SCHENO's Nauty and Traces C++ Wrappers

(and a bonus approximate isomorphism algorithm)

Justus Hibshman

March 2024

Outline

[1]These wrappers originally come from the SCHENO project. SCHENO (SCHEma NOise) measures how well one graph represents the underlying patterns (schema) in another graph. The idea behind SCHENO is that a real-world graph is a noisy manifestation of an underlying pattern. If you partition a graph's edges and non-edges into two sets, the schema set and the noise set, SCHENO measures how structured the schema set is, how random the noise set is, and how well the two sets represent the original graph – all in a single numeric score. You can find the code here: **LINK**

1 Introduction to Nauty and Traces

`Nauty` and `Traces` are graph isomorphism programs developed by Brendan McKay and Adolfo Piperno. As of the writing of this document, their code was available at `LINK`. A version of it is also available directly in this repository.

Perhaps surprisingly to a newcomer, `nauty` and `traces` do not directly compare two graphs to see if they are isomorphic. Rather, they provide the following features for a single graph:

1. Find the automorphism orbits of the nodes in the graph.
2. Find the size of the graph's automorphism group.
3. Find a canonical ordering of the nodes.
4. Do all of the above given an initial node coloring.

The third feature (canonical node ordering) can be used to compare two graphs to see if they are isomorphic. You can do so as follows: Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ be graphs. Assume a canonical node ordering o for a graph $G(V, E)$ is a bijective function $o : V \to [\ |V|\ ]$ where $[n]$ denotes the numbers 1 through n. Let o_1 and o_2 be canonical node orderings for G_1 and G_2 respectively. Then G_1 and G_2 are isomorphic if and only if:

$$\{(o_1(a),\ o_1(b)) \mid (a, b) \in E_1\} = \{(o_2(a),\ o_2(b)) \mid (a, b) \in E_2\}$$

Another way to think about this is that G_1 and G_2 are only isomorphic if the i'th canonical node connects to the j'th canonical node in both graphs or in neither graph for any pair i, j.

2 Features of This Repository

The code in this repository offers `C++` wrappers around the `nauty` and `traces C` code. These wrappers provide the following:

1. A more convenient interface for the `nauty` and `traces` features mentioned in Section 1.
2. The ability to call `nauty` and `traces` in multiple `C++` threads without errors.
3. The ability to find the automorphism orbits of *edges*[2].
4. The ability to conveniently manipulate graphs in `nauty` and `traces`' input format in amortized constant time, rather than needing to re-create the graph every time you want to modify it.
5. The ability to specify an initial *edge* coloring[2].
6. The ability to run `traces` on directed graphs[2].
7. A fast "fake" isomorphism algorithm that works on most graphs – especially most graphs with varying node degrees[3].

[2] not a direct feature of the original code – requires an augmentation of the input graph

[3] not associated with nauty or traces – based on the Weisfeiler Lehman algorithm (aka color refinement)

3 Setup and Compilation

Setting up the code is a simple as running: `./nauty_traces_setup.sh`

Whatever code you write will need to include `nauty_traces.h`. You may also want to use `file_utils.h` for reading and writing `SparseGraph`'s from/to files. The details of these are discussed in Section 4.

To compile your own code using the wrappers, you will need to include the following files:

- `nt_partition.cpp`
- `graph.cpp`
- `sparse_graph.cpp`
- `nt_sparse_graph.cpp`
- `nauty_traces.cpp`

Example compilation:

```
g++ -Wall -Wextra -o my_output -std=c++11 my_program.cpp
    nt_partition.cpp graph.cpp sparse_graph.cpp nt_sparse_graph.cpp
    nauty_traces.cpp nauty27r4_modified/nauty.a
```

If you want to use the code in `file_utils.h`, then make sure to add `file_utils.cpp` to the list.

3.1 Esoteric Note

If for some reason you want to experience the pain modifying the `NTSparseGraph` code and you want to debug it using the `SCHENO__NT_SPARSE_GRAPH_FULL_DEBUG_MODE` preprocessor flag, then make sure to include `debugging.cpp` in the list as well.

You can see an example of using this feature in `test_nt_code.cpp`. The flag is defined in `nt_sparse_graph.h`.

4 Interface

4.1 Essential Components

The key classes and structs you will need are `NTSparseGraph` which stores a graph that can be loaded directly into `nauty` and/or `traces`, `NautyTracesOptions` which stores info on what you would like `nauty/traces` to compute, and `NautyTracesResults` which contains the result of a call to `nauty/traces`.

4.1.1 NTSparseGraph

`NTSparseGraph` is a subclass of `SparseGraph`, which is a subgraph of `Graph`. The `NTSparseGraph` and the `SparseGraph` always label their n nodes 0 through $n-1$.

There are three main ways to get an `NTSparseGraph`. The first is to load an edge list text file (see Section 4.4), then initialize your `nauty/traces` graph. For example:

```
#include "file_utils.h"
#include "nauty_traces.h"

bool directed = 0;
NTSparseGraph g_nt(read_graph(directed, "nt_test_graphs/karate.txt"));
```

The second way is to build up the graph from scratch using the `add_edge()` function. For example:

```
#include "nauty_traces.h"

bool directed = 1;
size_t n = 12;
NTSparseGraph g_nt(directed, n);
g_nt.add_edge(0, 5);
g_nt.add_edge(3, 5);
g_nt.add_edge(5, 3);
g_nt.add_edge(11, 10);
```

The third way is to copy a pre-existing `SparseGraph` or `NTSparseGraph`. For example:

```
#include "nauty_traces.h"

bool directed = 1;
SparseGraph g(directed);  // Defaults to 1 node
g.add_node();
g.add_node();
g.add_edge(0, 1);
g.add_edge(1, 2);
g.add_edge(2, 0);

// Three ways of copying a graph:
NTSparseGraph g_nt_1(g);
NTSparseGraph g_nt_2(directed);
g_nt_2 = g;
NTSparseGraph g_nt_3(g_nt_1);
```

The methods from the `NTSparseGraph` class that you are most likely to use are the following:

- `NTSparseGraph(bool directed)`
- `NTSparseGraph(bool directed, size_t num_nodes)`
- `NTSparseGraph(const Graph& g)`

- `size_t num_nodes()`
- `size_t num_edges()`
- `size_t num_loops()` – Returns the number of self-loops
- `directed` – Not a function - just a constant boolean
- `int add_node()` – Adds a node and returns the new node's ID
- `int delete_node(int a)` – Deletes node a. Relabels the node with the largest label to have the label a. Then returns what used to be the label of what used to be the largest node.
- `bool add_edge(int source, int target)` – Returns true iff the edge was new
- `bool delete_edge(int source, int target)` – Returns true iff the edge was there to be deleted
- `void flip_edge(int s, int t)` – Deletes edge (s, t) if it was present, adds edge (s, t) if it was absent.
- `const std::unordered_set<int> &neighbors(int a)` – Returns an unordered set (`C++` standard library) of all the nodes connected to node a
- `const std::unordered_set<int> &out_neighbors(int a)` – Returns all the nodes that node a points to. In an undirected graph, this returns the same thing as `neighbors(a)`.
- `const std::unordered_set<int> &in_neighbors(int a)` – Returns all the nodes that point to node a. In an undirected graph, this returns the same thing as `neighbors(a)`.

4.1.2 NautyTracesOptions

The `NautyTracesOptions` struct contains three boolean fields:

- `get_node_orbits`
- `get_edge_orbits`
- `get_canonical_node_order`

They are largely self-explanatory. If they are set to true, then a call to `nauty/traces` using these options will populate the corresponding fields in the relevant `NautyTracesResults` struct. Setting them to false may improve runtime, so only set them to true when you want the information.

4.1.3 NautyTracesResults

The `NautyTracesOptions` struct stores the output of a `nauty/traces` computation. It has the following fields:

- `int error_status` – Will be non-zero if an error occurred
- `size_t num_node_orbits` – Number of automorphism orbits of the nodes
- `size_t num_edge_orbits` – Number of automorphism orbits of the edges
- `double num_aut_base`
- `int num_aut_exponent` – The number of automorphisms of the graph is roughly $\text{num\_aut\_base} \times 10^{\text{num\_aut\_exponent}}$.
- `std::vector<int> canonical_node_order` – Stores the node IDs (0 through `num_nodes() - 1`) in a canonical order. This field is only populated when the `get_canonical_node_order` option is used.
- `Coloring<int> node_orbits` – Stores a "coloring" which gives every node a color corresponding to which automorphism orbit it is in. The `Coloring` class is described in Section 4.2. This field is only populated when `get_node_orbits` option is used.
- `Coloring<Edge, EdgeHash> edge_orbits` – Stores a "coloring" which gives every edge a color corresponding to which automorphism orbit it is in. The `Coloring` class is described in Section 4.2 and the `Edge` class is described in Section 4.3. This field is only populated with the `get_edge_orbits` option is used.

4.2 Colorings

Colorings are used to store automorphism orbit information. They can also be used to force automorphisms to match nodes to other nodes of the same color.

4.2.1 Reading a Coloring

If you simply want to read the colorings that `nauty` and `traces` provide, then the only feature you really need is the access operator [].

For example, if `node_col` is a node coloring, then to see what color node 7 is, simply use `node_col[7]`.

Accessing the color of an edge is a tiny bit more complicated. If `edge_col` is an edge coloring and `dir` is a boolean indicating whether or not you have directed edges, then the color of edge (a, b) is accessed as `edge_col[EDGE(a, b, dir)]`. The `EDGE` macro produces an `Edge` (which is really just a `std::pair<int, int>`) that respects a convention the code uses for undirected edges.

There are a few other methods for accessing the `Coloring` class that can be useful:

- `size_t size()` – Returns the number of colored elements.
- `const std::set<int>& colors()` – Returns a set of all the colors in the coloring.
- `const std::unordered_set<T, THash>& cell(int color)` – Returns the set of nodes (type `int`) or edges (type `Edge`) that have the color `color`.

4.2.2 Creating a Coloring

If you want to *create* a `Coloring` for nodes, it should be initialized as follows:

```
Coloring<int> my_node_coloring();
```

If you want to *create* a `Coloring` for edges, it should be initialized as follows:

```
Coloring<Edge, EdgeHash> my_edge_coloring();
```

To set the color of an element or to remove an element from the coloring, use the following two methods:

- `set(const T& elt, int color)` – Sets node or edge `elt` to color `color`.
- `erase(const T& elt)` – Removes node or edge `elt` from the coloring.

4.3 Edge Class for Edge Colorings

The `Edge` class is only needed if you want to work with edge colorings. It is simply a `typedef` for `std::pair<int, int>`. The code requires that undirected edges put the smaller node ID first. The easiest way to create an `Edge` is probably to use the `EDGE(source, target, directed)` macro, where `source` and `target` are integers and `directed` is a boolean.

4.4 File Utils

To load a graph from a file or write a graph to a file, you can use the functions defined in `file_utils.h`:

Read Graph from File Version 1

This function assumes that the nodes are numbered 0 through the largest node ID found in the edge list.

```
SparseGraph read_graph(const bool directed,
                       const std::string& edgelist_filename);
```

Read Graph from File Version 2

This function puts all nodes in the nodelist into the graph, even if they do not appear in the edgelist.
Note that if the nodes in the node list are not labeled 0 through $n-1$, they will be relabeled in sorted order as they are loaded.

```
SparseGraph read_graph(const bool directed,
                       const std::string& nodelist_filename,
                       const std::string& edgelist_filename);
```

Write Graph to File

If `nodelist_filename` is empty then no nodelist is written.
`SparseGraph` and `NTSparseGraph` are both subclasses of `Graph`.

```
void write_graph(const Graph& g, const std::string& nodelist_filename,
                                 const std::string& edgelist_filename);
```

Construct a node list file from an edge list file

Reads the edgelist file and makes a nodelist for it.
If `full_range` is true, then the nodelist will consist of the interval from 0 through the largest node ID in the edgelist. If `full_range` is false, only nodes mentioned in the edgelist will be listed in the nodelist.

```
void make_nodelist(const std::string& edgelist_filename,
                   const std::string& nodelist_filename,
                   bool full_range);
```

5 Calling Nauty or Traces

There are four functions available for calling `nauty` and `traces`.

5.1 Simple Version

Two of the functions simply run the program on the graph:

```
NautyTracesResults nauty(NTSparseGraph& g, const NautyTracesOptions& o)

NautyTracesResults traces(NTSparseGraph& g, const NautyTracesOptions& o)
```

Even though the graph `g` is not passed as a constant reference, it is effectively left un-modified from the perspective of the user.

5.2 Extra Options for Constraining Automorphisms

The other two functions allow you to specify a partitioning or "coloring" that the automorphisms must respect, meaning that nodes (or edges) of color c can only be mapped to other nodes (or edges) with the same color c.

```
NautyTracesResults nauty(NTSparseGraph& g, const NautyTracesOptions& o,
                         NTPartition& p)

NautyTracesResults traces(NTSparseGraph& g, const NautyTracesOptions& o,
                          NTPartition& p)
```

To get one of these partitionings, you use one of the `nauty_traces_coloring()` methods of an `NTSparseGraph` to convert a node and/or edge coloring into a partition. Note that the partition object might be modified by the `nauty` or `traces` call.

The three methods converting colorings to a partitioning of a *particular* graph are as follows:

```
NTPartition nauty_traces_coloring(const Coloring<int> &node_colors)

NTPartition nauty_traces_coloring(const Coloring<Edge, EdgeHash> &edge_colors)

NTPartition nauty_traces_coloring(const Coloring<int> &node_colors,
                                  const Coloring<Edge, EdgeHash> &edge_colors)
```

Note that you **must** call these methods on the `NTSparseGraph` object which you are going to use the partition for. Even if you use a graph which is identical from the perspective of the user (i.e. same edge set), the partition could be invalid because the hidden `nauty`/`traces` representation might have been constructed in a different order, leading to different hidden node labels.

6 Simple Example

```
// minimal_nt_example.cpp

#include "file_utils.h"
#include "nauty_traces.h"

#include<cmath>
#include<iostream>

int main(void) {

    bool directed = false;
    NTSparseGraph karate(directed);
    karate = read_graph(directed, "nt_test_graphs/karate.txt");

    std::cout<<"# Nodes: "<<karate.num_nodes()<<std::endl;
    std::cout<<"# Edges: "<<karate.num_edges()<<std::endl<<std::endl;

    NautyTracesOptions nto;
    nto.get_node_orbits = true;
    nto.get_edge_orbits = true;
    nto.get_canonical_node_order = true;

    NautyTracesResults ntr = traces(karate, nto);

    double log10_aut = std::log10(ntr.num_aut_base) + ntr.num_aut_exponent;
    std::cout<<"Log10 of Automorphisms:  "<<log10_aut<<std::endl;
    std::cout<<"Number of Automorphisms: "<<(std::pow(10.0, log10_aut))<<std::endl;
    std::cout<<"Number of Node Orbits:   "<<ntr.node_orbits.colors().size()<<std::endl;
    std::cout<<"Number of Edge Orbits:   "<<ntr.edge_orbits.colors().size()<<std::endl;
    std::cout<<"First Node in Canonical Ordering: "
             <<ntr.canonical_node_order[0]<<std::endl;

    return 0;
}
```

The above example can be compiled with the following command:

```
g++ -Wall -Wextra -o minimal_nt_example -std=c++11 minimal_nt_example.cpp
        nt_partition.cpp graph.cpp sparse_graph.cpp nt_sparse_graph.cpp
        nauty_traces.cpp file_utils.cpp nauty27r4_modified/nauty.a
```

7 Details Concerning fake_iso

This repository contains a third isomorphism algorithm dubbed "`fake_iso`" which is unafilliated with `nauty` or `traces`. The `fake_iso` algorithm is known to fail on some graphs, but works on the majority of graphs.

Because `fake_iso` does not pursue a formal guarantee of correctness, it can sometimes be orders of magnitude faster than `nauty` and/or `traces`.

Depending on your goals, `fake_iso` may be your best option.

7.1 Interface

The interface for `fake_iso` is *exactly* the same as the interface for `nauty` and `traces`. To use `fake_iso`, just replace your call to `nauty` or `traces` with a call to `fake_iso`.

7.2 Algorithm Details and Failure Cases

The `fake_iso` algorithm returns the 1-dimensional Weisfeiler Lehman (aka 1DWL) coloring as the set of automorphism orbits. Two nodes (or edges) who shouldn't be in the same orbit could be listed as being part of the same orbit, but two nodes (or edges) who should be in the same orbit will never be listed in separate orbits.

To get a "canonical" node order, `fake_iso` repeatedly runs 1DWL, then selects a node from a non-trivial orbit and gives it a new, unique color. Then 1DWL is re-run, further refining the colors. This process of "refine, select, refine, select," continues until all nodes have their own color/orbit/partition-cell.

The runtime of `fake_iso` is guaranteed to be polynomial in the size of the graph.

The estimate for the size of the automorphism group could be too large but will never be too small. On many graphs, including many real-world graphs, the value is exactly correct.

1DWL is known to fail on regular graphs[4] with more than one node orbit. There may be other failure cases as well. In short, things that look like symmetries but turn out not to be can cause `fake_iso` to incorrectly treat nodes (or edges) is if they are automorphically equivalent.

[4]meaning all nodes have the same degree