SCHENO's Binaries and C++ Classes

Justus Hibshman

April 2024

Outline

1 Introduction to SCHENO

The core idea behind `SCHENO` is that real-world graphs are messy manifestations of underlying patterns. `SCHENO` offers a principled way to measure how well you've done at uncovering those patterns.

The formal definitions and explanations can be read in the article located at: **LINK**

2 Features of This Repository

This repository offers the following components:

1. An efficient `C++` implementation of the `SCHENO` scoring function, available in a static library and as an executable.
2. Code to obtain `SCHENO` scores for many schema-noise decompositions in parallel.
3. A genetic algorithm "`SCHENO_ga`" that searches for good schema-noise decompositions, guided by `SCHENO` as its fitness function.
4. Elegant `C++` wrappers around the `nauty` and `traces` isomorphism algorithms[1].

[1] These wrappers are available in a standalone repository at `https://github.com/schemanoise/Nauty_and_Traces`

3 Setup and Compilation

Setting up the code is a simple as running `./setup.sh` to compile the libraries and then running `./compile.sh` to compile the executables.

Whatever custom code you write will need to include the file `scheno.h` from inside the `scheno` folder.

To compile your own code, you will need to include the static library `scheno.a` located inside the `scheno` folder.

Example compilation:

```
g++ -Wall -Wextra -o my_output -std=c++11 my_program.cpp scheno/scheno.a
```

4 Using the Executables

4.1 SCHENO_score

4.2 SCHENO_ga

5 Interface

5.1 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`.

5.1.1 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 5.4), then initialize your `nauty/traces` graph. For example:

```
#include "nt_wrappers/nauty_traces.h"

int main( void ) {
    bool directed = 0;
    NTSparseGraph g_nt(read_graph(directed, "nt_test_graphs/karate.txt"));
    return 0;
}
```

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

```
#include "nt_wrappers/nauty_traces.h"

int main( void ) {
    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);
    return 0;
}
```

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

```
#include "nt_wrappers/nauty_traces.h"

int main( void ) {
    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);
    return 0;
}
```

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)`.

5.1.2 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.

5.1.3 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 5.2. 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 5.2 and the `Edge` class is described in Section 5.3. This field is only populated with the `get_edge_orbits` option is used.

5.2 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.

5.2.1 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`.

5.2.2 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.

5.3 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.

5.4 File Utils

To load a graph from a file or write a graph to a file, you can use the following functions:

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);
```

6 Simple Example

```
// minimal_nt_example.cpp

#include "nt_wrappers/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_wrappers/nt_wrappers.a
```