MG-GA

This repository contains an implementation of the MinGen brute force table anonymizer algorithm, with heavy optimization, alongside a specialized Genetic Algorithm for achieving the same goal. Their purpose is as a learning aid to better understand the ideas of Brute-Force and Genetic algorithms, and how they can be applied to Information Security.

Outline

There are three ways to approach this repository:

1. Delve into the codebase to get a real implementation of these algorithms in C++. All the code is located in the `src` folder, with `mg.h` containing the MinGen implementation, and `ga.h` containing the Genetic Algorithm. You can also look into `domains.h` for the table domain implementation, `metrics.h` for scoring functions and k-anonymity, `table.h` for the CSV class, and `shared.h` for various utility functions. These functions also contain significant optimizations, and can be used to learn about ideas like Pruning and Caching.
2. Test the `main` application that compiles these source files into a working, command line application that can anonymize input tables.
3. Explore the Doxygen HTML rendered documentation of the codebase for detailed information about the algorithms and their functions, but without all the dense C++ to parse.

Codebase Walkthrough

This repository is written in C++. However, even if you aren't very familiar with the language, the codebase has been thoroughly documented, and you can understand the flow of logic using the comment and documentations—even if the particulars of the language's syntax may seem dense. If you are strictly interested in the algorithms in question, you'll want to direct your attention to either `mg.h` for the MinGen implementation, or `ga.h` for the Genetic Algorithm implementation; both Brute-Force and Genetic are rather general concepts, to which these two implementations have been tailored for the task of anonymizing tables. Particularly optimizations, such as caching and pruning, are particularly focused upon for the MinGen implementation as an illustration for how such algorithms can be optimized into acceptable performance.

MinGen

A Brute-Force algorithm, such as MinGen, operates by exhaustively iterating through every possible solution to a problem. MinGen finds every possible way a table could be anonymized, determines the score of that table, and returns the best one. The main advantage of such an algorithm is that the returned solution is the *best* solution; MinGen has scoured every possible table, and can say with certainty that its solution is the best one based on the predefined conditions and constraints. Another benefit to these sorts of algorithms is that they are relatively easy to implement. Both a recursive approach and iterative approach can work particularly well, and can be implemented rather easily by having a function repeated call itself with an end condition, exhausting every possibility. However, the chief disadvantage of such an algorithm is that of speed: exhausting every possible solution only works when a search space of the problem is small (At least, small for a computer).

To illustrate this issue, consider the most simple anonymizing algorithm, where a cell can either be left as is, or suppressed entirely. This presents a search space of 2^n for n cells in the table. Even a relatively small table of six columns and ten rows borders on the largest value that a 64-bit computer is capable of *storing*, much less iterating over. Trying to exhaustively search such a massive space is equivalent to trying to brute-force AES encryption by simply guessing every possible key.

Fortunately, we need not check every solution if we know that a particular branch in the search space will yield to results that won't be good enough. This brings in the key idea of *pruning*, where we use information gathered through iteration, alongside the metrics we are seeking to optimize, to skip checking certain branches in the search tree. Our MinGen implementation checks every table by checking every possible value for each cell, going column by column. We use two values to prune the search space:

1. Our scoring metrics, Minimal Distortion and Certainty, only *add* points for scoring; there is no means of decreasing a score. So, if the score for the first x columns in a table is *already* greater than what our current best table is scored at, then we can guarantee that *any* further permutations of this table will yield to a worse score than what we have, and thus we can stop iteration immediately. If we consider our above example again, and make a prune after the first column, we remove 2^{50} states in a single operation.
2. K-Anonymity can also be used for pruning. If a particular row can be uniquely identified using only a subset of its columns, then it is guaranteed that it can also be uniquely identified no matter the subsequent modifications. If, with $k = 2$, we can uniquely identify each row by a *Name* column, and we did not suppress a value, then the K-Anonymity score of the column will be 1, and no further modifications will be able to change that score. Therefore, if we didn't not achieve the desired K-Anonymity at the end of the column, then we can safely prune every other combination after that as well, with similar gains to the first method.

> **Note**
>
> These pruning checks are used against the finished column, *and all previous* columns, as while a single column may be K-Anonymous, a combination of columns may cause that to change.

Another profound optimization, which applies to both MinGen and the Genetic Algorithm, is through caching. To determine whether a particular table is better than what we've run into thus far, we need to check both the *Score* and *K-Anonymity*. With our above pruning scheme, we need to further check each *column* of the table, for every permutation of the table. Such a setup means that the functions responsible for these values will be called *often*, and benefits from even minor optimizations such as function inlining. A property of this problem is that a lot of these calculations will be redundant: there are going to be a lot of columns with the exact same value, and when we check previous columns, those values will be entirely unchanged. Such a situation is what caching is best suited for, where we can store the result of a computationally expensive operation in a lookup table, and when we run into the same inputs can simply pull from that cache rather than performing the computation.

> **Info**
>
> *Function Inlining* is a compiler optimization in which rather than creating a subroutine within the generated machine code, requiring the program to save the state of registers and *jump* to that location, we instead emplace the content of the function *verbatim* every time it is called. This can significantly improve program speed when functions are called repeatedly, such as with our scoring functions, but will also increase the size of the resulting executable given the duplicate code added to each function invocations. Compilers like GCC and Clang are smart enough to automatically inline functions, but we can explicitly provide the `inline` keyword to enforce* it.

To effectively store this caching data, especially since there will be a lot of data to store, we use a custom built tree, where each node in the tree corresponds to a column in a row. Each node has a value, which means that we can store a value for the first column, the first two columns, and so in, which works perfectly with out per-column scoring for pruning. If we wanted to store the rows `[1,2,3,4]`, `[1,5,6,7]`, and `[1,8,9,0]` into such a tree, it would look like:

```
   / 2 - 3 - 4
1 -  5 - 6 - 7
   \ 8 - 9 - 0
```

Importantly, each of these nodes can have a value, so we could store the certainty score for `[1]`, `[1,2]`, `[1,2,3]`, and `[1,2,3,4]` all within the same branch of the tree; while a hash map theoretically boasts a $O(1)$ lookup time, that does not account for collisions in the hash (Which will happen when storing this much data), alongside the space complexity of having to store all that data in separate entries. In contrast, we are guaranteed to find our match bounded by the amount of *columns* in the table, rather than the amount of values in the tree.

> **Note**
>
> See `shared.h` for more details on the `shared:Tree` implementation.

To put the effects of branch pruning into perspective, `examples/test.csv` prunes the search space by a factor of 3657, reducing a total search space of $2,902,376,448$ distinct tables down to only $793,647$. This lead to an "effective" speed (The total amount of possible states divided by the time it took) of resolving a state in less than *half a nanosecond* for a PGO build on a laptop.

To put the effects of caching into perspective, we can compare a run of `examples/test.csv` using the `--no-cache` argument to disable the cache. On the same setup above, enabling the cache decreases the run time from 2734.42 milliseconds to 1305.53 milliseconds, a speedup by a factor of two and with a 99.98% hit rate on the K-Anonymity cache, and a 99.993% hit rate on the Score Cache, which are staggeringly high rates for a cache.

> **Info**
>
> The hit rate of a cache is defined by the amount of times that the cache was *hit*, or a pre-computed result was already in the cache and could be returned immediately, divided by the amount of *misses*, or the amount of times a result was not in the cache, and had to be computed and added, and the *hits*.

GeneticAlgorithm

The above section outlined numerous ways of trying to improve the speed of brute-force, but even then optimizations can only take it so far. There is a clear need, especially for larger tables, to change the strategy, but all solutions come with trade offs. In a Genetic Algorithm, we trade the exhaustive guarantee of finding the *best* solution with a significant improvement to program execution. However, unlike a blind stochastic algorithm (Such as the MinGen implementation with a fixed `--iterations` argument), we "guide" the random generation through the use of a fitness value.

A genetic algorithm tries to mimic natural evolution, using the principal of "survival of the fittest" to create successive generations that gradually improve. There are four core steps of a genetic algorithm:

1. Create a initial population, typically randomly
2. Score the generation by a fitness metric.
3. Take the best of that generation, and combine them together to produce a new generation.
4. Repeat steps 2 and 3 until a predefined end state is reached, such as a limit on generations, or when a certain fitness is reached.

Like MinGen, there is significant room for interpretation and customization in this scheme, predicated on the peculiarities of the specific problem we wish to solve. Let us go through each step for our table anonymizer implementation:

1. Our initial population is sourced through random permutations of the original table, including domain hierarchies, numerical ranges, absolute suppression, and leaving the values as-is. A random initial population is fast, but requires a robust fitness score to guide that randomness into coherent tables that achieve our goals. The population defaults to a size of 100 members, but it can be configured by the user.

   > **Warning**
   > Increasing the population size significantly slows generation, as it not only increases the amount of tables to generate and score, but the amount of children that must be created for the successive generations. Likewise, decreasing the size significantly decreases the genetic diversity of the generation, leading to more frequent local maximums that cannot be resolved easily.

2. Our fitness metric works in two stages: the first stage seeks to reach the desired K-Anonymity, and the second seeks to minimize our scoring metric. This metric took (quite fittingly) numerous iterations to perfect, to which a more thorough remark can be found in the source code. However, the current iteration of the score can be expressed thusly:

$$F(T_i, k) = \begin{cases} anonymity(T_i) & \text{if } anonymity(T_i) < k \\ (k \times c)/score(T_i) & \text{if } anonymity(T_i) \geq k \end{cases}$$

For a given table permutation T_i and desired k score, where $anonymity$ returns the average K-Anonymity for each row of T_i and $score$ returns either the Minimal-Distortion or Certainty Score depending on user selection, and c is an arbitrary boost to the score to favor K-Anonymity, which currently uses the amount of cells in the table (Otherwise, the case for when K-Anonymity is reached would actually lower fitness). Unlike the $score$ function, which we want to *minimize*, fitness should be *maximized*, with higher fitness being better tables.

3. We can use the fact that our tables contain a set of discrete cells in a similar way to genetic recombination in biology. We take the top 10% of each generation, and randomly pair them off with each other (Almost certainly more than once as we want to maintain the same population size, not because we necessarily need it for the algorithm, but because it makes our randomness functions easier). Each cell is randomly assigned as the value from either of the two parents, such as that it averages to 50% of the cells coming from one parent, and 50% of the other. At least, it would be a 50/50 split if we didn't incorporate a key aspect of genetic algorithms: mutations. A fixed mutation rate (Which can be configured by the user) is added to the random number generator, and if a mutation rate is hit, the value of the cell is randomized from any of the possible values given the original table, numerical ranges, domain hierarchies, etc (Just as the initial population is formed). This allows for children to express traits not present in the previous generation. To further improve this idea, we also *increase* the mutation rate over time, such that what starts as strictly recombination devolves into a stochastic algorithm and lets us break through local maxima; while this idea isn't required for a genetic algorithm (And some implementations would suffer from it), increasing the mutation rate drastically improves our output.

   > **Info**
   > The default mutation rate begins at 10, which leads to a $10/110$ change of mutations for the first generation. Every ten percent of the way through the generation cap, we double this value, meaning that the last ten percent of the iteration boasts a $2560/2660$ chance for a cell to be mutated, or a four percent change of actually inheriting a cell from the parents.

4. Our algorithm stops after a certain, user defined generation count is reached, which defaults to 1000.

> **Note**
>
> Increasing the generation count also slows the rate at which mutations increase. This may make it seem that changes are not happening with the same frequency that a lower generation count does, as it gives more time for the algorithm at a specific mutation level to find better changes; that said, from tests within the `examples` folder a higher generation count does not significantly improve fitness.

Despite starting from random noise, and with only a fitness score to guide it, the Genetic Algorithm returns the best result for both `table.csv` and `table4.csv` in the `examples` folder (MinGen can't churn through `table2.csv`).

Application Walkthrough

If you want to see the MinGen and Genetic Algorithm implementations in action, the `main` application offers a command line utility to parse tables.

> **Warning**
>
> These implementations are effective for small tables, but you shouldn't try them on even moderately sized CSVs. `table2.csv`, which contains sixteen rows, is out of MinGen's reach, and it takes the Genetic Algorithm a little bit as the cache is populated. You are encouraged to use your own tables for testing, but you may want to temper your expectations

> **Tip**
>
> You are *strongly* encouraged to compile your own version of the application, rather than using the pre-compiled version. There is both an optimized and PGO optimized recipe that can be invoked by `make optimized` and `make pgo` respectively. For reference, an unoptimized version runs in 12749 milliseconds, optimized in 1281, and PGO in 1044, or a speed by a factor of 10 for optimized, and a 20% speedup on top of that for PGO. There are no dependencies outside of the standard library and compiler, so if you have `g++` and `libc`, so you should be able to compile it using `make`. All examples henceforth will use the name `main` as the program name, but if you are using the precompiled version you should change that to `main_pc`. See BUILDING for details

> **Info**
>
> PGO, or Profile Guided Optimization, is a technique available in both GCC and Clang in which a profile build of the application is run, and statistics, chiefly the amount of times a conditional (IE an `if` statement) falls through, which allows it to make informed optimizations on the application, particularly in being able to *unroll* loops (Which allows for multiple iterations of the loop to be vectorized and run in parallel). There are no additional dependencies or setup needed to create a PGO optimized version of `main`, just run `make pgo` in the project directory.

`main` offers a bevy of configurations, which can look daunting, so we'll step through each option:

```
main [--mode=m] [--input=filename] [--sensitivities=q,q,q,...] (--domains=fi
lename) (--delim=delimiter) (--types=s,s,s,...)
```

```
(--weights=1,1,1,1) (--metric=md) (--k=2) (--iterations=-1) (--population=10
0) (--mutation-rate=10) (--single-thread) (--no-cache) (--help)
```

Arguments in square brackets (`[]`) are *mandatory* arguments, while those in curved brackets (`()`), are optional. The value listed in the list shows the default for these arguments if they are not specified.

- `--mode` specifies the algorithm to use, which can either be `mg` for MinGen, or `ga` for Genetic Algorithm
- `--input` specifies the location of the table that you want to anonymize, such as `examples/table.csv`
- `--sensitivities` specifies a comma-delimited list of the classification of each column within the table. Your choices are
  `q` *: Quasi: Will be considered for anonymization. This is the default value.*
  `s` : Sensitive/Secret: The values of interest that will be left as-is. You should have at least one of these.
  * `i` : Ignore: Columns that will be ignored for anonymization, such as those without sensitive information

  > **Note**
  > As with all comma-delimited arguments, the ordering is first-column to last-column, and you need not exhaustively specify the value for each column; more values than there are columns will simply be trimmed to size, and missing columns will be filled with a default value. Check the PGO recipe within the `Makefile` to see how you can omit columns, particularly for `--types`.

- `--domains` specifies an optional domains file, which includes the domain hierarchy for columns and can be embedded into the table to refine possible variations of a cell. Check `examples/domains.txt` for an example, alongside the documentation in `src/domains.h` for an explanation of the formatting.
- `--delim` specifies the character used for separating values in the table. While `CSV` should be delimited by commas, and `TSV` should be delimited by tabs (as you would expect from the names), this isn't typically the case, so `main` will guess the delimited for all input files (It doesn't even check extension). If it makes an incorrect guess, you can manually specify the character.
- `--types` specifies a comma-delimited list of the *types* of each column, to which there are two options:
  `s` *: Strings Only relevant domain hierarchies and absolute suppression are supported.*
  `i` : Integers. Ranges, such as `[X-Y]` will be constructed based on the contents of the column, and used for anonymization. Note the syntax, where `[]` define *inclusive* bounds.

  > **Warning**
  > Integer columns add a substantial amount of combinations to the search space, especially with numerous unique values; `main` will generate a range for each pairing of values! If you can afford the loss in information, you can substantially speed up anonymization by simply instructing `main` to treat the column as a string, to which values will either be kept or suppressed entirely; alternatively, you can specify your own domains for the column to explicitly provide your own ranges, such as `[1-100]` → `[50-100]` → `[90-100]`, but note that `main` will not check the correctness of these relations as they are treated as strings, rather than ranges.

- `--weights` specifies a comma-delimited set of unique weighting that is applied to cells in a particular column for Minimal-Distortion and Certainty Score calculation. You can specify floating point numbers of any precision, to which the default is `1.0`. Because these values are *multiplied* against scores for the column, a higher weight will *increase* the score of making changes to the column, which encourages `main` to preserve the information within the column; likewise, lower values tell `main` that the information in the table is not as important, with `0.0` effectively telling the program that there is no cost to suppressing the values entirely. For example, consider

`examples/test.csv`. Say that we want to preserve the value in the `Job` columns. Compare the output of MinGen between not providing weights (Hence defaulting to 1.0 for each column):

```
./main --mode=mg --input=examples/test.csv -s=q,q,q,q,q,s --domains=examples/domains.txt
--metric=c --types=s,i

| Name |     Age | Sex |   Education |          Job | Salary  |
#################################################################
|    * | [25-30] |   M |           * |            * |  50000  |
|    * |      21 |   F |     Diploma | White Collar |  40000  |
|    * | [25-30] |   M |           * |            * |  90000  |
|    * |      21 |   F |     Diploma | White Collar |  40000  |
```

Against increasing the weight of the Job column (IE increasing the value of its information):

```
./main --mode=mg --input=examples/test.csv -s=q,q,q,q,q,s --domains=examples/domains.txt
--metric=c --types=s,i --weights=1,1,1,1,20

| Name |     Age | Sex |   Education |         Job | Salary  |
#################################################################
|    * | [21-25] |   * |           * |           * |  50000  |
|    * | [21-25] |   * |           * |           * |  40000  |
|    * |       * |   * |      Higher |       Admin |  90000  |
|    * |       * |   * |      Higher |       Admin |  40000  |
```

Further, let's assume that we don't care about age being preserved, and thus assign it a weight of 0.0

```
./main --mode=mg --input=examples/test.csv -s=q,q,q,q,q,s --domains=examples/domains.txt
--metric=c --types=s,i --weights=1,0,1,1,20

| Name | Age | Sex |   Education |         Job | Salary  |
#############################################################
|    * |   * |   * |           * |           * |  50000  |
|    * |   * |   * |           * |           * |  40000  |
|    * |   * |   * |      Higher |       Admin |  90000  |
|    * |   * |   * |      Higher |       Admin |  40000  |

| Name |     Age | Sex |   Education |         Job | Salary  |
#################################################################
|    * |       * |   * |           * |           * |  50000  |
|    * | [21-25] |   * |           * |           * |  40000  |
|    * |       * |   * |      Higher |       Admin |  90000  |
|    * |       * |   * |      Higher |       Admin |  40000  |

| Name |     Age | Sex |   Education |         Job | Salary  |
#################################################################
|    * | [21-25] |   * |           * |           * |  50000  |
|    * |       * |   * |           * |           * |  40000  |
```

```
|    * |       * |   * |      Higher |       Admin |  90000  |
|    * |       * |   * |      Higher |       Admin |  40000  |

| Name |     Age | Sex |   Education |         Job | Salary  |
################################################################
|    * | [21-25] |   * |           * |           * |  50000  |
|    * | [21-25] |   * |           * |           * |  40000  |
|    * |       * |   * |      Higher |       Admin |  90000  |
|    * |       * |   * |      Higher |       Admin |  40000  |
```

Because every modification is valued at 0, we get multiple results (You'll get multiple results for many operations with `main`, this is just a case where, because there is no penalty for suppression, we get increasing suppression since the algorithm has no qualms just removing *all* the information indiscriminately as much as it does keeping it in).

> **Note**
>
> There is no minimum or maximum weight value (A value less than zero would *incentivize* the program to suppress the values), which means you may need to tweak the weights to get desired results, especially increasing/decreasing the weights to have your preference reflected in the output; note that while no weight will break the desired K-Anonymity for MinGen, because K-Anonymity is not necessarily *enforced* in the Genetic Algorithm, just strongly encouraged, setting excessively high/low weights can cause the Genetic Algorithm to return non K-Anonymous tables!

- `--metric` specifies the score metric used for determining the value of a table. You can either select `md` for Minimal Distortion, which merely adds a point for *any* modification, regardless of how much information is quantitatively is preserved, or `c` for Certainty, which takes into consideration domains and ranges to give a more nuanced score reflecting the degree to which information is suppressed, and encouraging changes that keep as much information as possible. It defaults to Minimal Distortion.
- `--k` specifies the k value for K-Anonymity, which requires a given row be anonymized such that at *least* K rows could match it. Higher K values require more suppression, and a $k = 1$ would result in the table being returned verbatim.
- `--iterations` has a two-fold purpose depending on the `--mode` switch:
  When `--mode=mg`, it defines a non-exhaustive, stochastic search that will attempt the defined amount of iterations before returning the best found result. `-1` asserts an exhaustive search, as it its default behavior.
  When `--mode=ga`, it defines the amount of generations that should be used in the algorithm, to which the default is 1000.

  > **Info**
  > Our iteration count is stored as an *unsigned* number, which means that rather than using the last bit of the number to hold the sign of the number (Positive or Negative), it is used to double the amount of numbers that can be stored, at the cost of not being able to hold negative numbers. Due to the way that negative numbers are stored (Two's Compliment), a *signed* value of -1 equates to a bit string of all 1 which is the largest value an unsigned number can store!

- `--population` defines the size of each generation when `--mode=ga` (This does nothing when using MinGen). A higher population size increases the genetic diversity of each sample (As it also increases the size of the pool of parents to which the generation is sourced), but also significantly increases runtime. It defaults to 100.

- `--mutation-rate` specifies the initial rate of mutation. The formula for mutation is $\frac{m}{m+100}$, so the default value of 10 starts with a 9% chance of mutation for any cell. This value doubles every 10% of the max generation (From `--iterations`), which devolves the Genetic Algorithm into a largely stochastic one at the end to try and break through any local maxima that may have been reached.

> **Tip**
>
> Mutations are a key aspect of genetic algorithms. Try disabling mutations altogether with `--mutation-rate=0`, and watch as the algorithms fails to even achieve K-Anonymity!

- `--help` prints out a wall of text that contains more information about these switches.
- `--no-cache` disables the metric caching, which can allow you to quantitatively see the effects of caching upon runtime speed.
- `--single-thread` for if you're a compiler creating a profile.

If you need some help stringing together a command, here are some commands you can run against the example files:

```
./main --mode=mg --input=examples/test.csv -s=q,q,q,q,q,s --domains=examples/domains.txt --metric=c --types=s,i
```

- Run MinGen against `test.csv`, with all columns save the last being quasi-sensitive, using our example domain definitions, with Centrality Score, and with the second column as integers.

```
./main --mode=ga --input=examples/test2.csv -s=q,q,q,q,s --domains=examples/domains.txt --metric=c --types=s,i --iterations=100
```

- Run the Genetic Algorithm against `test2.csv`, with the last column as the sensitive column (Note that the columns are different from `test.csv`, hence the change), using our domains, with centrality score, and with Age as integers. We also decrease the iteration count to speed up generation.

> **Tip**
>
> The pre-compiled version of `main` may take a while to parse through `test2.csv`. You can significantly speed it up by compiling it yourself with either the `optimized` or `pgo` recipe.