SCHENO's Binaries and C++ Classes

Justus Hibshman

April 2024

Outline

- 1 Introduction to SCHENO
- 2 Features of this Repository
- 3 Setup and Compilation
- 4 Using the Executables
- 5 Interface

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 dissertation located at: https://www.proquest.com/docview/2916028057

1.1 Notation

Let A and B be sets. I use $A \oplus B$ to mean $(A \setminus B) \cap (B \setminus A)$. It can be thought of as analogous the logical XOR operation. The set $A \oplus B$ includes the things that are in exactly one of A or B, but not both.

Similarly, given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, I define $G_1 \oplus G_2$ to be the graph $(V, E_1 \oplus E_2)$.

Lastly, given a graph $G = (V, E)$, I define a "Schema-Noise Decomposition" of G to be two graphs $S = (V, E_S)$ and $N = (V, E_N)$ such that $S \oplus N = G$. In other words, if you start with the schema S and then "XOR" its edges with the noise edges, you get the graph G.

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].[2].

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

[2] Frankly, these wrappers provide a much more elegant code interface than the rest of the SCHENO code. If you find the SCHENO code interface crufty, you still may very well like the isomorphism interface.

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

This program requires two things as input: a graph $G = (V, E)$ given as an edgelist and a "noise set" $N = (V, E_N)$ also given as an edge list. The schema is inferred to be $S = G \oplus N$, which is the only edge set such that $S \oplus N = G$.

It gives the SCHENO score for the schema-noise decomposition S, N. It also displays some information about the number of automorphisms of G and S.

Finally, it also reports how much of the decomposition score is due to taking G and deleting all the edges to some nodes, thereby forming "singletons" which are all structurally equivalent to each other, vs. how much of the decomposition score is due to the other ways the schema differs from the graph.

There are many optional flags you can use when running the program. You can see a description of all of them by running `SCHENO_score -h`

A few of the most important flags are:

- `-nodes <arg>` – This flag allows you to specify to text file with a list of node IDs. That way, if your edgelist does not have all the nodes you want, they can be included.
- `-d` and `-u` – These flags allow you to specify whether the graph is directed or undirected. Undirected is the default.
- `-approx` – Causes the program to use a heuristic isomorphism program which might not always return the correct result. This is often correct, and can be significantly faster, particularly if the graph is large enough. If the graph is small enough, it can actually be slower. See more details at the end of the other manual, `nauty_and_traces_wrappers.pdf`.

4.2 SCHENO_ga

This program takes a single graph as input in the form of an edgelist. It tries to find a good schema-noise decomposition of the graph.

The program also takes a filepath argument `-o <f>` to specify its output. It will produce three files: `<f>_graph.txt`, `<f>_noise.txt`, `<f>_nodes.txt`. The graph file stores the schema. The noise file stores the noise. Neither file is guaranteed to reference all the nodes in the original graph – hence the nodelist output file[3].

There are many optional flags you can use when running the program. You can see a description of all of them by running `SCHENO_ga -h`

A few of the most important flags are:

- `-nodes <arg>` – This flag allows you to specify to text file with a list of node IDs. That way, if your edgelist does not have all the nodes you want, they can be included.
- `-d` and `-u` – These flags allow you to specify whether the graph is directed or undirected. Undirected is the default.
- `-n_itr <arg>` – How many iterations the genetic algorithm runs for
- `-topk <arg>` – The number of top-scoring noise sets to report on the command line (only the very best decomposition is written to a file)
- `-sample_heuristic` – `SCHENO_ga` can use a heuristic when randomly sampling edges to add to the noise set. This feature is experimental; it requires a hefty pre-computation step and is not guaranteed to improve performance. You can turn this feature on with this flag.
- `-nt` – Number of threads to use. Defaults to 0, which means max parallelism available on the machine.

[3] Also, output node IDs might not line up with the original input edgelist's node IDs if the original IDs were not 0 through $n - 1$.

- `-approx_iso` – Lets the `SCHENO` score computation make use of an approximate isomorphism algorithm. This can sometimes speed up the computation on medium and large graphs.
- `-seed <arg>` – Lets you specify a starting schema-noise decomposition by giving it an edgelist that corresponds to a noise set. The algorithm will use this as its starting point.
- `-legal_noise <arg>` – Lets you specify limits on which edges and non-edges can be considered as noise. The argument should be a filename; the file should contain a list of node pairs (i.e. an edgelist).

5 Interface

5.1 What You Need for the Scoring Function

The SCHENO scoring function has the following interface:

```
long double score(NTSparseGraph& g,
                  const CombinatoricUtility& comb_util,
                  const Coloring<int>& node_orbit_coloring,
                  const Coloring<Edge, EdgeHash>& edge_orbit_coloring,
                  Coloring<Edge, EdgeHash>& editable_edge_orbit_coloring,
                  const std::unordered_set<Edge,EdgeHash>& edge_additions,
                  const std::unordered_set<Edge,EdgeHash>& edge_removals,
                  const long double log2_p_plus,
                  const long double log2_p_minus,
                  const long double log2_1_minus_p_plus,
                  const long double log2_1_minus_p_minus,
                  const size_t max_change,
                  bool full_iso);
```

You can see an example of using it's sister function, `score_breakdown()` in the file `SCHENO_score.cpp`. The two functions take exactly the same input arguments.

The input graph is `g`.

The `CombinatoricUtility` class is basically a storage unit for math computations that are likely to come up again and again if you score many different decompositions. It's discussed in Section 5.2.

The arguments `node_orbit_coloring` and `edge_orbit_coloring` as well as `editable_edge_orbit_coloring` are required for efficiency reasons. Precomputing them speeds things up if you score multiple different decompositions for a single graph. These colorings can be produced with a call to one of the isomorphism algorithms available in the repository, `nauty`, `traces`, and `fake_iso`. Essentially, the inputs record which nodes and edges in the original graph are structurally equivalent to each other. Make sure that `editable_edge_orbit_coloring` is a different object from `edge_orbit_coloring`.

The `edge_additions` and `edge_removals` arguments represent the noise set being considered in the schema-noise decomposition. They are the edges added to and removed from `g` respectively.

The next four inputs, `log2_p_plus`, `log2_p_minus`, `log2_1_minus_p_plus`, `log2_1_minus_p_minus` are parameters that affect the scoring function's calculation. They affect how likely noise is considered to be. `log2_p_plus` is the base-2 logarithm of the probability that an edge is randomly added to the *schema* (randomly *removed* from `g`). `log2_p_minus` is the base-2 logarithm of the probability that an edge is randomly removed from the *schema* (randomly *added* to `g`). It is strongly recommended that you stick with the default values, as discussed in Section 5.3.

The parameter `max_change` is the maximum number of node pairs you want to allow in the noise set. Any noise set with more than `max_change` node pairs will automatically receive a score of $-\infty$.

Finally, `full_iso` indicates whether or not the scoring function should use a full-fledged isomorphism algorithm (in this case `traces`) or an approximate algorithm "`fake_iso`." A value of `true` means `traces`; a value of `false` means `fake_iso`. The `fake_iso` algorithm often returns the correct result and can be much faster on large enough graphs.

The following classes, structs, and functions are important and are discussed at length in the other pdf manual associated with this repository (`nauty_and_traces_wrappers.pdf`):

- `Graph`, `SparseGraph`, and `NTSparseGraph`
- `NautyTracesOptions` and `NautyTracesResults`
- `Coloring<T>`
- `Edge`
- `read_graph()` and `write_graph()`
- `nauty()`, `traces()`, and `fake_iso()`

5.2 CombinatoricUtility

The combinatoric utility class pre-computes useful values like the logarithms of factorials. Its constructor takes two inputs:

```
CombinatoricUtility(size_t max_e, size_t max_f);
```

max_e must be larger than the max number of edges your graph COULD ever have, including self-loops. max_f must be larger than both of the following:

- the max number of edges your graph WILL have PLUS the number of edges you will delete from it
- (the max number of edges you COULD have MINUS the number of edges your graph WILL have) PLUS the number of edges you will delete from it

It's probably a safe bet to make: max_f $\geq \min(3m,$ max_e $-(3($max_e$-m)))$ where m is the number of edges.

The constructor for the CombinatoricUtility class will take O(max_e) time, and the class's data structures will use O(max_f) space.

5.3 Default Log-Prob Parameters

To get the default values for log2_p_plus, log2_p_minus, log2_1_minus_p_plus, and log2_1_minus_p_minus, use the function:

```
std::vector<long double>
        default_log2_noise_probs(NTSparseGraph& g,
                                 const CombinatoricUtility& comb_util);
```

It returns a vector of the four log-probabilities *in the following order*:

- log2_p_plus
- log2_1_minus_p_plus
- log2_p_minus
- log2_1_minus_p_minus

5.4 The Thread-Pool Scorer

If for some reason you want to score many schema-noise decompositions for the same graph in parallel, you could consider using the ThreadPoolScorer class.

It is initialized in much the same way that the scoring function is called (discussed above):

```
ThreadPoolScorer(size_t nt, const Graph& g,
                 const CombinatoricUtility& comb_util,
                 const Coloring<int>& node_orbit_coloring,
                 const Coloring<Edge,EdgeHash>& edge_orbit_coloring,
                 long double log2_p_plus,
                 long double log2_p_minus,
                 long double log2_1_minus_p_plus,
                 long double log2_1_minus_p_minus,
                 size_t max_size,
                 bool use_heuristic,
                 bool full_iso);
```

There are two parameters not asked for in the `score()` function: `nt` and `use_heuristic`. The number `nt` is the number of threads to use. If you pass 0, the `ThreadPoolScorer` will use the parallelism that the machine offers.

You will almost certainly want to set the `use_heuristic` value to `false`. It was part of an experimental feature to give a fuzzy tie-breaker score meant to say which decompositions were *closer* to getting a better `SCHENO` score even if they had the same *actual* `SCHENO` score.

Once you have initialized your `ThreadPoolScorer` for a particular graph as shown above, you can score decompositions with the function:

```
const std::vector<std::pair<long double, long double>>& get_scores(
                          std::vector<std::unique_ptr<EdgeSetPair>> *tasks);
```

The input `tasks` contains a list of all the different noise sets to be scored.

The output is a vector of pairs of long doubles. The score for task i is the *first* of the two long doubles in the i'th output pair.

IMPORTANT: The size of the output vector can be larger than the size of the input vector. Never make use of the output vector's size. Always use `tasks->size()`.

An `EdgeSetPair` is a base class. You will likely want to use the `BasicEdgeSetPair` subclass, which is basically just a wrapper around two sets of edges: one set for noise edges removed from the base graph (i.e. noise edges added to the schema), the other for noise edges added to the base graph (i.e. noise edges deleted from the schema).

A `BasicEdgeSetPair` has the constructor:

```
BasicEdgeSetPair(const std::unordered_set<Edge, EdgeHash>& removed,
                 const std::unordered_set<Edge, EdgeHash>& added)
```

5.4.1 Esoteric Trivia

The reason for the class/sub-class functionality is that the genetic algorithm `SCHENO_ga` encodes edges more space-efficiently as integers. That way the `tasks` vector is smaller. When the thread-pool scorer asks for the two edge sets within an edge set pair, the genetic algorithm's `EdgeSetPair` subclass, `GeneEdgeSetPair`, temporarily converts the integers to sets of `Edge` objects.