High Performance Genetic Algorithms for Steganalysis

Joseph Charles Bingham

Department of Mathematics

Iowa State University of Science and Technology

Ames, Iowa 50010, United States of America

jbingham@iastate.edu

April 27, 2018

Abstract

This research outlines a novel implementation of a genetic algorithm that leverages high performance parallelizations to detect steganaraphically embedded images. The two main components that are new to this project are the application of parallelization for genetic algorithms and the application of genetic algorithm for steganalysis for LSB embedding. Typical steganalytic methods which use machine learning techniques require an unreasonable amount of pre-classified data and copious amounts of time for training the engine. The data needed for such operation usually must be lab generated, which can lead to the biases when compared to real world, and often is constrained to specific parameters, such as they must remain within either the spacial domain or the JPEG domain, must be the same pixel width, etc., making these engines limited to what image space they detect over. The need for all data to be used in training the engine, as well as the linear nature of the engines used precludes them from being parallelized in any meaningful fashion.
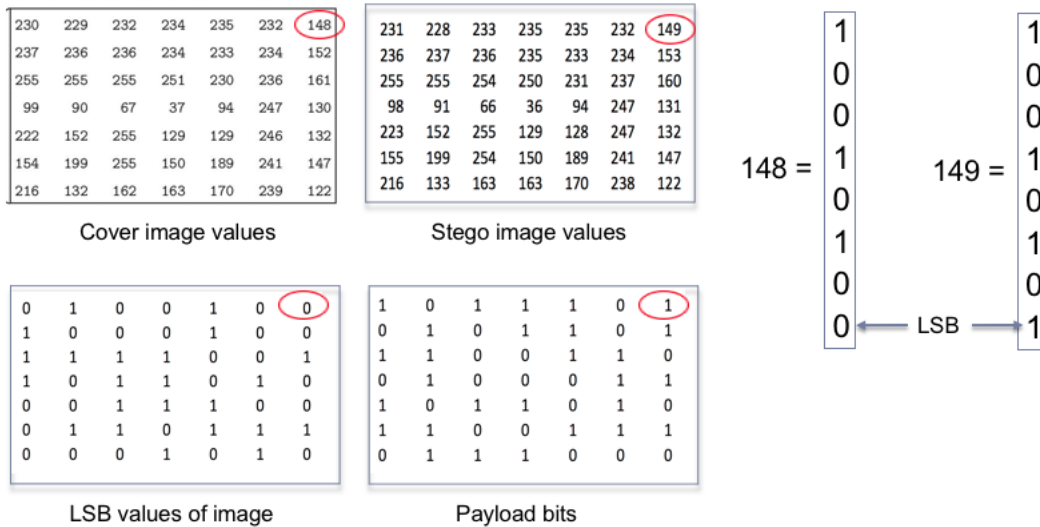
This new algorithm does not succumb to these shortfalls. By using solution sets of pixels as the data which the genetic engine trains over, and fixing the image(s) under suspicion, the engine can be parallelized, vastly increasing the efficiency. Since the algorithm searches for sets

solutions and then uses a fitness function to determine whether the findings were significant using a known statistical test, it does not require training data at all, mean that there are no biases introduced by lab generated data. The algorithms flexibility does not limit it to just one format of images, meaning that the same program can be used without the need to generate a new set of data or train a new engine.

1 Background Information

Steganography and Steganalysis Steganography is the practice of embedding information into photographs or audio media in such a way that it is not detectable to the average person. The form of steganography that the scope of this project is most interested in is LSB (Least Significant Bit) embedding of images using random paths [1].

This, as the name suggests, is where the path of the embedding stream is random in nature. When a pixel from the image is selected to be altered, the least significant bit is overridden to be the same as the next bit in the embedding stream. Since it is just the least significant bit, very little visual alterations occur. This makes detection difficult, if not impossible without the aid of software. (Example below)

230	229	232	234	235	232	148
237	236	236	234	233	234	152
255	255	255	251	230	236	161
99	90	67	37	94	247	130
222	152	255	129	129	246	132
154	199	255	150	189	241	147
216	132	162	163	170	239	122

Cover image values

231	228	233	235	235	232	149
236	237	236	235	233	234	153
255	255	254	250	231	237	160
98	91	66	36	94	247	131
223	152	255	129	128	247	132
155	199	254	150	189	241	147
216	133	163	163	170	238	122

Stego image values

0	1	0	0	1	0	0
1	0	0	0	1	0	0
1	1	1	1	0	0	1
1	0	1	1	0	1	0
0	0	1	1	1	0	0
0	1	1	0	1	1	1
0	0	0	1	0	1	0

LSB values of image

1	0	1	1	1	0	1
0	1	0	1	1	0	1
1	1	0	0	1	1	0
0	1	0	0	0	1	1
1	0	1	1	0	1	0
1	1	0	0	1	1	1
0	1	1	1	0	0	0

Payload bits

$$148 = \begin{matrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{matrix} \qquad 149 = \begin{matrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{matrix}$$

LSB

A random embedding path in steganography implies that the pixels whose bits are overwritten in the above fashion are chosen at random.

Steganalysis is the discovery of the existence of hidden information; therefore, like cryptography and cryptanalysis, the goal of steganalysis is to discover hidden information and to break the security of its carriers [2].

Genetic Algorithms   Genetic Algorithms are algorithms that follow a specific machine learning paradigm [3]. They are classified by their similarity to natural selection as found in evolution in the wild. They consist of three main parts: a cost function, a tester bot, and a builder bot.

The cost function determines the viability of each solution. Typically this function determines how well the solution completes the task that is trying to be accomplished, and is usually from $\mathbb{R}^n \mapsto [0, 1]$.

The tester bot tests each solution based on the cost function. Its job is to determine what the value of the test function and to rank each solution.

The builder bot takes each the best solutions, as ranked by the tester bot, generates more solutions based on their attributes. It generates the new solutions by randomly modifying the best solutions to descend along the gradient of the solution space to an optimal (minimal cost) solution.

2   Implementation :: Theory

Assumptions   The assumptions that are made for this research are :

1. The bit stream that is generated by the message has an equal amount of 0's as it does 1's. This is reasonable as the message is likely encrypted using some hash function, which would likely convert the message to a binary string with a uniform distribution of 0's and 1's.

2. The path of embedding is uniform random, meaning that each pixel is equally likely to have received a bit from the bit stream. This implies that each groups of pixels of equal size will likely contain an equal number of overwritten bits. This is a reasonable to assume as there already exist many techniques for finding embedding paths which are not random.

3. The message overwrites 50% of the image. For theoretical purposes, this makes the computation easier, but for the practical aspects, this embedding rate is enough to detect with most methods, and would provide a good benchmark for the algorithm.

Genetic Algorithm for Steganalysis   One of the novel components to this research is the use of genetic algorithm for steganalysis. The thought behind this is to generate a classifier for steganographically embedded images versus normal cover images.

This is done by creating a evolutionary solution for a path finding algorithm. The path finding algorithm optimizes the possible paths of embedding to determine the probability of payload being embedded. If the genetic algorithm's internal weight function is triggered, then the probability along the path found was deemed high enough to consider the image having a payload embedded.

The weight function is the χ^2-attack from steganalysis [4]. The basis of this attack is that grey scale values that differ by only the last bit (the bit that is overwritten) would show up equally in the embedding path. This is because an LSB within the embedding path has an equal likelihood of being 0 or 1. For example:

An image, I, whose pixels are represented by 3 randomly generated bits in the following matrix :

$$\begin{bmatrix} 101 & 011 & 111 & 100 \\ 001 & 001 & 100 & 001 \\ 101 & 110 & 010 & 101 \\ 101 & 110 & 010 & 101 \end{bmatrix}$$
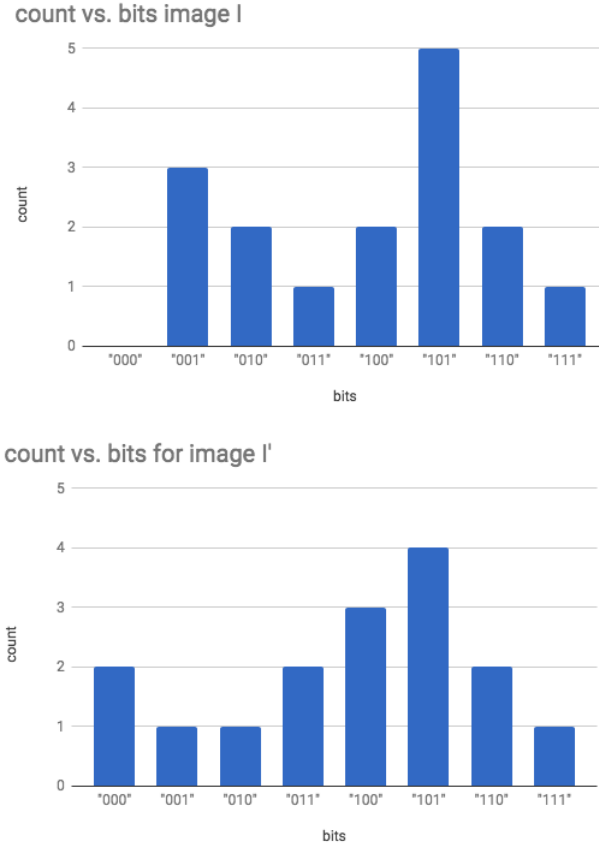
with a lexicographical embedding of the message stream

$$\begin{bmatrix} 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0 \end{bmatrix}$$

becomes I'

$$\begin{bmatrix} 101 & 010 & 111 & 100 \\ 001 & 000 & 101 & 000 \\ 101 & 110 & 011 & 100 \\ 101 & 110 & 011 & 100 \end{bmatrix}$$

This gives the histograms,

count vs. bits image I

count vs. bits for image I'

As one can see, the values before the embedding, I, are far less pairwise normalized than those in I'. By feeding these pairwise differences into the cumulative distribution function for the χ^2 distribution, with the degree of freedom equal to the number of bins in the sample set, we get a value that is between 0 and 1, and is directly correlated to how likely it is that the pixels are stegonagraphically embedded. This provides us with a function $F : \mathbb{Z}_{2^k}^{n^2} \mapsto [0, 1]$, where k is the number of bits to represent the grey scale value and $n \times n$ is the square pixel array size of the image in pixels. For the above example, k = 3 and n = 4. This provides a natural candidate for use as a fitness function.

The tester bot goes through each solution of pixels and evaluates them based on this fitness function, and ranks them on this value. A key difference between this genetic algorithm and other genetic algorithms is that the typical actor of the data over which the genetic algorithm would be the images that are under suspicion. However, in this algorithm, the data is the solution set and the image is fixed. This allows for a much more compact implementation with regards to memory as each solution would only need to keep track of the pixels it is referencing, and not all of the images.

For testing the functions, the bot not only tests the overall fitness of the total solution, but it also chunks the solution into four, with the first fourth being the pixels of the solution which lie within the first quarter of the image lexicographically, the second fourth being from the second fourth of the image and so on. The bot then evaluates the fitness of each chunk. (Sample below).

$$
\begin{bmatrix}
101 & 010 & 111 & 100 \\
001 & 000 & 101 & 000 \\
101 & 110 & 011 & 100 \\
101 & 110 & 011 & 100
\end{bmatrix}
\mapsto
\begin{bmatrix}
\begin{array}{cccc}
101 & 010 & 111 & 100 \\
\hline
001 & 000 & 101 & 000 \\
\hline
101 & 110 & 011 & 100 \\
\hline
101 & 110 & 011 & 100
\end{array}
\end{bmatrix}
\mapsto
\begin{bmatrix}
\begin{array}{c}
\text{fitness val 0} \\
\hline
\text{fitness val 1} \\
\hline
\text{fitness val 2} \\
\hline
\text{fitness val 3}
\end{array}
\end{bmatrix}
$$

The philosophy of the builder bot for this implementation is to combine two solutions based on their fitness scores in such a way that gives priority to pixels that appear in both, then pixels from the solution with the higher fitness, then random pixels from the same section of the image (to add non-determinism into the solutions). This is done by taking the intersection of the two sets, add these to the new solution. Then adding the amount determined by the ratio of the two fitness values from remainder of the pixels from each solution. Then randomly changing up to one pixel. This will have the solutions converge to the highest fitness value, which is used to determine whether the image has been embedded.

Parallel Implementation of Genetic Algorithms   The other novel component is the parallelization of genetic algorithms to speed up the training time of the evolutionary model. The reason this project requires a parallel training model, as opposed to a serial implementation is, because the quantity of data (the solution sets) that needs to be ingested by the engine is immense, $O(n^8)$, where n is the side length of the image in pixels.

The parallelization is achieved by using a "island" inspired model. This is where each node of the cluster getting its own evolutionary model, but with

the same cost function. Then after some number of iterations, the nodes give each other their top solutions to reproduce with the other solutions. This speeds up, and aids with, the convex optimization.

3   Implementation :: Code

This section includes some highlights of the code implementation, along with the mathematical reasoning and proofs of their complexities.

```
/*
gamm:
param: double x, input to function
return: double, the gamma function applied
implemented using an approximation.
This was an approximation using a rational approximation found by SCHMELZER
and TREFETHEN
The memory complexity of this is O(1) and time complexity of O(1)
*/
double gamm(double x)
{
    double ret = (1.000000000190015 +
                  76.18009172947146  / (x + 1) +
                  -86.50532032941677 / (x + 2) +
                  24.01409824083091  / (x + 3) +
                  -1.231739572450155 / (x + 4) +
                  1.208650973866179e-3/ (x + 5) +
                  -5.395239384953e-5 / (x + 6));

    return ret * sqrt(2.0*PI)/x * pow(x + 5.5, x+.5) * exp(-x-5.5);
}
```

By an easy inspection, the memory complexity of this is O(1) and time complexity of O(1), as there is no dynamically allocated memory, nor are there any loops [5].

```
Solution *mixSolutions(Solution *other)
{
 Solution *ret = new Solution;
 for(int i = 0; i < 4; i++)
 {
  if(this->fitVal[i] >= other->fitVal[i])
  {
   int k = 0;
   int T[SOL_SIZE * 2] = {0};
```

```
for(int j = 0; j < SOL_SIZE; j++)
{
 T[this->values[i][j]]++;
 T[other->values[i][j]]++;
}
for(int j = 0; j < SOL_SIZE*2; j++)
{
  if(T[j] == 2)
  {
   ret->values[i][k] = T[j];
   k++;
  }
}
int upperLim = (SOL_SIZE - k)*(this->fitVal[i]/other->fitVal[i]);
while(k < upperLim)
{
 for(int l = 0; l < SOL_SIZE; l++)
 {
   if(!in(this->values[i][l], ret->values[i])
   {
    ret->values[i][k] = this->values[i][l];
    k++;
    break;
   }
 }
}
 while(k < SOL_SIZE)
 {
  for(int l = 0; l < SOL_SIZE; l++)
  {
   if(!in(other->values[i][l], ret->values[i])
   {
    ret->values[i][k] = other->values[i][l];
    k++;
    break;
   }
  }
 }
}
else
{
 int k = 0;
 int T[SOL_SIZE * 2] = {0};
 for(int j = 0; j < SOL_SIZE; j++)
 {
```

```cpp
      T[this->values[i][j]]++;
      T[other->values[i][j]]++;
    }
    for(int j = 0; j < SOL_SIZE*2; j++)
    {
      if(T[j] == 2)
      {
        ret->values[i][k] = T[j];
        k++;
      }
    }
    int upperLim = (SOL_SIZE - k)*(this->fitVal[i]/other->fitVal[i]);
    while(k < upperLim)
    {
      for(int l = 0; l < SOL_SIZE; l++)
      {
        if(!in(this->values[i][l], ret->values[i])
        {
          ret->values[i][k] = this->values[i][l];
          k++;
          break;
        }
      }
    }
    while(k < SOL_SIZE)
    {
      for(int l = 0; l < SOL_SIZE; l++)
      {
        if(!in(other->values[i][l], ret->values[i])
        {
          ret->values[i][k] = other->values[i][l];
          k++;
          break;
        }
      }
    }
  }
  int a = rand();
  if((a %= 400) < 4 && !in(a, ret->values[i])
  {
    ret->values[i][a] = ret->values[i][rand() % (SOL_SIZE)];
  }
 }
 return ret;
}
```

Looking at the call to in() , which is implemented in a O(n) fashion (where n is the number of pixels), it is within a for loop within a while loop within a for loop, giving this step $O(n^4)$. Although this is much larger than the typical highest complexity of $O(n^2)$ for industry applications, this is much better than the $O(2^n)$ of the CNN used by most steganalytic applications.

4 Results

In comparison to brute force searching, which would have a work case of $\binom{n^2}{\frac{n^2}{2}}$ where n is the side length in pixels, which would be looping though every possible combination of pixels, this algorithm, assuming that only half of the solution pixels need to be found, would only need

5 Next Steps

Relieving the 50% embedding requirement   One thing that I wish to experiment with is adding

6 Bibliography

[1] Yu, L., Zhao, Y., Ni, R. et al. EURASIP J. Adv. Signal Process. (2010) 2010: 876946. https://doi.org/10.1155/2010/876946

[2] Deborah Radcliff "Computer World" http://www.computerworld.com/securitytopics/security/s

[3] Whitley, D. Stat Comput (1994) 4: 65. https://doi.org/10.1007/BF00175354

[4] Orion.math.iastate.edu. (2018). Pairs of Values and the Chi-squared Attack. https://orion.math.iastate.edu/dept/thesisarchive/MSCC/CStanleyMSSS05.pdf.

[5] Schemelzer and Trefethen, SIAM J. NUMER. ANAL, Computing the Gamma Function Using Contour Integrals and Rational Approximations (2007) https://doi.org/10.1137/050646342