:
            cycle_length = random.randint(2, remaining)
            cycle_nodes = [node_counter + i for i in range(cycle_length)]
            G.add_edges([(base, cycle_nodes[0])] +
                        [(cycle_nodes[i], cycle_nodes[i + 1]) for i in range(cycle_length - 1)] +
                        [(cycle_nodes[-1], base)])
            node_counter += cycle_length

    return G
```

Cactus graphs are constructed incrementally by attaching new elements to an existing graph. The steps are:

1. Start with a graph containing a single vertex.
2. Initialize `node_counter` to 1.
3. Select a base vertex to attach a new element.
4. Calculate the number of remaining vertices to add.
5. Randomly choose to add an `edge` or a `cycle`.
6. If `edge`, connect a new vertex to the base and increment `node_counter`.
7. If `cycle`, choose a random number of vertices, form a cycle with the base, and increment `node_counter`.
8. Repeat until the desired number of vertices n is reached.

3.2.2 Bipartite

Custom bipartite graph generator:

```
def generate_bipartite(n, density):
    if not (0 <= density <= 1):
        raise ValueError("Density must be between 0 and 1")
    if n < 2:
        raise ValueError("At least 2 vertices are needed")

    vertices = list(range(n))
    random.shuffle(vertices)

    # Random split
    split_point = random.randint(1, n-1)
    group1 = vertices[:split_point]
    group2 = vertices[split_point:]

    # Add all vertexes to graph (without edges)
    G = Graph()
    G.add_vertices(vertices)

    # Connect each vertex from u to each vertex from v with probability = density
    for u in group1:
        for v in group2:
            if random.random() < density:
                G.add_edge(u, v)

    return G
```

The script performs the following steps:

1. Create a list of n vertices.
2. Randomly split the list into two disjoint sets U and V.
3. For each pair (u, v) with $u \in U$ and $v \in V$, add an edge with probability equal to the specified `density`.

3.2.3 Regular Bipartite

Custom generator for d-regular bipartite graphs:

```
def generate_regular_bipartite(n, d):
    if n < 2:
        raise ValueError("At least 2 vertices are required")
    if d < 1:
        raise ValueError("Degree must be at least 1")
    if n % 2 != 0:
        raise ValueError("n must be even for perfect bipartite regularity")
    if d > n // 2:
        raise ValueError("Degree too high for bipartite regular graph")

    # Parts of Regular Bipartite graph are always of the same size
    group1 = list(range(n // 2))
    group2 = list(range(n // 2, n))

    # Add all vertexes to graph (without edges)
    G = Graph()
    G.add_vertices(group1 + group2)

    edges = set()
```

```
    # Try multiple times to build a valid graph (to handle randomness)
    for attempt in range(REGULAR_BIPARTITE_MAX_ATTEMPTS):
        G.clear()   # Clear all edges for a new attempt
        degree_group1 = {u: 0 for u in group1}
        degree_group2 = {v: 0 for v in group2}
        edges.clear()

        # Generate all possible edges (u,v) with u in group1 and v in group2
        possible_pairs = [(u, v) for u in group1 for v in group2]
        random.shuffle(possible_pairs)  # Shuffle to randomize edge selection

        # Try to add edges while respecting the degree constraint
        for u, v in possible_pairs:
            if degree_group1[u] < d and degree_group2[v] < d and (u, v) not in edges:
                G.add_edge(u, v)
                edges.add((u, v))
                degree_group1[u] += 1
                degree_group2[v] += 1

        # Check if the graph is d-regular (every node in both groups has degree d)
        if all(degree == d for degree in degree_group1.values()) and all(
                degree == d for degree in degree_group2.values()):
            return G

    raise RuntimeError("Failed to generate regular bipartite graph after many attempts.")
```

The algorithm works as follows:

1. Divide n vertices equally into two groups.
2. Initialize an empty graph with all vertices from both groups.
3. Reset all edges and initialize degree counters and the edge set.
4. Generate all possible pairs (u, v) with u in group 1 and v in group 2, and shuffle the list.
5. Iterate through the shuffled pairs:
   (a) Check if both vertices have degrees less than d and that the edge is not already present.
   (b) If valid, add the edge, increment degrees, and record the edge.
6. If all vertices reach degree d, return the graph.
7. Otherwise, repeat from step 3 up to the maximum number of attempts.

4 Processing Application

The processing stage is implemented in C using the nauty library. The program is split across multiple source files for modularity and clarity:

1. `main.c` — handles input arguments, initiates processing of graph sets, and stores results.
2. `file_processor.c/h` — provides functions for file manipulation, such as reading graphs from files and writing results to CSV.
3. `graph_processor.c/h` — implements general-purpose graph processing functionality.
4. `nauty_isomorphism.c/h` — uses nauty's isomorphism detection algorithm for graph processing.
5. `tree_optimization.c/h` — provides a faster tree-specific isomorphism detection algorithm.
6. `my_graph.c/h` — defines a custom graph structure used in the tree optimization routines.

4.1 Dataset Detection

First, the program checks for subdirectories. If the specified dataset directory contains subdirectories named `isomorphic` or `non_isomorphic`, the datasets are loaded from there accordingly. If no such subdirectories exist, it is assumed that only the isomorphic dataset is present and located directly in the specified directory.

All files from the determined directory (or subdirectory) are read for processing. Filenames must follow the format `n.g6`, where `n` denotes the number of vertices in the graphs contained in the file. Each file represents one graph set to be processed.

4.2 General Graph Set Processing

```
double process_graph_set(graph **graphs, const int graph_count, const int n,
                                                    bool is_isomorphic, bool opt_tree_flag) {
    double total_time = 0.0;  // Total time for isomorphism checks
    int num_checks = 0;       // Number of comparisons made
    for (int i = 0; i < graph_count; i++) {
        for (int j = i + 1; j < graph_count; j++) {
            // Try optimization for trees, if flag --opt_tree was set
            bool opt_tree_success =
                    opt_tree_flag ?
                    try_tree_optimization(graphs[i], graphs[j], n, is_isomorphic, &total_time)
                    : false;

            // If no optimization successed, then run common nauty algorithm
            if (!opt_tree_success) {
                total_time += check_isomorphism_nauty(graphs[i], graphs[j], n, is_isomorphic);
            }

            num_checks++;
        }
    }

    return total_time / num_checks;
}
```

Graphs in a set are processed pairwise. If the `--opt_tree` flag is provided, tree optimization is attempted. If this fails or the flag is not set, the nauty algorithm is used instead. Processing time for each graph pair is measured, and the average time is returned. All graphs at this stage are in nauty format.

4.3 Nauty Graph Processing

Isomorphism detection using nauty is performed by the following function:

```
double check_isomorphism_nauty(graph *graph1, graph *graph2, const int n, bool should_be_isomorphic)
    DYNALLSTAT(int,lab1,lab1_sz);
    DYNALLSTAT(int,lab2,lab2_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    DYNALLSTAT(int,map,map_sz);
    DYNALLSTAT(graph,g1,g1_sz);
    DYNALLSTAT(graph,g2,g2_sz);
    DYNALLSTAT(graph,cg1,cg1_sz);
    DYNALLSTAT(graph,cg2,cg2_sz);
    static DEFAULTOPTIONS_GRAPH(options);
    statsblk stats;
    options.getcanon = TRUE;
```

```
    const int m = SETWORDSNEEDED(n);
    nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);
    DYNALLOC1(int,lab1,lab1_sz,n,"malloc");
    DYNALLOC1(int,lab2,lab2_sz,n,"malloc");
    DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
    DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
    DYNALLOC1(int,map,map_sz,n,"malloc");
    DYNALLOC2(graph,g1,g1_sz,n,m,"malloc");
    DYNALLOC2(graph,g2,g2_sz,n,m,"malloc");
    DYNALLOC2(graph,cg1,cg1_sz,n,m,"malloc");
    DYNALLOC2(graph,cg2,cg2_sz,n,m,"malloc");

    // Create canonical graphs
    const clock_t start = clock();
    densenauty(graph1,lab1,ptn,orbits,&options,&stats,m,n,cg1);
    densenauty(graph2,lab2,ptn,orbits,&options,&stats,m,n,cg2);

    // Compare canonically labelled graphs
    size_t k;
    for (k = 0; k < m*(size_t)n; ++k)
        if (cg1[k] != cg2[k]) break;
    bool is_isomorphic = k == m*(size_t)n;
    if (should_be_isomorphic != is_isomorphic) {
        printf("Error: graphs should be is_isomorphic=%hhd but was is_isomorphic=%hhd\n");
        exit(EXIT_FAILURE);
    }
    const clock_t end = clock();

    return (double)(end - start) / CLOCKS_PER_SEC;
}
```

This function, partially based on code from *nauty and Traces User's Guide* [BDM24], computes the canonical labeling of each graph and compares the results. The measured time includes both canonical labeling computations and their comparison. Graphs are assumed to be in nauty's internal format.

4.4 MyGraph Implementation

For tree-based optimizations, a simplified graph structure is used to improve code clarity and algorithm implementation. The nauty graph format is converted to this custom structure:

```
typedef struct {
    int id;
    int *neighbours;
    int neighbour_count;
} myVertex;

typedef struct {
    myVertex *vertices;
    int vertex_count;
} myGraph;
```

4.5 Tree Optimization

Tree optimization serves as an alternative to nauty's algorithm for detecting isomorphisms between trees.

```
bool try_tree_optimization(graph *graph1, graph *graph2, const int n,
```

```
                                                    bool should_be_isomorphic, double *total_time) {
    // Check if first graph is a tree
    myGraph *my_graph1 = convert_nauty_to_mygraph(graph1, n);

    const clock_t first_tree_check_start = clock();
    bool is_tree_first = is_tree(my_graph1);
    const clock_t first_tree_check_end = clock();

    *total_time += (double)(first_tree_check_end - first_tree_check_start) / CLOCKS_PER_SEC;
    if (!is_tree_first) return false;

    // Check if second graph is a tree
    myGraph *my_graph2 = convert_nauty_to_mygraph(graph2, n);

    const clock_t second_tree_check_start = clock();
    bool is_tree_second = is_tree(my_graph2);
    const clock_t second_tree_check_end = clock();

    *total_time += (double)(second_tree_check_end - second_tree_check_start) / CLOCKS_PER_SEC;
    if (!is_tree_second) return false;

    // Run tree isomorphism algorithm
    const clock_t isomorphism_check_start = clock();
    bool result = check_isomorphism_tree(my_graph1, my_graph2);
    const clock_t isomorphism_check_end = clock();

    *total_time += (double)(isomorphism_check_end - isomorphism_check_start) / CLOCKS_PER_SEC;

    // Check for error
    if (result != should_be_isomorphic) {
        printf("Error: graphs should be is_isomorphic=%hhd but was is_isomorphic=%hhd\n");
        exit(EXIT_FAILURE);
    }

    // Free memory
    free_mygraph(my_graph1);
    free_mygraph(my_graph2);

    return result;
}
```

Processing begins by converting nauty-format graphs to the `myGraph` format. This conversion time is excluded from the reported processing time.

Each graph is then checked to determine whether it is a tree. If either graph is not a tree, the optimization fails, and the nauty algorithm is used instead. Tree detection time is included in the total processing time.

If both graphs are confirmed as trees, a specialized tree isomorphism algorithm is used. Time taken for this step is also added to the total processing time.

4.5.1 Tree Detection

The tree detection algorithm is implemented based on a linear-time method described on this website.

Short description of the algorithm: The algorithm determines whether an undirected graph is a tree by verifying two key properties: the graph must be connected and acyclic.

1. `Cycle Detection via DFS`: The algorithm initiates a depth-first search (DFS) from vertex 0, marking visited vertices. During traversal, if it encounters an adjacent vertex that has already

been visited and is not the parent of the current vertex, a cycle is detected, and the function returns *false*.

2. `Connectivity Check:` After DFS, the algorithm iterates through all vertices to ensure each has been visited. If any vertex remains unvisited, the graph is disconnected, and the function returns *false*.

3. `Conclusion:` If the graph is both connected and acyclic, the function concludes that the graph is a tree and returns *true*.

4.5.2 Tree Center(s)

The algorithm for determining the center of a tree is described on this website and is required for the tree isomorphism algorithm described in the next section (4.5.3).

Short description of the algorithm: The algorithm operates by iteratively removing leaf nodes (nodes with degree 1) until only one or two nodes remain. These remaining nodes are identified as the tree's center(s).

1. `Initialization:` The degree of each node is calculated based on its number of neighbors. Leaf nodes are identified and stored in a list.

2. `Leaf Removal Process:` In each iteration, the algorithm processes all current leaves. For each leaf, it decrements the degree of its neighbors. If any neighbor's degree becomes 1, it is added to a list of new leaves. This process continues until all nodes except the centers are removed.

3. `Termination:` The loop stops when all leaves are removed, and the remaining nodes, either one or two, are the centers of the tree.

4.5.3 Tree Isomorphism

The algorithm for detecting isomorphism between trees in linear time is based on the description from this website and is implemented as part of the application.

Short description of the algorithm: The algorithm determines whether two trees are isomorphic by utilizing the AHU (Aho–Hopcroft–Ullman) method, which involves the following steps:

1. `Center Identification:` For each tree, the algorithm identifies its center(s). A tree can have one or two centers, which are nodes that minimize the maximum distance to all other nodes. The algorithm for determining the center(s) is described in the previous section (4.5.2).

2. `Tree Encoding:` Each tree is rooted at its center(s), and a unique encoding is generated for the rooted tree. This encoding is constructed by recursively encoding each subtree and concatenating the sorted encodings of the child subtrees, enclosed within parentheses. This process ensures that structurally identical trees yield identical encodings, regardless of node labeling.

3. `Isomorphism Check:` The algorithm compares the encodings of the two trees. If any pair of encodings (one from each tree) match exactly, the trees are considered isomorphic.

4.6 Processed CSV Specification

An example structure of the output 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.

5 Visualization

Visualization is implemented as a Python script utilizing the `pandas` library for CSV file parsing and `matplotlib` for plotting graphs.

5.1 Visualization Function

The core function used for visualization is as follows:

```
import pandas as pd
import matplotlib.pyplot as plt

def visualize(data_file, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    output_file = os.path.join(output_dir, "data.png")
    output_file_scaled_y = os.path.join(output_dir, "data_scaled_y.png")
    output_file_scaled_xy = os.path.join(output_dir, "data_scaled_xy.png")

    # Read the CSV file
    df = pd.read_csv(data_file, delimiter=',')

    # Split data based on is_isomorphic value
    iso_data = df[df['is_isomorphic'] == True]
    non_iso_data = df[df['is_isomorphic'] == False]
    only_isomorphic = non_iso_data.empty

    # Create the plot
    plt.figure(figsize=(20, 12))

    # Plot isomorphic results
    plt.scatter(iso_data['node_count'], iso_data['average_time'],
                                            color='blue', marker='o', label='Isomorphic')

    # Plot non-isomorphic results
    if not only_isomorphic:
        plt.scatter(non_iso_data['node_count'], non_iso_data['average_time'],
                                            color='red', marker='x', label='Non-Isomorphic')

    # Add labels and legend
    plt.xlabel('Node Count')
    plt.ylabel('Average Time (seconds)')
    plt.legend()
    plt.grid(True)

    # Save the plot as an image
    plt.savefig(output_file, dpi=300, bbox_inches='tight')
    print(f"Plot saved as '{output_file}'")

    # Set y-axis to logarithmic scale
    plt.yscale('log')

    # Save the scaled-y plot as an image
    plt.savefig(output_file_scaled_y, dpi=300, bbox_inches='tight')
    print(f"Scaled-y plot saved as '{output_file_scaled_y}'")

    # Set x-axis to logarithmic scale
    plt.xscale('log')
```

```
    # Save the scaled-xy plot as an image
    plt.savefig(output_file_scaled_xy, dpi=300, bbox_inches='tight')
    print(f"Scaled-xy plot saved as '{output_file_scaled_xy}'")
```

The provided CSV file is read using `pandas`, and the data is split into two subsets based on the `is_isomorphic` column: one for isomorphic graph sets and another for non-isomorphic ones. The data file must conform to the specification described in section 4.6.

The script then generates a scatter plot of average processing time versus node count. This visualization is exported in three formats:

- A standard linear scale plot.
- A plot with a logarithmic scale applied to the y-axis.
- A plot with logarithmic scales on both axes (x and y).

These plots help visualize how processing time scales with graph size, distinguishing between isomorphic and non-isomorphic datasets.

6 Estimation Script

The estimation stage is implemented as a Python script using the following libraries:

- `pandas` for reading from and writing to CSV files,
- `numpy` for mathematical operations,
- `scipy` for estimating parameters of fitted functions,
- `sklearn` for computing evaluation metrics,
- `matplotlib` for visualization.

6.1 Estimation Functions

The following functions are used to model and evaluate the time complexity of the graph isomorphism detection process:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from sklearn.metrics import r2_score

def compute_metrics(y_true, y_pred):
    rss = np.sum((y_true - y_pred) ** 2)
    r2 = r2_score(y_true, y_pred)
    return rss, r2

def exponential(x, a, b, c):
    return a * b ** (c * x)


def quasi_polynomial(log_x, alpha, beta, c):
    return alpha + beta * (log_x ** c) * np.log(2)


def polynomial3(x, a):
    return a * x ** 3
```

```
def polynomial2(x, a):
    return a * x ** 2


def polynomial(x, a, b):
    return a * x ** b

def estimate(data_file, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    output_picture = os.path.join(output_dir, "estimation.png")
    output_metrics = os.path.join(output_dir, "metrics.csv")
    data = pd.read_csv(data_file, delimiter=',').sort_values(by='node_count')
    x = np.array(data['node_count'])
    y = np.array(data['average_time'])

    log_x = np.log(x)
    log_y = np.log(y)

    # Fit the data
    popt_exp = curve_fit(exponential, x, y,
                         p0=(1, 1, 1))
    popt_quasi = curve_fit(quasi_polynomial,
                           log_x, log_y,
                           bounds=([-np.inf, 0, 1.5], [np.inf, np.inf, np.inf]))
    popt_poly = curve_fit(polynomial, x, y)
    popt_poly2 = curve_fit(polynomial2, x, y)
    popt_poly3 = curve_fit(polynomial3, x, y)

    # Generate predictions
    y_exp = exponential(x, *(popt_exp[0]))
    y_quasi = np.exp(quasi_polynomial(log_x, *(popt_quasi[0])))
    y_poly = polynomial(x, *(popt_poly[0]))
    y_poly2 = polynomial2(x, *(popt_poly2[0]))
    y_poly3 = polynomial3(x, *(popt_poly3[0]))

    # Count and save metrics as CSV
    metrics = []
    for func_name, y_pred, popt in zip(
            ["exponential", "quasi_polynomial", "polynomial", "polynomial2", "polynomial3"],
            [y_exp, y_quasi, y_poly, y_poly2, y_poly3],
            [popt_exp, popt_quasi, popt_poly, popt_poly2, popt_poly3]
    ):
        rss, r2 = compute_metrics(y, y_pred)
        params = list(popt[0]) + [None] * (3 - len(popt[0]))
        metrics.append([func_name, rss, r2, *params])

    metrics_df = pd.DataFrame(metrics, columns=["functionName", "RSS", "r2", "parameter1",
                                                "parameter2", "parameter3"])
    metrics_df.to_csv(output_metrics, index=False)
    print(f"Metrics saved as '{output_metrics}'")

    # Plot the data and fits
    plt.figure(figsize=(16, 12))
    plt.scatter(x, y, label="Data", color="black", marker=".")
```

```
    plt.plot(x, y_exp, label="Exponential fit", color="blue")
    plt.plot(x, y_quasi, label="Quasi-Polynomial fit", color="red")
    plt.plot(x, y_poly, label="Polynomial fit", color="green")
    plt.plot(x, y_poly2, label="Polynomial2 fit", color="purple")
    plt.plot(x, y_poly3, label="Polynomial3 fit", color="brown")
    plt.legend()
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Function Fits")
    plt.savefig(output_picture, dpi=600, bbox_inches='tight')
    print(f"Plot saved as '{output_picture}'")
```

First, the processed CSV file is read and sorted by the number of nodes. Data is then split into two arrays: `x` for node counts and `y` for execution times. The script fits five types of functions to this data:

- Exponential
- Quasi-polynomial (fitted using log-transformed inputs and outputs)
- General polynomial
- Quadratic polynomial
- Cubic polynomial

For each function, predictions are generated and evaluation metrics—Residual Sum of Squares (RSS) and R^2 score—are computed and stored in a CSV file. The fitted curves are plotted against the original data, and the resulting figure is saved as a PNG image.

7 Additional Scripts

This section describes auxiliary scripts used for generating datasets of graphs with specific structural properties. These scripts primarily handle file conversions and graph generation tasks using SageMath and shell utilities.

7.1 adj_to_graph6.py

```
def read_adjacency_matrix(filename):
    with open(filename, "r") as f:
        lines = f.readlines()
        matrix = [list(map(int, line.strip())) for line in lines if line.strip()]
    return matrix

def convert_all_graphs(input_dir, output_dir):
    if not os.path.isdir(input_dir):
        print(f"Input directory '{input_dir}' does not exist.")
        sys.exit(1)

    os.makedirs(output_dir, exist_ok=True)

    # Read all txt files
    for filename in os.listdir(input_dir):
        if filename.endswith(".txt"):
            input_path = os.path.join(input_dir, filename)
            # If error occurred during file converting, skip this file
            try:
                # Read txt file
```

```
            matrix = read_adjacency_matrix(input_path)

            # Convert to Sage Matrix and then to Sage Graph
            M = Matrix(matrix)
            G = Graph(M, format='adjacency_matrix')

            # Take name of original file and add .g6 extension
            base_name = os.path.splitext(filename)[0]
            output_path = os.path.join(output_dir, base_name + ".g6")

            # Save graph6 file
            with open(output_path, "w") as f:
                f.write(G.graph6_string() + "\n")

            print(f"[OK] {filename} → {base_name}.g6")
        except Exception as e:
            print(f"[ERROR] Failed to process {filename}: {e}")
```

This script reads all `.txt` files in a specified directory, interprets each file as an adjacency matrix, and constructs a Sage graph object. It then converts the graph to the graph6 format using Sage's built-in functionality. The resulting Graph6 string is saved to a new file with the same base name but with a `.g6` extension.

7.2 edges_to_graph6.py

```
def read_edges_from_file(filename):
    edges = []
    with open(filename, "r") as f:
        # Skip first line. Boltzmann generator writes additional info there.
        next(f)

        # Read and process each line (each edge)
        for line in f:
            line = line.strip()
            # Skip last line. Boltzmann generator puts "0 0" there.
            if line == "" or line.startswith("#") or line.startswith("0 0"):
                continue

            # Add edge to list of edges
            parts = line.split()
            if len(parts) != 2:
                raise ValueError(f"Invalid edge line: {line}")
            u, v = map(int, parts)
            edges.append((u, v))

    return edges

def main():
    if len(sys.argv) < 3:
        print("Usage: sage -python edges_to_graph6.py <input_file> <output_dir>")
        sys.exit(1)

    input_file = sys.argv[1]
    output_dir = sys.argv[2]

    edges = read_edges_from_file(input_file)
```

```
    # Create Sage Graph from edges list
    G = Graph(edges)

    # All graphs with more than 2000 nodes are ignored
    if G.order() > 2000:
        print(f"Skip graph with {G.order()} vertices")
        return

    # Save graph6
    os.makedirs(output_dir, exist_ok=True)
    output_path = os.path.join(output_dir, f"{G.order()}.g6")
    with open(output_path, "w") as f:
        f.write(G.graph6_string() + "\n")

    print(f"Graph saved in graph6 format to {output_path}")
```

This script is used in planar graph generation (see section 7.4). It reads a text file containing a list of graph edges (excluding the first and last lines), constructs a graph using Sage, and saves it in Graph6 format. The output file is named according to the number of nodes in the graph (`G.order()`). Graphs with more than 2000 nodes are ignored to restrict the dataset to graphs of small size.

7.3 graph6_iso_dup.py

```
SET_SIZE = 3

def read_graph6_file(filename):
    with open(filename, 'r') as f:
        line = f.readline().strip()
        return Graph(line, format='graph6')

def shuffle_labels(graph):
    permutation = list(graph.vertices())
    random.shuffle(permutation)
    return graph.relabel(permutation, inplace=False)

def duplicate_all_graphs(input_dir, output_dir):
    if not os.path.isdir(input_dir):
        print(f"Input directory '{input_dir}' does not exist.")
        sys.exit(1)
    os.makedirs(output_dir, exist_ok=True)

    # Read all g6 files
    for filename in os.listdir(input_dir):
        if filename.endswith(".g6"):
            input_path = os.path.join(input_dir, filename)
            # If graph cannot be duplicated, then skip it
            try:
                # Read graph from file and shuffle its labeling
                original_graph = read_graph6_file(input_path)
                graph_set = [original_graph] +
                            [shuffle_labels(original_graph) for _ in range(SET_SIZE - 1)]

                # Save to file with the same name
                base_name = os.path.splitext(filename)[0]
                output_path = os.path.join(output_dir, base_name + ".g6")

                with open(output_path, "w") as file:
```

```
                for g in graph_set:
                    file.write(g.graph6_string() + "\n")

            print(f"[OK] {filename} → {base_name}.g6")
        except Exception as e:
            print(f"[ERROR] Failed to process {filename}: {e}")
```

This script reads each file from a specified input directory, assuming each file contains a single graph in graph6 format. For each graph, it generates multiple isomorphic copies by permuting the vertex labels and saves them under the original filename, effectively creating duplicates with different labelings.

7.4 planar_generator.sh

```
for i in {1..500}; do
  # Use Boltzmann planar graph generator
  cd BoltzmannPlanarGraphs/build/classes
  echo 1000 | java boltzmannplanargraphs.Main

  # Convert Boltzmann graph from edges to graph6 format
  cd ../../..
  sage -python edges_to_graph6.py BoltzmannPlanarGraphs/ListEdges.txt graph6_single/planar/
done
```

This is a Bash script that runs an implementation of the Boltzmann planar graph generator [Fus09]. The output edge lists are then converted to Graph6 format using the script described in section 7.2. The script executes 500 iterations to produce a diverse sample of randomly generated planar graphs of varying sizes.

Example output from the Boltzmann generator:

```
PIG:0 Graph
2 1
3 1
0 0
```

7.5 srg_from_params.py

```
def generate_graphs_from_csv(csv_file, output_dir):
    os.makedirs(output_dir, exist_ok=True)

    with open(csv_file, newline='') as file:
        reader = csv.DictReader(file)

        # Read each line of csv file
        for row in reader:
            # If SRG graph cannot be generated from the given parameters, then skip
            try:
                # Fetch parameters
                v = int(row['v'])
                k = int(row['k'])
                l = int(row['lambda'])
                mu = int(row['mu'])

                # Try to generate the graph
                G = strongly_regular_graph(v, k, l, mu)

                # Save graph
                output_path = os.path.join(output_dir, f"{v}.g6")
```

```
            with open(output_path, "w") as out:
                out.write(G.graph6_string() + "\n")

            print(f"[OK] Generated graph with v={v} saved to {output_path}")
        except Exception as e:
            print(f"[ERROR] Could not generate graph for v={row.get('v')}: {e}")
```

This script processes a CSV file containing parameter sets for strongly regular graphs (SRGs). For each set of parameters, it attempts to construct the corresponding SRG using Sage's `strongly_regular_graph` function, which queries an internal database. Each successfully generated graph is saved in graph6 format, named by its node count.

The parameter data used in this script is available on the project's GitHub repository. The SRG parameters were sourced from an external database.

References

[BDM24] Adolfo Piperno Brendan D. McKay. nauty and traces user's guide (version 2.8.9), 2024.

[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 user's guide, 2025.