Data Compression via Genetic Algorithm

29 November 2019

Madeline V. Hundley

University of Louisville

Table of Contents

Introduction

This project presents a Genetic Algorithm (GA) for data compression. The program evolves a customized compressed representation for specific data as opposed to a general algorithm which is applied to any data submitted. It is a dictionary technique which maps variable-length input symbols to shorter output codewords. The GA's goal is to evolve an optimized dictionary for the particular input. The input to the algorithm is a large integer which can represent any other type of data. The output is a package of bytes with the algorithm's custom solution and the compressed data. This algorithm represents a novel approach to compressing such input data.

Genetic algorithms mimic the real-world phenomena whereby biological populations reproduce and the genetics of each generation are passed down from parents to their children. Natural selection steers each generation towards superior genes. The genetic algorithm discussed here initializes a random population of code dictionaries then reproduces successive generations of solutions.

The input was both randomly-generated incompressible large integers and patterned compressible large behaviors, in order to observe the behavior of the GA and compare it to other compression methods. Large integers are good representations of any type of input data, making them a good case study for application in a wide variety of real-world compression scenarios.

Each member of the genetic population is represented as a dictionary mapping symbols to codes. Each symbol is some subset of the input data and each code is the binary to which its symbol is mapped for compression. Therefore a gene of a population member is one member of this dictionary: one symbol and one code. Below is a trivial example of such a code dictionary.

Symbol	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Codeword	000	001	0100	0101	011	100	101	1100	1101	111

Each member of the population is analyzed for its reproductive fitness and the algorithm probabilistically selects strong parents to reproduce children. The fitness test is the compression ratio achieved. In each reproduction there is a chance mutation will occur. The next generation's population is formed of the strongest members from the old population and from the newly-bred children, while weaker members are discarded. Reproduction occurs a high percentage of the time but there is also a chance for new randomly-generated code dictionaries to be added to the population, increasing the genetic diversity of the gene pool.

The motivation behind creating this algorithm was to explore the ability of a GA to compress data more effectively than other compression algorithms because the solution is unique to each input.

Dictionary Technique

Dictionary techniques for compression replace symbols from the input with codewords in the output; compression is achieved when the encoded output plus the dictionary form a smaller package than the original input. In order to guarantee the compressed output can be accurately decompressed, in all dictionary techniques both the symbols and codes must possess the prefix property. This means no symbol or code may be the prefix to another i.e. it cannot be the beginning of a different symbol or code. For example, including both the symbols '4' and '46' or both the codes '000' and '0000' in the dictionary violates the prefix property. Following this rule guarantees the output is uniquely decodable because it removes all ambiguity for how the codes should be mapped. When '0000' is found in the output, it would match the initial '000' to a symbol if that were an option so it must only be '0000' which can map to a symbol. The reverse is true of the encoding process.

Related Research

Genetic Algorithms have been applied to data compression in a variety of ways, though none specifically with code dictionaries as the genetics applied to an input integer. Following is an overview of related research.

Genetic algorithms, also called Genetic Programming (GP), have been applied in lossless compression of images, microcode, floating-point numbers, and text. In [1], GP was applied to images and used to evolve nonlinear predictors for the gray level a pixel will take based on the gray values of a subset of its neighbors. The prediction errors together with the model's description represented a compressed version of the original image. These were further compressed using Huffman encoding. Intel patented a method in [2] to compress microcode. The method utilizes a GA to find common patterns in the microcode's bit strings and stores them in a table with a unique ID for each pattern. The authors of [3] used a GA to evolve a dictionary of syllables for Huffman coding of text, to improve the Huffman coding performance. The authors of [4] extended this approach with Czech and English text. [5] discusses an application of GP to find an optimal dictionary for lossless compression of text. In [6], the researchers present a hybrid compression system which depends on GP to find the best Huffman tree for input text and then applies the Oring bits compression method on the results.

A number of explorations have been made applying Genetic Algorithms to optimize existing lossless compression. In addition to [3], [4], and [6], the authors of [7] also applied GP to improve the standard Huffman coding of text and the authors of [8] proposed an online text compression system based on a GA which finds the most repetitive n-grams within text. Researchers have also applied GP to tune the parameters of other forms of compression, as with Floating-Poing Compression (FPC) in [9], fuzzy image compression in [10], and conventional wavelet compression of images in [11]. The proposal in [12] evolved preprocessors to reduce the entropy of the original image for lossless compression using GP and the proposal in [13] evolved preprocessing for text compression. A final important application of GP to lossless compression is GP-Zip [14], which uses GP to select the compression method best for each segment of the input data. It defines this "ideal universal compression" as a system that "rapidly identifies different incompatible fragments within the data before applying a specialized compression technique for each fragment."

System Description

Needs Assessment

While data storage is decreasing in price and network speeds are increasing, data compression still plays an important role in today's digital world. Much of our media and digital products today are growing more complex and rich, with the downside they

are larger in size. Government and commercial entities all have a desire for smaller data sizes on-disk and therefore faster transfer speeds across networks. Therefore, the exploration of data compression techniques is valuable to drive discovery of compression that is more effective and/or faster depending on the user's individual priorities.

The goal of this project is to explore the applicability of genetic algorithms to data compression.

Software Overview

The program is written in Python 3.5 with graphs generated by the matplotlib module. For comparison against other compression techniques, the base64 and ujson modules were used to serialize the output for compressing by the zlib, lzma, and bz2 modules. The bitarray module provided support for the binary codes and binary output.

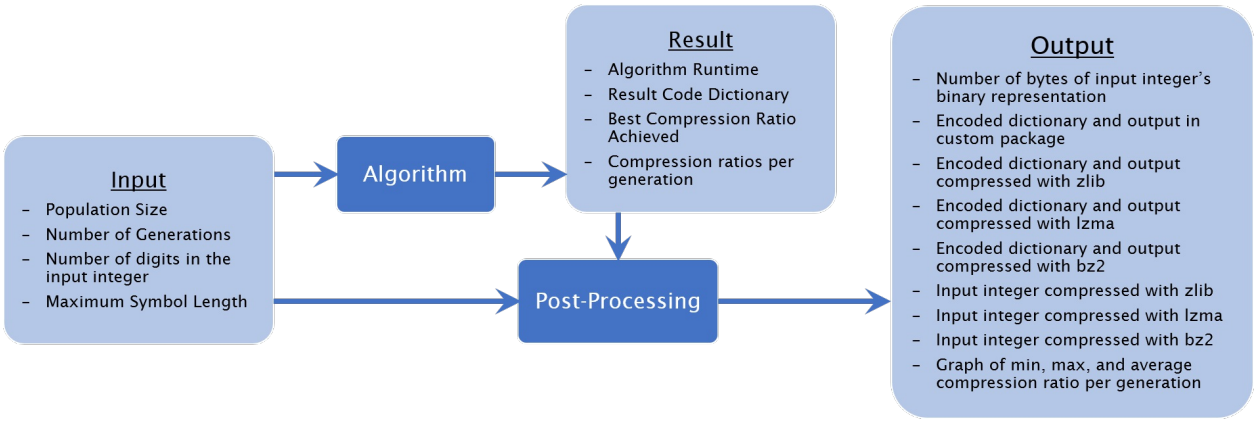
The random and numpy.random modules were key in making symbol selections while forming dictionaries and selecting parents during reproduction. I also made use of the logging, time, operator.attrgetter, math, datetime.datetime, and csv modules for various statistic and administrative functions in the program.

The program could be compiled for a variety of operating system architectures for widespread use.

System Requirements

To run the program as-is, a user would require Python 3.5+ and the modules described above installed on their system.

System Diagram



Detailed Implementation

The program's first order of business is to set or calculate certain required global variables. These variables are:

- `POP_SIZE` - The population size i.e. how many genetic members each generation will have
- `NUM_GENS` - The number of generations
- `DISCARD` - How many members of a population will be replaced each generation; for all tests this was approximately 1/3
- `INDATA_NUM_DIGITS` - The number of digits i.e. the length of the input integer to be compressed
- `INDATA` - The input integer
- `INSTR` - The string representation of this input integer; used for the symbols so `'0'` is represented and treated correctly by each operation in the program
- `MAX_SYM_LEN` - The maximum length allowed for any symbol

The program then calls the main algorithm function which generates an initial random population and reproduces successive generations selecting parents by their fitness to reproduce children. This main function is wrapped in one which measures and

records the runtime, the compression ratios of every generation's population members, and the final result, returning all of these for post-processing and output.

In its post-processing, the program compresses the output with a variety of other common, popular compression techniques in order to compare their performance against the algorithm's custom compression. The other compression algorithms used are Deflate (zlib), BZip2, and LZMA.

The program's final order of business is to plot the best, average, and worst results from each generation of the algorithm and to record relevant statistics for analysis.

CodeDictionary Class

This program is based on a `CodeDictionary` class with the following attributes and functions.

Instance Attributes:

- `symbols` - The symbols selected from the input to be mapped
- `codedict` - A dictionary with the symbols to be mapped as the keys and their binary codes as the values
- `outdata` - The binary encoding of the input according to the code dictionary
- `outbytes` - The custom packaged output composed of a header, the code dictionary, and the binary-encoded output.
- `comp_ratio` - The compression ratio achieved; a ratio of the number of bytes required to represent the input integer in binary to the number of bytes in the `outdata` package.

Class Attributes:

- `pm` - The probability of mutation occurring during reproduction
- `pc` - The probability of crossover occurring i.e. the probably that a child will be reproduced genetically via crossover of genes as opposed to randomly generated

Functions:

- `__init__` - Initializes a CodeDictionary from a dictionary object mapping symbols to binary codes.
- `__repr__` and `__str__` - Internal functions for representation of the CodeDictionary and string output.
- `__eq__` - A custom test for equality. Resolves to `True` if the symbols in two code dictionaries are the same and if they have the same compression ratio. Though the symbols may not map to exactly the same codes, if they have the same compression ratio they are functionally equivalent.
- `encode` - Processes the code dictionary's symbols against the input integer, producing the binary `outdata`, full packaged output `outbytes`, and calculating the `comp_ratio`.
- `mutate` - Called during production of a new child, this function checks whether a mutation occurs and performs it if so; discussed below in more detail in the context of the genetic algorithm.

To initialize a `CodeDictionary` the class must be provided with a dictionary mapping symbols to binary codes. All instance variables are then computed based on this `code`. It is possible to provide an empty, dummy dictionary; these are used when forming a new random `CodeDictionary` and are passed in the place a valid parent would take during reproduction. The binary encoding, binary output, and compression ratio are computed at this point so that information is available when the code dictionaries must be compared for their fitness in the population. `comp_ratio` compares the number of bytes required to represent the original integer `INDATA` in binary compared to the number of bytes of `outbytes`. This is the true compression ratio achieved because it takes into account the need to package the code dictionary with the compressed data; this is discussed in further detail below.

Custom Compression

A custom header, the code dictionary, and the input encoded via the code dictionary are packaged together to form the final output.

First, the function forms the header, composed of these fields:

1. `nsymd_bits` (4 bits) - The number of bits needed to store each symbol's number of digits; to store in a nibble `MAX_SYM_LEN` must not be greater than 15
2. `ncode_bits` (4 bits) - The number of bits needed to store each code's number of "digits" i.e. bits; must not be more than 15
3. `nsyms` (10 bits) - The number of symbols; must not be more than 1023
4. `npad_bits` (3 bits) - The number of trailing zeros; 0-7

In order to preserve possibly leading zeros in the symbols, each symbol cannot simply be stored as its integer equivalent. It must be stored digit-by-digit. Because each digit can only be one of 10 options ('0' - '9'), each digit requires a nibble to encode. Similarly, codes may have leading zeros, so the length of each must also be demarcated. This is why the header includes guidance on the length for each of these fields. It allows decompression to occur correctly.

After the header, each entry of the code dictionary is encoded and added to the binary output. Each entry is represented with the following:

1. `nsymd` (`nsymd_bits` bits) - The number of digits in the symbol, as its integer in binary
2. The symbol (4 bits each) - Each digit as its integer in binary
3. `ncodeb` (`ncode_bits` bits) - The number of bits in the code, as its integer in binary
4. The code - The codeword in its binary

After the code dictionary, the output from the encoding -- the input data after it has been parsed for its symbols and converted to their codes -- is added. Finally, padding is added if necessary so the entire package is divisible by 8, so that it can be stored as bytes.

As an example, if the trivial code dictionary above is applied to the simple input `0123456789`, the output will be calculated as follows. The longest symbol is 1 digit, therefore the largest value any `nsymd` field will need to be is 1 and so `nsymd_bits` need only be 1. (This is not a realistic example.) The longest code is 4 digits, therefore the largest value any `ncodeb` field will need to be is 4. The number of bits required to represent 4 in binary is 3, so `ncode_bits` will need to be 3. The package will be generated as such:

```
Header:   0001 0001 0000001010
Body:     1 0000 011 000
          1 0001 011 001
          1 0010 100 0100
          1 0011 100 0101
          1 0100 011 011
          1 0101 011 100
          1 0110 011 101
          1 0111 100 1100
          1 1000 100 1101
          1 1001 011 111
Output:   000 001 0100 0101 011 100 101 1100 1101 111
Padding:  00
```

The overall output is now bytes, which can be viewed in hexadecimal as `1102A0C4599449C5A37572CEDE6626E5F0515CB9BC`.

Genetic Algorithm

Initial Population

To begin, the genetic algorithm sets up its initial population with the `get_init_pop` function by generating code dictionaries randomly. The `get_rand_codedict` function calls `complete_dict` with a dummy kid and parent. `complete_dict` normally begins with a subset of the necessary symbols to form a dictionary and finds additional symbols necessary to complete it; however, in this case it finds all of the symbols. The `complete_dict` function then calls `complete_codes` with the original kid (the dummy in this case) and the new set of symbols. It calls the `gen_codes` function, which takes as input the number of codes required and recursively generates unique binary codes of the necessary length which obey the prefix property. When returned to

`complete_codes`, the new binary codes are matched to their symbols to complete the code dictionary. Returned to `get_rand_codedict`, this can now be initialized as a `CodeDictionary` and returned to `get_init_pop` to become a member of the initial population.

Successive Populations

Each successive population in the genetic algorithm is generated by the `get_next_pop` function. It calls the `reproduce` function to get children. The `reproduce` function first checks if crossover will occur i.e. if the parents' genes will be crossed with each other's to genetically reproduce. If not, two random code dictionaries are formed as the children. Setting the probability of crossover to less than 1.0 so the populations gets influxes of new, random code dictionaries increases the genetic diversity of the population. It can be thought of as avoiding "in-breeding". Such infusions of new genetics give the algorithm a greater chance of discovering an optimal combination of genes i.e. an optimal solution.

Children that are duplicates of existing members of the population are not retained, so if all parents have reproduced but not enough valid children were "born", `get_next_pop` will use `get_rand_codedict` to get new randomly-generated children. After the number of children required have been created, `get_next_pop` forms the next population from the most fit of the current population and the children, discarding any of the "old and weak" or "runts of the litter".

When genetically reproducing, the `reproduce` function uses the `get_fitness` function to judge the fitness of each potential parent in the current population. Better code dictionaries -- those with higher compression ratios -- are judged to have a higher reproductive fitness and therefore have a greater probability of being selected as parents for the next generation. When two code dictionaries are selected to be parents, the crossover function generates two new children from those parents' "genes." Every time a child is produced there is a possibility that a mutation will occur in its "genes," handled by the `CodeDictionary` class's built-in `mutate` function.

In the `get_fitness` function, the fitness ratios of every code dictionary in the population need to reflect how high a compression ratio they've achieved. Therefore it adjusts each compression ratio to a scale between the maximum of any of these values and the minimum, the scales again so all values add up to 1.0 i.e. they form a valid probability distribution.

The `reproduce` function, after selecting parents based on this probability distribution, sends these two parents to the `crossover` function. Here, half of the "mom's" genes are randomly selected as the starting point for "kid1" and half of "dad's" genes are used as the basis for "kid2". Selecting these starting genes randomly is purposeful: there is no computationally trivial way to compute the "best" genes. How effective a gene is in achieving compression depends on the length of its symbol, the length of its code, and its frequency in the input. Computing these outright is the approach some compression methods take when using a dictionary technique; genetic algorithms instead explore these possibilities iteratively, allowing the overall fitness of the preceding generating influence the genetics of the next, leading to a discovery of these optimal genes. After `kid1` and `kid2` have been setup, they are passed to the `complete_dict` function with `dad` and `mom`, respectively. The `complete_dict` function takes what symbols have been provided by the kid and attempts to complete that dictionary with the symbols from the parent provided. In this way, both kids will be influenced by the genes of both parents.

Completing the Dictionary

At the top of the `complete_dict` function, the input data is represented as a list with one item: the input integer as a string. For example, if the input integer is `0123`, it is represented as this stage as `['0123']`. It uses a string so that '0' is treated correctly by the program. Integer operations break or malfunction if a leading '0' is supplied, but this may very well be the case for some of the symbols chosen to encode the input. The function processes each of the kid's symbols against the input data via calls to `single_encode`. This function which finds every instance of the symbol in the input data and removes it. It also splits the input string at each occurrence of the symbol, ensuring the boundaries of where symbols began and ended are properly demarcated. For example, `single_encode('2',['0123'])` will return `['01','3']`. This demarcation guarantees future symbol selections and processing are accurate. As symbols are selected to be included in the dictionary, they are processed against this data so there is a running record of what remains to be matched. When empty, a valid, complete dictionary has been found.

Next, `complete_dict` tries to select from the parent's symbols to complete the kid's code dictionary. For each of the parent's symbols, the algorithm checks a number of conditions. If the symbol is already being used, it is not included. If it is a prefix of or a prefix to a symbol already included in kid, then it is not included. None of the symbols used can violate the prefix property. If the symbol overlaps another present and that overlap is in the data, it is not used. For example, if `0123` is a symbol already included

in the kid and the sequence `012345` occurs in the input, adding `345` as a symbol would create a conflict. As the input is compared against symbols to be encoded, the `0123` will be matched leaving only `45` remaining. Therefore, `345` would not be found and is therefore not a valid symbol. Finally, given the symbols already selected, if the symbol would not match when compared against the input data, it is not included. For example, if `0123` is already part of the kid's genes, the symbol `12` from dad, while present in the data and an effective member of dad's genes, is unnecessary as part of the kid's genes. It will not match any of the digits that are not already covered.

Then, `complete_dict` must find whatever symbols are necessary to fill in the "gaps". If the input data has not been completely matched, the function must find symbols which do this and are a valid member of the dictionary. The function begins this process by enumerating all possible symbols for the remaining data, of a limited length. Because of the method of conflict resolution used later in the function, discussed below, the only possibilities enumerated at this point are symbols no more than half the length of the maximum allowed symbol. For example, if `12345` is what remains of the input to match and the maximum symbol length allowed is 4, the remaining possibilities are limited to a length of 2 and are therefore `1`, `12`, `2`, `23`, `3`, `34`, and `4`. Keeping the choices shorter at this stage pays dividends later. From these possibilities, the function removes any which conflict with already chosen symbols. This includes prefixes and any known overlaps (these are tracked as they are identified).

The function then randomly selects a new symbol from this list to attempt to add. If it overlaps one already being used, in the same fashion as discussed earlier, it is not kept. Finally, if it is contained within any symbol already being used, if is also not kept. This prevents situations like the one presented above: if `0123` is already being used because that sequence is present in the input but `12` is also added, when encoding, if `12` is compared against the input data before `0123`, the input data at those digits will become `['0','3']`. Then, `0123` will not match. This renders `0123` ineffective and strands those `0` and `3` sequences without any symbol to match against them. Therefore, such combinations of symbols must not be allowed because they do not form a valid, complete dictionary. Finally, there is one more confirmation that the symbol will be used. It is passed to `single_encode` with the current state of the data and run. This serves the dual purpose of updating the data for the next iteration of the loop. The symbol is now added and its prefix and overlap information are added for tracking as well.

After adding as many symbols as possible in this fashion, there may be the issue that some data remains that does not have a symbol to match, but there are no more valid possibilities after considering prefixes, overlaps, and other conflicts. The algorithm must now perform "conflict resolution". First, it identifies what digits in the input remain unmatched and where they are located in the input. It selects the first i.e. the one closest to the front to process first. It looks for the symbol which precedes this unmatched digit or digits and attempts to add the unmatched digits to the back of this symbol. For example, if `3` has not been matched yet and is preceded by `012` where it resides in the input data, the algorithm will get remove `012` as a symbol and see if `0123` can be used.

This may fail for a number of reasons, including if the unmatched digit in question is the very first digit in the input and therefore there is no preceding symbol. It may also fail if `0123` is an overlap with other data elsewhere in the input. However, if `0123` passes all checks, it will be added and the algorithm will move on, either to completion or back to looking for additional symbols to incorporate into the dictionary. It is also important to note that a combination of unmatched digits plus its preceding symbol cannot be allowed to form a new symbol that is longer than the maximum allowed. For example, if the maximum symbol length is 4 and the algorithm is working on the unmatched digit `4` preceded by `0123`, the algorithm cannot use `01234` so will attempt to use `1234` as its new symbol and the `0` will be addressed in a future round of conflict resolution.

If adding the digit to its preceding symbol fails, the algorithm will attempt to add it to the symbol which follows it. The same checks are performed: it will be limited to the maximum symbol length, confirmed it is not a prefix, overlap, or other conflict, and that after removing the old symbol, the new one is still needed to match some of the data. If successful, the proposed new symbol will be added and again, the algorithm will move on to either completion of the dictionary or back to looking for symbols to incorporate.

If both of those approaches fail, both the symbol before and after the unmatched digits are removed and the algorithm goes back to the top to reassess what new symbol possibilities that presents. If all of that fails, the algorithm can fail cleanly. It will issue a message that it was unable to find a complete set of symbols and it will continue. Depending on the situation, this may mean the algorithm does not have a new genetic kid to add to the population at this point or it may mean it will try again to generate a new random dictionary. Either way, failing cleanly and proceeding is more effective and efficient overall than continuing to hunt for a complete solution to the set of symbols in question. The Genetic Algorithm accommodates when this happens and it does not prevent its search for an optimal solution.

Mutation

If the kid's dictionary is successfully formed, it is passed back to the `crossover` function where it is checked for mutation using the `CodeDictionary` class's built-in `mutate` function. Mutation does not always occur. A test is conducted against the probability of mutation; this GA mutates with a probability of 0.33. If mutating, the function randomly selects a symbol. If longer than 1 digit, it splits the symbol into two new ones. If these do not violate the prefix property, they are kept and the updated set of symbols and all necessary variables are updated. This mutation experiments with whether shorter symbols in this code dictionary or its children can have a positive effect on the overall compression ratio.

Post-Processing

In the final steps of the program, it compresses the input and the custom bytes output with a variety of methods. This comparison against other compression methods is discussed in further detail in the next sections. The program also calculates the minimum compression ratio, maximum compression ratio, and average compression ratio of each generation of the genetic algorithm, so the effects of its reproduction can be observed over its generations. The program saves these statistics to a file and graphs them for visualization.

Experimental Procedure and Analysis of Results

In order to test the genetic algorithm, I used a variety of crafted input integers both compressible and incompressible. Obviously, the algorithm was incapable of compressing incompressible integers such as those randomly generated. Here is an example of how the Genetic Algorithm processes a random input integer:

Initial Variables:

Number of Digits in Input	1024
Maximum Symbol Length	6
Population Size	16
Number of Generations	20

Results:

Runtime	243.20 seconds
Compression Ratio Achieved	0.38172043
Number of Bytes of Input as Integer	426
Number of Bytes in Custom Output	1116

Input Integer:

```
626594656350028055606067716719640248565087570357286214101684563942342385539896846879818971814326016 5
210467615257986526403613409563504636193712749632301705981947285566261393927191036008677657598219366 9
678553497441999988646474607742323460021485867727296702749201576015878370733109645222550335803725611 2
648542321581380539142071358083878665306068059194246711830600989910522398654468304391258484419912802 0
451099011080836844853196296115883117129183304175625028722582173608975296586395304487914526638369394 1
323278921727848362601726921718988757716204572439475857659660605610249798738944263109987131306283141 1
449397437508857254380238106259415138035085293072803677358190057226968136464583257284328059214543101 7
219408778978690706574487084773134447513944030976617481009796040485652380239775835440060420126081812 6
371940873195433555769284266443785745510873250057995162982668689258203790253028235958127625016212146 4
436754365298298198936955451002670526119428724888230285474039571074455032159541481121164109759896038 5
531713810937840341595301
```

Code Dictionary:

This is the code dictionary evolved by the Genetic Algorithm.

```
| 257284: 01010110 | 966060: 0110000  | 038553: 00000110 | 060677: 1111000  | 853196: 1001001  |
| 653060: 0111100  |   1954: 00101011 | 565087: 11001111 |    898: 11110111 | 513803: 0101001  |
| 561024: 0110010  | 397437: 01011110 | 554510: 0001001  |    656: 11110011 | 585765: 0101110  |
| 810097: 00110111 | 260172: 0111010  | 958127: 00011011 | 670274: 1101100  | 132327: 0111000  |
```

```
|  16719: 11101110 | 260818: 0011000  |   6644: 00101010 | 592145: 00111110 | 084773: 0100100  |
| 028547: 00001110 | 431017: 00111111 |  64109: 0000000  |  88311: 01111011 | 989369: 0001000  |
| 692842: 00100110 | 692171: 01110110 | 436529: 00011110 |   6805: 10010110 | 093784: 1110100  |
| 784836: 01110011 | 562502: 01111111 | 986526: 10110010 | 134447: 01001010 | 960404: 0011100  |
| 598219: 11000110 | 126485: 1110010  | 443675: 0001100  | 686892: 0001110  | 968468: 10101010 |
| 310964: 11011110 | 712918: 0111110  |   0556: 11101111 | 513944: 01001011 | 378574: 0010110  |
|  79951: 00100111 | 285566: 10111110 | 403957: 0001010  | 798189: 10101011 |  28235: 00001111 |
|   7358: 11111010 | 774232: 1101000  | 690706: 01000110 | 791452: 10001110 | 280204: 1000110  |
| 287248: 00010110 | 615257: 1011000  |    307: 1111100  | 572862: 10011111 | 813805: 11101010 |
| 671183: 1001100  | 811211: 00001010 | 391420: 11101011 | 226968: 0100000  | 658639: 1000100  |
|   8823: 0000110  |    414: 10100110 | 920157: 11011010 | 713130: 0110110  | 677657: 1100010  |
| 125848: 1000010  | 582037: 00011111 | 830439: 01101110 | 802397: 00111011 | 855398: 1010100  |
| 778978: 0100010  |   6638: 01110111 | 759896: 0000001  | 845639: 1010010  | 892172: 01110010 |
| 372561: 1110001  | 872258: 1000000  | 023810: 0101100  | 625941: 0101000  | 927191: 1100000  |
|   0341: 11100111 | 280367: 00011010 | 981947: 1011110  | 243947: 01101111 | 403613: 10110011 |
| 894426: 0110100  | 141016: 1010001  | 713580: 1110110  | 210467: 10101111 | 350028: 1111010  |
| 902530: 0010010  | 836844: 1001000  | 991052: 10011011 | 940873: 00110110 | 979873: 0110001  |
|    136: 01001110 | 369394: 01111010 |   6174: 0011010  | 310998: 01101011 | 647460: 11001110 |
| 346002: 1101001  | 036008: 1100001  | 574487: 01000111 | 620457: 01100111 | 239865: 1001110  |
| 030976: 0100110  |   5703: 10011110 | 423215: 11100110 |  62831: 01011010 | 212146: 0010001  |
| 126371: 00110010 | 411449: 01100110 | 998864: 11001011 | 510990: 10001010 | 033580: 1110000  |
| 330417: 01111110 | 296115: 1001010  | 217360: 1000001  | 856523: 00111010 | 026705: 00000111 |
|    107: 11010111 | 321595: 0000010  | 772729: 11010110 | 148586: 1101010  | 190057: 11111110 |
|    400: 11110110 | 601587: 11011011 | 718143: 1010110  | 301705: 10111011 | 749632: 10111010 |
| 625016: 0010000  | 837073: 1101110  | 409563: 1011010  | 522255: 11011111 |  44550: 11111111 |
| 530448: 10001011 | 464583: 01001111 |  82981: 00010111 | 855349: 1100100  | 838786: 1010000  |
| 366967: 11000111 | 423423: 10100111 | 508529: 0101010  | 171381: 0000100  | 441991: 10000110 |
| 744199: 11001010 | 335557: 11110010 | 595301: 11111011 | 640248: 1100110  | 219408: 0100001  |
| 626594: 1011100  | 604201: 00101111 | 919424: 10010111 | 504636: 10110110 | 758354: 0011110  |
| 629826: 0010100  | 261194: 00001011 | 193712: 10110111 |  11080: 10001111 | 060098: 10011010 |
|  25438: 01010111 | 260165: 10101110 |    446: 01101010 | 897529: 10000111 | 875771: 1111110  |
| 325005: 00110011 | 508857: 01011111 |   3280: 01011011 | 551087: 00101110 | 261393: 10111111 |
```

Figure 1 shows the minimum, average, and maximum compression ratios in red, blue, and green, respectively, shown per generation:

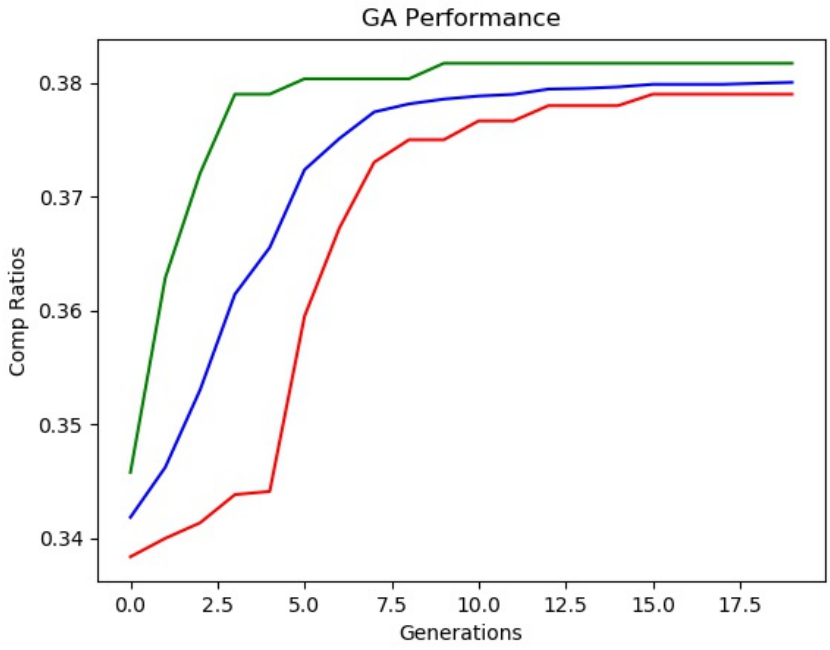


Figure 1

As can be easily observed, because of the lack of repetition and any pattern in the input, the dictionary requires a very high number of symbols to completely encode the input. Such a large dictionary and high number of symbols necessitate long codes; altogether these result in a longer output than the original integer.

A compressible integer, however, can be successfully compressed by this algorithm. Here is an example case:

Initial Variables:

Number of Digits in Input	4000
Maximum Symbol Length	8
Population Size	10
Number of Generations	10

Results:

Runtime	59.27 seconds
Compression Ratio Achieved	4.973054
Number of Bytes of Input as Integer	1661
Number of Bytes in Custom Output	334

Input Integer:

This is how the 4,000 digit input integer was generated:

```
'1122334455667788990099887766554433221100'*100
```

Code Dictionary:

This is the code dictionary evolved by the Genetic Algorithm.

```
| 11223344: 011   |  5566778: 010   |    43322: 000   |     1100: 101   |
|   899009: 111   |       98: 110   |       54: 100   |   877665: 001   |
```

Figure 2 shows the minimum, average, and maximum compression ratios in red, blue, and green, respectively, shown per generation:

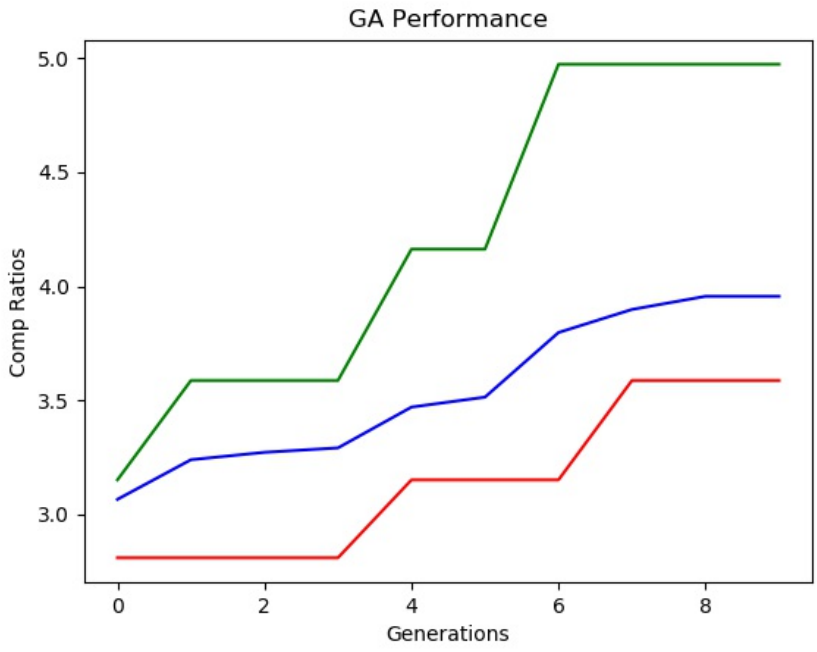


Figure 2

It can be observed in both scenarios how the Genetic Algorithm works to improve its solution through the generations. The best, worst, and average compression ratios all follow a positive trend as each generation breeds stronger members and eliminates

w eaker ones. It should also be noted how much smaller the code dictionary is w hen there is repetition and pattern in the input data.

Follow ing are additional results generated from compressible integer inputs.

Input Variables:

Input	Figure	Number of Digits	Maximum Symbol Length	Population Size	Number of Generations
`'12345678910'*20`	3	220	8	10	10
`'12345678910111213141516171819 20'*10`	4	310	8	10	10
`'10110210310410510610710810911 0'*100`	5	3000	8	10	10
`'12345678900987654321'*200`	6	4000	8	10	10
`'12345678910'*400`	7	4400	8	10	10
`range(1000,1100)*10`	8	4000	8	10	10

Results:

Input	Runtime (s)	Compression Ratio Achieved	Number of Bytes of Input as Integer	Number of Bytes in Custom Output
`'12345678910'*20`	444.347	3.791666667	91	24
`'12345678910111213141516171819 20'*10`	104.749	2.388888889	129	54
`'10110210310410510610710810911 0'*100`	450.910	3.787234043	1246	329
`'12345678900987654321'*200`	150.810	5.206896552	1661	319
`'12345678910'*400`	480.195	6.9732824427	1827	262
`range(1000,1100)*10`	226.640	1.53512014787	1661	1082

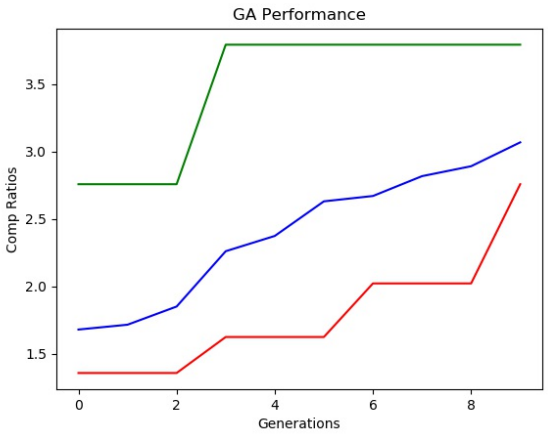


Figure 3

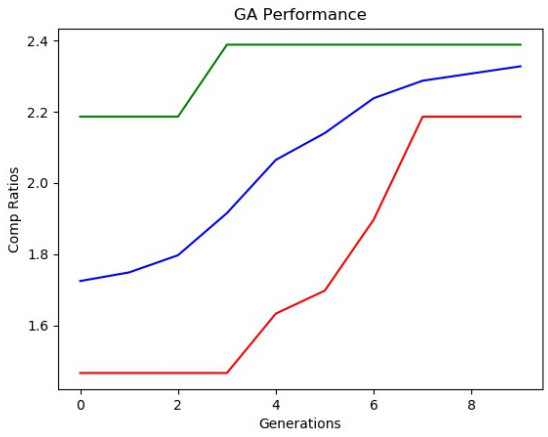


Figure 4

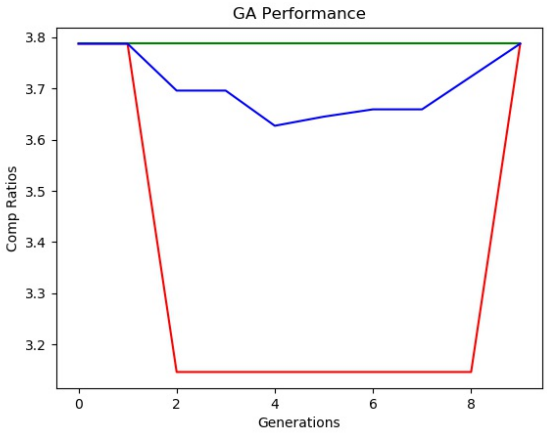


Figure 5

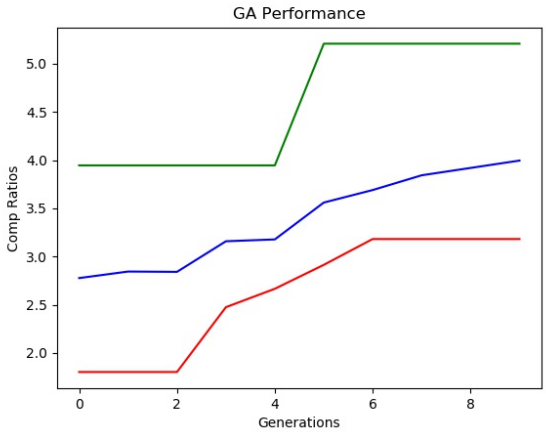


Figure 6

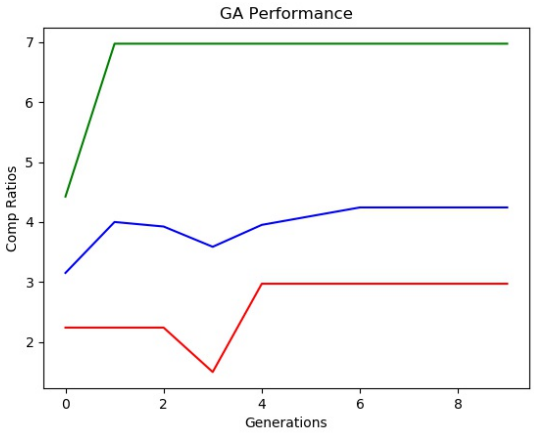


Figure 7

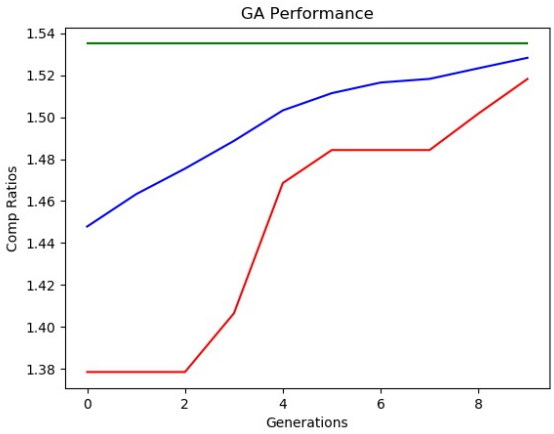


Figure 8

Because the algorithm is capable of handling failures to complete a dictionary, some early generations do not always contain the specified number of members. This results in the behavior seen in Figures 5 and 7, where the minimum and average compression ratios for certain generations decrease before increasing again. New members with lower compression ratios were added to the population, bringing the population size up to meet the limit defined by the algorithm. Figures 5 and 8 demonstrate scenarios where the algorithm happened to find a good solution right away and subsequent generations did not evolve a stronger solution.

A maximum symbol length of 8, population size of 10, and 10 generations showed as an optimal combination of variables to allow the genetic algorithm to produce a good solution in a reasonable amount of time. Smaller symbol lengths constrained the ability of the algorithm to find higher compression ratios, as longer symbols can often yield better local and therefore better overall compression. Because the input was fixed and every member of the population was matching a code dictionary to that same input, there was never a high degree of diversity in the population. This meant producing and maintaining a large population was unnecessary; it would simply mean more duplicates that would need to be thrown out. 10 was a large enough size to allow the diversity necessary for good evolution but avoided extra overhead. Similarly, 10 proved to be a suitable number of generations. This allowed the GA enough iterations to evolve a good solution but did not waste time running extra generations that would fail to find any further improvements.

Applicable Standards

To benchmark the performance of the algorithm, I compress the input with other popular compression methods: Deflate (zlib), BZip2, and LZMA. All three are popular lossless compression techniques. Deflate uses Lempel-Ziv-Storer-Szymanski (LZSS) and Huffman coding techniques. LZSS is a dictionary coding technique which makes replacements of input symbols with references to a dictionary location of the same string. Huffman coding is "the classic" dictionary technique which calculates the frequency of symbols in the input and maps the most frequently-occurring symbols to the shortest output codes and the least frequent symbols

to the longest output codes. BZip2 uses, at its heart, the Burrow s-Wheeler algorithm. Lempel-Ziv-Markov (LZMA) is another dictionary technique w hich dynamically encodes each bit based on a range encoding method. Each of these, being fellow lossless compression techniques, are good comparators for the GA.

In order to compare, I compressed both a serialized version of the code dictionary plus the output and the original input as an integer w ith each compression method.

As can be seen in the table below, my custom compression method for the code dictionary and output alw ays outperformed a serialized then compressed version of the same for random input. Though all failed to successfully compress the data, in fact requiring more space than the original input, the custom method achieved the best size of these compression methods. This makes sense as the custom method w as specifically tailored to the structure of the algorithm's output. For compressible input, how ever, the other compression methods outperformed the GA. This is because w hat w as serialized to be compressed by these algorithms w as the code dictionary plus the binary encoding of the input integer w hich is the input's symbols replaced by its codes. The fact that the input w as patterned means its translation to its binary codes w ould also be patterned and therefore compressible. Of the three alternate compression methods tested, the Deflate algorithm had the best result every time.

Input	Bytes of Input as Integer	Custom Output Package	zlib Package	bz2 Package	lzma Package
Random	213	518	713	692	748
Random	426	1091	1400	1287	1344
Random	426	1116	1431	1321	1396
Random	426	460	691	718	736
Patterned	91	24	49	78	108
Patterned	129	54	82	121	144
Patterned	1246	329	92	130	152
Patterned	1661	334	107	148	172
Patterned	1827	262	61	81	120
Patterned	1661	319	80	110	140
Patterned	1661	1082	612	787	656

We can also compare the performance of the algorithm to simply compressing the input integer, as its bytes, via other compression methods. For random inputs, of course the custom, Deflate, BZip2, and LZMA methods all failed to compress, w ith Deflate consistently having the best results of those. For compressible integers, simply attempting to compress the input integer w ith Deflate, BZip2, or LZMA w as unsuccessful. The Deflate algorithm (zlib) performed the best of the three alternate compression methods, but still had an output larger than the original input.

Input	Bytes of Input as Integer	Custom Output Package	zlib Input Integer	bz2 Input Integer	lzma Input Integer
Random	213	518	224	356	276
Random	426	1091	437	588	488
Random	426	1116	437	588	488
Random	426	460	437	592	488
Patterned	91	24	102	175	148

Patterned	129	54	140	206	192
Patterned	1246	329	1257	1598	1308
Patterned	1661	334	1672	2053	1724
Patterned	1827	262	1672	2067	1724
Patterned	1661	319	1838	2218	1888
Patterned	1661	1082	1672	2065	1724

These results show that the best application of the genetic algorithm in this case, then, is to use it to develop a code dictionary and its output but combine that with another compression algorithm to produce the final result. As discussed previously, popular compression methods often are combinations of different algorithms. The genetic algorithm is consistently much slower than the other methods, but combined with another method with a negligible effect on the overall runtime results in the smallest output data size for compressible input.

Security and Privacy Considerations

Because this algorithm develops a custom encoding for every input, it could assist in improving security and privacy of data. The program as written packages the encoding and output together so that a recipient with the decompression program can easily process the custom dictionary and decompress the data accordingly. However, the program could easily be adapted so the dictionary and the compressed data itself are stored separately. That means they could be transported separately as well. With that implementation, it would be more difficult for a nefarious party to get the data. The encoding dictionary by itself is useless. The compressed data by itself is also meaningless, until it can be decompressed and, unlike general compression algorithms, it requires its one unique developed solution to decompress/decode. An attacker would need to seize both in order for the data to be compromised.

Effects in Society

Exploring various data compression techniques can affect government, commercial, and private entities. In today's highly digital world, every sector can be positively affected by finding ways to store and transfer data more efficiently. This algorithm offers one such exploration into compression with a new use for genetic algorithms with a dictionary technique applied to integers.

Conclusions

During my Undergraduate years, I was a recipient of knowledge and as a Graduate student I feel I am now contributing to the knowledge of my field. In this project, I explored the applicability of Genetic Programming to data compression and found that it can successfully be used for that purpose, especially in conjunction with another compression method such as the Deflate algorithm. This is a new and improved method of compression for large integers which can be extrapolated to many forms of compressible input data.

Appendices

References

1. Fukunaga, Alex, and Andre Stechert. "Evolving nonlinear predictive models for lossless image compression w ith genetic programming." *Genetic Programming* (1998): 95-102.
2. Wu, Youfeng, and Mauricio Breternitz Jr. "Genetic algorithm for microcode compression." U.S. Patent No. 7,451,121. 11 Nov. 2008.
3. Üçoluk, Göktürk, and I. Hakki Toroslu. "A genetic algorithm approach for verification of the syllable-based text compression technique." *Journal of information science* 23.5 (1997): 365-372.
4. Kuthan, Tomáš, and Jan Lánský. "Genetic algorithms in syllable-based text compression." *Databases, Texts* 21 (2007).
5. Ng, Wee Keong, Sunghyun Choi, and Chinya Ravishankar. "Lossless and lossy data compression." *Evolutionary algorithms in engineering applications.* Springer, Berlin, Heidelberg, 1997. 173-188.
6. Ullah, Fahad, and Khaw aja M. Yahya. "A new data compression technique using an evolutionary programming approach." *International multi topic conference.* Springer, Berlin, Heidelberg, 2012.
7. Zaki, Mohammed, and M. Sayed. "The use of genetic programming for adaptive text compression." *International Journal of Information and Coding Theory* 1.1 (2009): 88-108.
8. Oroumchian, Farhad, et al. "Experiments w ith persian text compression for w eb." *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters.* ACM, 2004.
9. Burtscher, Martin, and Paruj Ratanaw orabhan. "High throughput compression of double-precision floating-point data." *2007 Data Compression Conference (DCC'07).* IEEE, 2007.
10. Beretta, Mauro L., Degli G. Antoni, and Andrea GB Tettamanzi. "Evolutionary synthesis of a fuzzy image compression algorithm." *Proceedings of the Fourth European Congress on Intelligent Techniques and Soft Computing.* Vol. 1. 1996.
11. Klappenecker, Andreas, and Frank U. May. "Evolving better w avelet compression schemes." *Wavelet Applications in Signal and Image Processing III.* Vol. 2569. International Society for Optics and Photonics, 1995.
12. Parent, Johan, and Ann Now e. "Evolving compression preprocessors w ith genetic programming." *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation.* Morgan Kaufmann Publishers Inc., 2002.
13. Robert, L., and R. Nadarajan. "Simple lossless preprocessing algorithms for text compression." *IET software* 3.1 (2009): 37-45.
14. Kattan, Ahmed. *Evolutionary Synthesis of Lossless Compression Algorithms: The GP-zip Family.* Diss. University of Essex, 2010.

Source Code

```
# -*- coding: utf-8 -*-
"""
Author: Madeline Hundley

This program uses a genetic algorithm in order to compress data.
For further information see the accompanying report.

Note: To change the level of logging statements, change the line at the beginning
of the main program.
- DEBUG is extremely verbose and shows detailed inner workings of the code.
- INFO shows the algorithm's steps.
"""

################################## IMPORTS #####################################

from bitarray import bitarray # used for codes and output data
import random # for probabilities and to generate input data
import logging # generates logging statements
import time # to test algorithm performance
from operator import attrgetter # for sorting
import math # for rounding, min, max, ceil, etc.
import numpy.random as nprandom # probabilities
import matplotlib.pyplot as plt # to graph stats
from datetime import datetime # naming stats
import csv # for saving stats

# Compare to other compression methods
from base64 import b64encode, b64decode
import ujson
import zlib
import lzma
import bz2

################### ALGORITHM CLASS AND FUNCTION DEFINITIONS ####################

class CodeDictionary(object):
    """
    Class for the members of the genetic population -- a dictionary mapping input
    digits to output codes

    Being variable-length, the codes must have the "prefix property", which
    requires that there is no whole codeword in the system that is a prefix of
    any other codeword in the system.

    Where necessary/applicable, attributes are of type 'string' to account for
    symbols that begin with '0'. This also makes the individual digits of the
    symbols easier to access.

    Instance attributes:
    symbols (list) - symbols to be mapped (str)
    codedict (dict) - keys: symbols to be mapped (str)
                      values: codes (bitarray)
    outdata (bitarray) - binary encoding of the input data
    outbyes (bytes) - the packed final product: codedict and outdata
    comp_ratio (float) - the compression ratio achieved

    Class attributes:
    pm - (float) the probability, Pm, between 0 and 1, of a mutation occuring in
                 the "genes" of the data
    pc - (float) the probability, Pc, between 0 and 1, of crossover occuring when
                 forming children from the "genes" of parent data
    """

    pm = 0.33 # the probability of mutation
    pc = 0.9 # the crossover probability


    def __init__(self, codedict):
        """
```

```
        Initialize a new codedict
        """
        self.codedict = codedict
        self.symbols = list(codedict.keys())
        # Checking for this allows for dummy CodeDictionaries to be created
        if any(self.codedict):
            logging.debug("Initializing new Code Dictionary...")
            self.encode()
        return

    def __repr__(self):
        """
        Represent the CodeDictionary object
        """
        return str(self.codedict)

    def __str__(self):
        """
        Print the code dictionary
        """
        NUM_COLS = 4
        CODE_WIDTH = 8
        output = ""
        rows, rem = divmod(len(self.symbols), NUM_COLS)
        for i in range(rows):
            row = "|"
            for j in range(NUM_COLS):
                row += "{:>{w}}".format(self.symbols[(i*NUM_COLS)+j],
                                        w=MAX_SYM_LEN+1)
                row += ": "
                row += "{:<{w}}".format(self.codedict[
                    self.symbols[(i*NUM_COLS)+j]].to01(),w=CODE_WIDTH)
                row += "|"
            output += row
            output += "\n"

        if not rem:
            output += "      Compression Ratio: "
            output += "{:.7f}".format(self.comp_ratio)
            return output

        row = "|"
        for i in range(-rem,0):
            row += "{:>{w}}".format(self.symbols[i],w=MAX_SYM_LEN+1)
            row += ": "
            row += "{:<{w}}".format(self.codedict[self.symbols[i]].to01(),
                                    w=CODE_WIDTH)
            row += "|"
        output += row

        output += "\n      Compression Ratio: "
        output += "{:.7f}".format(self.comp_ratio)
        return output

    def __eq__(self, other):
        """
        Evaluates to True if this CodeDictionary and another have the same
        symbols and the same compression ratio
        """
        if self.symbols != other.symbols: return False
        if self.comp_ratio != other.comp_ratio: return False
        else: return True # they are functionally the same

    def encode (self):
        """
        Performs the encoding of the input data to get the output data; also
        calculates the compression ratio achieved
        Input: None
        Output: outdata (bitarray), comp_ratio (float)
        """
        logging.debug("Encoding {}...".format(INSTR))
        outdata = bitarray()

        sym_len = 1
```

```
        sym = ''
        for i in range(len(INSTR)):
            sym += INSTR[i]

            # When a symbol is found, encode and look for next symbol
            if sym in self.codedict:
                outdata += self.codedict[sym]
                sym_len = 1 # reset
                sym = '' # reset
                continue # move on

            # If checking for a symbol larger than possible, throw an error
            sym_len += 1
            if sym_len > MAX_SYM_LEN:
                raise Exception ("Couldn't match a symbol to input data")

        # If there are digits left at the end unmatched
        if sym: raise Exception ("Digits left unmatched in input data")

        logging.debug("Outdata is {}".format(outdata))
        self.outdata = outdata

        outbytes = pack(self)
        comp_ratio = len(INBYTES) / len(outbytes)
        logging.info("Compression ratio is " + str(comp_ratio))

        self.outbytes = outbytes
        self.comp_ratio = comp_ratio
        return


    def mutate(self):
        """
        Checks if a mutation occurs and performs the mutation if so
        """
        # Test if mutation will occur; if not, simply return
        if random.random() > CodeDictionary.pm: return

        logging.info("Attempting mutation...")

        # Randomly select a symbol and divide it into two
        # (Unless it is only 1 digit long, then do nothing)
        sym = random.sample(self.symbols,1)[0]
        if len(sym) == 1:
            logging.info("Mutation failed")
            return
        sym1 = sym[:len(sym)//2]
        sym2 = sym[len(sym)//2:]

        # Make sure they are not prefixes
        for curr_sym in self.symbols:
            if sym1+sym2 == curr_sym: continue # skip
            if sym1 == curr_sym[:len(sym1)] or sym2 == curr_sym[:len(sym2)]:
                logging.info("Mutation failed")
                return

        logging.info("Dividing {} into two new symbols: {} and {}".format(
            sym, sym1, sym2))
        logging.debug("Before:\n{}".format(self))

        # Remove old symbol and add new ones
        self.symbols.remove(sym)
        self.symbols.append(sym1)
        self.symbols.append(sym2)
        self.codedict.pop(sym)
        # Get new codes for the new symbols
        self.codedict = complete_codes(self.codedict, self.symbols)
        logging.debug("After:\n{}".format(self))

        # Recompute outdata, outbytes, and compression ratio
        self.encode()
        return
```

```
def single_encode(sym, data):
    """
    Function to find, remove, and demarcate all instances of a symbol in data
    Input: symbol (str), data (list[str])
    Output: data (list[str])
    """
    logging.debug("--- Running symbol {} against data ---".format(sym))
    i = 0
    while i < len(data):
        data_set = data[i]

        if len(data_set) < len(sym): # symbol too long
            i += 1 # move on to next digit set
            continue
        elif data_set == sym: # perfect match
            data.pop(i) # just get rid of this set of data
            continue # move on to next digit set (at this index)
        elif data_set[:len(sym)] == sym: # symbol at front
            data[i] = data_set[len(sym):] # save just remainder
            continue # check this digit set again
        elif data_set[-len(sym):] == sym: # symbol at end
            data[i] = data_set[:-len(sym)] # save just remainder
            continue # check this digit set again
        elif len(data_set) == len(sym) and data_set != sym:
            i += 1 # move on to next digit set
            continue
        elif sym in data_set: # in middle of the digit set
            j = data_set.index(sym) # finds first occurrence
            # Save the data that came before symbol
            data[i] = data_set[:j]
            # Save the data that came after symbol
            data.insert(i+1, data_set[j+len(sym):])
            i += 1 # move to newly created next digit set
            continue # move on to next digit set

        i += 1 # if here, done with this digit set

    logging.debug("Data at end: {}".format(data))
    return data


def gen_codes(num_symbols):
    """
    Generates valid codes for the number of symbols
    Input: num_symbols (int)
    Output: codes (list[bitarray])
    """
    codes = [bitarray() for i in range(num_symbols)] # initialize

    def recursive_code(codes): # pass (portion of) codes
        if len(codes)//2 is 0: # recursion base case
            return codes
        else: # not done; divide and keep adding bits
            for b in range(len(codes)//2):
                codes[b] += bitarray('0') # "bottom" symbols get '0'
            for t in range(len(codes)//2, len(codes)):
                codes[t] += bitarray('1') # "top" symbols get '1'
            bot = recursive_code(codes[0:len(codes)//2])
            top = recursive_code(codes[len(codes)//2:len(codes)])
        return bot+top

    recursive_code(codes)
    return codes


def get_rand_codedict():
    """
    Function to randomly generate a new codedict
    Input: None
    Output: A codedict (CodeDictionary)
    """
    logging.debug("Generating a new random code dictionary...")
```

```
    # Use empty, dummy kid and parent for complete_dict to process
    new_codedict = complete_dict(dict(), CodeDictionary(dict()))
    logging.debug("New codedict: {}".format(new_codedict))
    return CodeDictionary(new_codedict)


def get_init_pop():
    """
    Function to randomly generate an initial population of code dictionaries
    Input: None
    Output: a population (list) of codedicts (CodeDictionary)
    """
    init_pop = [] # list for the population of codedicts
    for i in range(POP_SIZE): # try to generate as many codedicts as we need
        try:
            new_codedict = get_rand_codedict()
        except Exception as e:
            logging.warning("complete_dict failed with Exception of " \
                            "type {}: {}".format(type(e),e))
            pass
        else:
            # check codedict is not a duplicate; save it in the population
            duplicate = False
            for cd in init_pop:
                if new_codedict == cd: # make use of custom function
                    duplicate = True
                    logging.debug("new codedict is a duplicate; not keeping")
                    break # stop looking
            else: init_pop.append(new_codedict)

    return init_pop


def locate_unmatched(kid_syms, unmatched):
    """
    Function finds the locations in INSTR of the provided unmatched digits
    Input: kid_syms (list[str]) and unmatched (str)
    Output: locations (set(int))
    """
    logging.debug("Finding {} with locate_unmatched".format(unmatched))
    data_locs = set(range(INDATA_NUM_DIGITS))

    i = 0
    while i < len(INSTR):
        for j in range(1,MAX_SYM_LEN+1):
            if INSTR[i:i+j] in kid_syms:
                data_locs = data_locs - set(range(i,i+j))
                i += j-1 # advance
                break # stop looking
        i += 1

    un_locs = set()

    for loc in data_locs:
        if INSTR[loc:loc+len(unmatched)] == unmatched:
            for i in range(0,len(unmatched)):
                if loc+i in data_locs: pass
                else: break
            else:
                logging.debug("Found {} at location {}".format(unmatched, loc))
                un_locs.add(loc)

    if un_locs:
        return un_locs
    else:
        logging.debug("Returning None")
        return None


def complete_dict(kid, parent):
    """
    Function to take the start of a code dictionary, find all necessary symbols
    to complete it (based on the parent's symbols) and compute the codes
```

```
    Can accept a dummy kid and dummy parent to funtionally generate a new,
    random codedict; in that case it uses gen_codes to compute the codes

    Input: incomplete kid codedict (dict[sym:code]) and parent codedict
           (CodeDictionary)
    Output: complete kid codedict (dict[sym:code])
    """
    logging.debug("Completing the code dictionary...")

    data = [INSTR] # list of strings
    for sym in kid: data = single_encode(sym, data[:])

    # Try to use parent's symbols to complete the dictionary

    for psym in parent.symbols:
        if not any(data): break # done looking
        logging.debug("Trying second parent's {}".format(psym))

        # If the other parent already gave this symbol
        if psym in kid:
            continue # move on

        # If symbol is a prefix to/of one already being used
        for ksym in kid:
            prefix = False
            if ksym == psym[0:len(ksym)]: # ksym is at front of psym
                prefix = True
                break # no need to keep checking
            elif psym == ksym[0:len(psym)]: # pysm is at front of ksym
                prefix = True
                break # no need to keep checking
        if prefix: continue # move on to next parent symbol

        # If symbols overlaps one already in use
        if find_overlaps(psym, list(kid.keys())):
            continue # move on to next parent symbol

        # Check if the symbol would be used by the data
        testdata = single_encode(psym, data.copy()) # process data with new symbol
        if testdata == data:
            continue # move on to the next symbol

        # Use this parent symbol
        data = testdata
        kid[psym] = parent.codedict[psym]
        logging.debug("Keeping parent symbol {}.".format(psym))

    # Fill in any gaps left after using parent's symbols

    kid_syms = list(kid.keys()) # track symbols that will be used
    conflicts = set() # symbols once removed that should not be reused
    overlaps = dict() # symbols not used because they overlap
    for s in kid_syms: overlaps[s] = set() # initialize

    # Compute initial dict of prefixes to/of current symbols
    logging.debug("Computing prefixes to/of current symbols...")
    prefixes = dict() # initialize dict for tracking
    for s in kid_syms: prefixes[s] = set() # initialize
    for p in prefixes:
        # Add symbols that prefix this one, and this symbol itself
        for i in range(len(p)): prefixes[p].add(p[:i+1])
        # Add symbols prefixed by this one
        if len(p) == MAX_SYM_LEN: continue # this check does not apply
        for i in range(10**(MAX_SYM_LEN-len(p))):
            prefixes[p].add(p+str(i))

    while any(data): # while there is still data to match

        # Enumerate all possible symbols for remaining data
        logging.debug("Enumerating all possible symbols for remaining data...")
        sym_choices = set()
        for data_set in data:
            length = min(len(data_set),MAX_SYM_LEN//2)
            for i in range(len(data_set)): # location
```

```
            for j in range(1, length+1): # length
                sym_choices.add(data_set[i:i+j])

    # Remove conflicting symbols from possibilities
    logging.debug("Removing prefixes from choices...")
    # Never let prefixes be part of sym_choices
    for p in prefixes: sym_choices = sym_choices.difference(prefixes[p])
    if any(conflicts): # Remove conflicts from sym_choices
        logging.debug("Removing tracked conflicts: {}".format(conflicts))
        sym_choices.difference_update(conflicts) # remove all
    logging.debug("Removing tracked overlaps from choices...")
    for s in overlaps: # Remove overlaps from sym_choices
        sym_choices.difference_update(overlaps[s])

    # If there are valid choices; randomly select one of those
    if len(sym_choices) > 0:
        new_sym = random.sample(sym_choices,1)[0]
        logging.debug("Selected new_sym from choices.")
    else:
        new_sym = None
        logging.debug("There are no symbols left to choose from")


    # Do not keep any symbol that overlaps ones already chosen
    if new_sym:
        logging.debug("Checking new_sym {} for overlaps...".format(new_sym))
        new_sym_overlaps = find_overlaps(new_sym, kid_syms)
        if new_sym_overlaps:
            logging.debug("Keeping track of these; not using this symbol")
            for n in new_sym_overlaps:
                overlaps[n].add(new_sym)
            continue # start over, find a different new_sym
        else: logging.debug("No overlap conflict with new_sym")

    # Do not keep any symbol that is contained within another
    if new_sym:
        logging.debug("Checking if new_sym is in any current symbol...")
        contains = False
        for k in kid_syms:
            if new_sym in k:
                logging.debug("new_sym is inside {}".format(k))
                overlaps[k].add(new_sym)
                contains = True
        if contains:
            logging.debug("Not using this symbol")
            continue


    # If there are no valid choices; find matches for the remaining digits
    pre_sym = None
    if not new_sym: # Don't have a new symbol to try yet

        logging.debug("CONFLICT RESOLUTION")

        # Start at the front
        unmatched = data[0][:MAX_SYM_LEN]
        logging.debug("Trying to find digit(s) {} a symbol...".
                      format(unmatched))

        # Get first instance (or None)
        loc = sorted(locate_unmatched(kid_syms, unmatched))[0]
        logging.debug("Working on unmatched digit(s) at location {}".
                      format(loc))
        if loc is None: raise Exception("Failed to locate digit(s)")

        if loc is not 0:
            # Find the preceding symbol
            for i in range(1,MAX_SYM_LEN+1):
                if INSTR[loc-i:loc] in kid_syms:
                    pre_sym = INSTR[loc-i:loc]
                    logging.debug("Symbol {} precedes unmatched".
                                  format(pre_sym))
                    break # done looking
```

```
        # If a symbol preceding the unmatched digit(s) was successfully located,
        # see if it can be removed and a new one formed
        cons = set() # track conflicts for processing below
        if pre_sym and not new_sym:

            # Try adding this hanging digit to the preceding symbol
            new_sym = (pre_sym+unmatched)[-MAX_SYM_LEN:]
            logging.debug("Old symbol {} may be removed, seeing about {} " \
                          "instead".format(pre_sym, new_sym))

            logging.debug("- Checking if new_sym is a prefix...")
            for p in prefixes:
                if p == pre_sym: continue # skip this match
                if new_sym in prefixes[p]:
                    logging.debug("  pre_sym+unmatched created a " \
                                  "prefix with {}".format(p))
                    new_sym = None # undo the selection
                    break # no need to keep checking


            # Check if the proposed new_sym is in conflicts
            if new_sym and new_sym in conflicts:
                logging.debug("- Reforming with preceding symbol is a conflict")
                new_sym = None # undo this selection


            # See if new_sym overlaps any current symbols
            if new_sym:
                logging.debug("- Checking new_sym {} for overlaps...".
                              format(new_sym))
                new_sym_overlaps = find_overlaps(new_sym, kid_syms)
                if new_sym_overlaps == {pre_sym}:
                    logging.debug("- No overlap conflict with new_sym")
                    pass
                elif new_sym_overlaps:
                    logging.debug("- Not going to use this symbol.")
                    new_sym = None # go to next step


            # Attempting all updates; seeing if new_sym works
            if new_sym:
                logging.debug("- Removing current symbol {}".format(pre_sym))
                kid_syms.remove(pre_sym) # Remove from the chosen symbols
                prefixes.pop(pre_sym) # Remove it from prefix tracking
                overlaps.pop(pre_sym) # Remove it from overlap tracking
                try: kid.pop(pre_sym) # Remove it from kid if present
                except: pass # If not, that's fine

                logging.debug("- Resetting the data...")
                data = [INSTR] # Start with fresh data and reprocess
                for sym in kid_syms: data = single_encode(sym, data[:])

                # Double check the new symbol is needed
                testdata = data.copy()
                data = single_encode(new_sym, data.copy())
                if testdata == data: # the new symbol is not needed
                    logging.debug("- New_sym {} wasn't found...".format(new_sym))
                    new_sym = None
                else:
                    logging.debug("- Keeping symbol {}".format(new_sym))
                    kid_syms.append(new_sym)
                    logging.debug("- Current symbols: {}".format(kid_syms))
                    # Update overlap tracking
                    logging.debug("- Updating overlap tracking...")
                    overlaps[new_sym] = set() # initialize new empty set
                    overlaps[new_sym].add(new_sym[-1]) # easy kill
                    # Update prefix tracking
                    logging.debug("- Updating prefixes to/of current symbols...")
                    # Add symbols that prefix this one, and this symbol itself
                    prefixes[new_sym] = set() # initialize new empty set
                    for i in range(len(new_sym)):
                        prefixes[new_sym].add(new_sym[:i+1])
                    # Add symbols prefixed by this one
                    if len(new_sym) == MAX_SYM_LEN:
```

```
                continue # this check does not apply
            for i in range(10**(MAX_SYM_LEN-len(new_sym))):
                prefixes[new_sym].add(new_sym+str(i))
            continue # done with new symbol, go back to top


    # If that was not successful, see if the unmatched digit(s)
    # can be added to the symbol that follows them
    post_sym = None
    if not new_sym:
        logging.debug("No preceding symbol or there was a conflict.")

        # Find the trailing symbol
        for i in range(1,MAX_SYM_LEN+1):
            if INSTR[loc+len(unmatched):loc+len(unmatched)+i] in kid_syms:
                post_sym = INSTR[loc+len(unmatched):loc+len(unmatched)+i]
                logging.debug("Found symbol {} following unmatched " \
                              "digit(s)".format(post_sym))
                break # done looking


    # If a symbol following the unmatched digit(s) was successfully located,
    # see if it can be removed and a new one formed
    if post_sym and not new_sym:

        # Try adding the unmatched digit(s) to the trailing symbol
        new_sym = (unmatched+post_sym)[:MAX_SYM_LEN]
        logging.debug("Old symbol {} may be removed, seeing about {} " \
                      "instead".format(post_sym, new_sym))

        logging.debug("~ Checking if new_sym is a prefix...")
        for p in prefixes:
            if new_sym in prefixes[p]:
                logging.debug("post_sym+unmatched created a " \
                              "prefix with {}".format(p))
                logging.debug("Getting rid of that symbol instead...")
                cons.add(p)
                break # no need to keep checking

        # Check if we're at the front so the old symbol won't be readded
        if loc == 0:
            logging.debug("We're at the front so don't let the old symbol " \
                          "get re-added.")
            logging.debug("Adding to conflicts.")
            conflicts.add(post_sym)

        # Check if the proposed new_sym is in conflicts; this indicates add-
        # ing these digits to the front and to the back has failed
        # One reason for this is a "yo-yo" e.g. given '44083848', '44' and
        # '48' could keep gaining and losing '0838' in a loop
        if new_sym and new_sym in conflicts:
            logging.debug("Reforming with trailing symbol is a conflict.")
            new_sym = None # undo this selection

        if new_sym:
            logging.debug("~ Checking new_sym {} for overlaps...".
                          format(new_sym))
            new_sym_overlaps = find_overlaps(new_sym, kid_syms)
            if new_sym_overlaps == {post_sym}:
                logging.debug("~ No overlap conflict with new_sym")
                pass
            elif new_sym_overlaps:
                logging.debug("~ Not going to use this symbol.")
                new_sym = None # go to next step

        # Attempting all updates; seeing if new_sym works
        if new_sym:
            logging.debug("~ Making update; evaluating if it works")
            logging.debug("~ Removing current symbol {}".format(post_sym))
            kid_syms.remove(post_sym) # Remove from the chosen symbols
            prefixes.pop(post_sym) # Remove it from prefix tracking
            overlaps.pop(post_sym) # Remove it from overlap tracking
            try: kid.pop(post_sym) # Remove it from kid if present
            except: pass # If not, that's fine
```

```
            if any(cons):
                for c in cons:
                    logging.debug("~ Removing current symbol {}".format(c))
                    try: kid_syms.remove(c)
                    except: pass
                    prefixes.pop(c)
                    overlaps.pop(c)
                    try: kid.pop(c)
                    except: pass

            logging.debug("~ Resetting the data...")
            data = [INSTR] # Start with fresh data and reprocess
            for sym in kid_syms: data = single_encode(sym, data[:])

            # Double check the new symbol is needed
            testdata = data.copy()
            data = single_encode(new_sym, data.copy())
            if testdata == data: # the new symbol is not needed
                logging.debug("~ New_sym {} didn't work...".format(new_sym))
                new_sym = None
            else:
                logging.debug("~ Keeping symbol {}".format(new_sym))
                kid_syms.append(new_sym)
                logging.debug("~ Recording old symbol {} as a conflict".
                              format(post_sym))
                conflicts.add(post_sym)
                logging.debug("~ Current symbols: {}".format(kid_syms))
                # Update overlap tracking
                logging.debug("~ Updating overlap tracking...")
                overlaps[new_sym] = set() # initialize new empty set
                overlaps[new_sym].add(new_sym[-1]) # easy kill
                # Update prefix tracking
                logging.debug("~ Updating prefixes to/of current symbols...")
                # Add symbols that prefix this one, and this symbol itself
                prefixes[new_sym] = set() # initialize new empty set
                for i in range(len(new_sym)):
                    prefixes[new_sym].add(new_sym[:i+1])
                # Add symbols prefixed by this one
                if len(new_sym) == MAX_SYM_LEN:
                    continue # this check does not apply
                for i in range(10**(MAX_SYM_LEN-len(new_sym))):
                    prefixes[new_sym].add(new_sym+str(i))
                continue # done with new symbol, go back to top



        # If neither adding to the front or to the back was successful,
        # remove both to try again
        if not new_sym and pre_sym and post_sym:
            logging.debug("Both preceding and following symbols failed to " \
                          "form a good new symbol.")

            # Remove both and track so they aren't re-used
            conflicts.add(pre_sym)
            try: kid_syms.remove(pre_sym)
            except: pass
            try: prefixes.pop(pre_sym)
            except: pass
            try: overlaps.pop(pre_sym)
            except: pass
            try: kid.pop(pre_sym)
            except: pass

            conflicts.add(post_sym)
            try: kid_syms.remove(post_sym)
            except: pass
            try: prefixes.pop(post_sym)
            except: pass
            try: overlaps.pop(post_sym)
            except: pass
            try: kid.pop(post_sym)
            except: pass

            logging.debug("Resetting the data...")
```

```
            data = [INSTR] # Start with fresh data and reprocess
            for sym in kid_syms: data = single_encode(sym, data[:])

            continue


        # If unsuccessful, cannot proceed
        if not new_sym:
            raise Exception("Could not find a complete set of symbols.")

        # By this point, a new_sym has been chosen
        logging.debug("Trying new symbol {}".format(new_sym))

        # Remove any symbol(s) identified as needing removing
        if any(cons):
            logging.debug("Removing current symbol(s) {}".format(cons))
            conflicts.update(cons) # Track these so they aren't reused later
            for c in cons:
                try: kid_syms.remove(c) # Remove from the chosen symbols
                except: pass
                try: prefixes.pop(c) # Remove it from prefix tracking
                except: pass
                try: kid.pop(c) # Remove it from kid if present
                except: pass # If not, that's fine

            logging.debug("Resetting the data...")
            data = [INSTR] # Start with fresh data and reprocess
            for sym in kid_syms: data = single_encode(sym, data[:])

        # Double check the new symbol is needed
        testdata = data.copy()
        data = single_encode(new_sym, data.copy())
        if testdata == data: # the new symbol is not needed
            logging.debug("Didn't need to keep {}. Moving on...".format(new_sym))
            continue # go back and re-try without keeping this symbol

        # Keep the new symbol
        kid_syms.append(new_sym)
        logging.debug("Keeping symbol {}".format(new_sym))
        logging.debug("Current symbols: {}".format(kid_syms))
        # Update overlap tracking
        logging.debug("Updating overlap tracking...")
        overlaps[new_sym] = set() # initialize new empty set
        overlaps[new_sym].add(new_sym[-1]) # easy kill
        # Update prefix tracking
        logging.debug("Updating prefixes to/of current symbols...")
        # Add symbols that prefix this one, and this symbol itself
        prefixes[new_sym] = set() # initialize new empty set
        for i in range(len(new_sym)): prefixes[new_sym].add(new_sym[:i+1])
        # Add symbols prefixed by this one
        if len(new_sym) == MAX_SYM_LEN: continue # this check does not apply
        for i in range(10**(MAX_SYM_LEN-len(new_sym))):
            prefixes[new_sym].add(new_sym+str(i))


    logging.debug("Chose symbols: {}".format(kid_syms))

    return complete_codes(kid, kid_syms)


def find_overlaps(sym, list_syms):
    """
    Function compares the provided symbol against the others provided in the
    list; if it finds an overlapping symbol that exists in the indata, it
    returns it, otherwise returns 'None'
    Input: sym (str) and list_syms (list[str])
    Output: overlaps (set(str)) or None
    """
    overlaps = set()
    for k in list_syms:
        for i in range(1,len(sym)+1):
            if k[-i:] == sym[:i]: # k<->sym
                if k+sym[i:] in INSTR:
                    logging.debug("  Sym overlaps {} as {} in the data".
```

```
                                format(k,k+sym[i:]))
                    overlaps.add(k)
            if sym[-i:] == k[:i]: # sym<->k
                if sym+k[i:] in INSTR:
                    logging.debug("  Sym overlaps {} as {} in the data".
                                format(k,sym+k[i:]))
                    overlaps.add(k)

    if overlaps: return overlaps
    else: return None


def complete_codes (kid, kid_syms):
    """
    Function to take a complete set of symbols but incomplete codedict and
    complete the set of codes
    Input: kid (dict[sym:code]) and kid_syms (list)
    Output: codedict (dict[sym:code])
    """
    new_kid = kid.copy()
    codes = gen_codes(len(kid_syms))
    # See which members of kid get to keep their original symbols
    for k in kid:
        if kid[k] in codes: # if its code is valid and not already taken
            codes.remove(kid[k]) # remove it as an option
        else: # its code is not valid or has already been taken
            new_kid.pop(k) # make sure it gets a new code
    # Now, new_kid has symbols with their original codes, if valid

    # Next, assign remaining valid codes to remaining symbols
    for k in kid_syms:
        if k not in new_kid: # does not already have its symbol
            new_kid[k] = codes.pop()

    return new_kid


def crossover(mom, dad):
    """
    Function performing the crossover (and potential mutation) of "genes" from
    two parents to make two children
    Input: two parent codedicts (CodeDictionary)
    Output: two child codedicts (CodeDictionary)
    """
    logging.debug("Making children...")
    # Crossover point is approximately 1/2 of the symbols and codes
    cross = len(mom.symbols)//2

    # Randomly select mom and dad genes to pass down
    mom_genes = random.sample(mom.symbols,cross)
    dad_genes = random.sample(dad.symbols,cross)

    # start with genes from mom, complete with genes from dad
    kid1 = dict([(s[0],mom.codedict[s[0]]) for s in mom_ratios[:cross]])
    logging.debug("kid1 from mom:\n{}".format(kid1))
    try:
        kid1 = complete_dict(kid1.copy(), dad)
        kid1 = CodeDictionary(kid1)
        kid1.mutate() # check/execute mutation
        logging.debug("Kid 1:\n{}".format(kid1))
    except:
        logging.warning("complete_dict failed for kid1")
        # Getting a random code dictionary instead
        while True:
            try:
                kid1 = get_rand_codedict()
                break
            except:
                logging.warning("complete_dict failed")
                continue

    # start with genes from dad, complete with genes from mom
    kid2 = dict([(s[0],dad.codedict[s[0]]) for s in dad_ratios[:cross]])
    logging.debug("kid2 from dad:\n{}".format(kid2))
```

```
    try:
        kid2 = complete_dict(kid2.copy(), mom)
        kid2 = CodeDictionary(kid2)
        kid2.mutate() # check/execute mutation
        logging.debug("Kid 2:\n{}".format(kid2))
    except:
        logging.warning("complete_dict failed for kid2")
        # Getting a random code dictionary instead
        while True:
            try:
                kid2 = get_rand_codedict()
                break
            except Exception as e:
                logging.warning("complete_dict failed with Exception of " \
                                "type {}: {}".format(type(e),e))
                continue

    return kid1, kid2


def get_fitness(pop):
    """
    Function to calculate the fitness ratios of all codedicts in a population
    Input: a population (list) of codedicts (CodeDictionary)
    Output: a list of probabilities corresponding to the input population
    """
    logging.debug("Evaluating population...")
    xmax = max([x.comp_ratio for x in pop]) # ID the max codedict comp_ratio
    xmin = min([x.comp_ratio for x in pop]) # ID the min codedict comp_ratio
    if xmax==xmin: return [1/len(pop) for x in pop] # all codedicts are the same

    fitness = [((x.comp_ratio-xmin)/(xmax-xmin)) for x in pop] # initial scale
    fitness = [f/sum(fitness) for f in fitness] # adjust so sums up to 1
    logging.debug("Population fitnesses: {}".format(fitness))
    return fitness


def reproduce(pop):
    """
    Function to select parents according to their fitness within the population
    and get their children
    Input: a population (list) of codedicts (CodeDictionary)
    Output: the remaining population (list) minus the mom and dad,
            kid1 (CodeDictionary), and kid2 (CodeDictionary)
    """
    # If crossover hits, make new children
    if random.random() <= CodeDictionary.pc and len(pop) >= 2:
        logging.info("Reproducing children genetically...")
        popfit = get_fitness(pop)
        if popfit[:2] == [1.0, 0.0]: # Check for specific case
            mom, dad = pop.pop(0), pop.pop() # get the best choice and one other

        # Select 2, without replacement, with probabilities popfit
        else:
            mom, dad = nprandom.choice(len(popfit), size=2,
                                       replace=False, p=popfit)
            logging.debug("Mom, Dad indices: {}, {}".format(mom, dad))
            # Get the parents themselves, removing them from the population
            if mom > dad: mom, dad = pop.pop(mom), pop.pop(dad)
            else:         dad, mom = pop.pop(dad), pop.pop(mom)

        logging.debug("Mom:\n{}".format(mom))
        logging.debug("Dad:\n{}".format(dad))

        # Execute crossover
        kid1, kid2 = crossover(mom, dad)
        return pop, kid1, kid2

    # When crossover does not hit, return two randomly generated codedicts
    else:
        logging.info("Using random codedicts as children...")
        while True:
            try:
                kid1 = get_rand_codedict()
```

```
                break
            except Exception as e:
                logging.warning("complete_dict failed with Exception of " \
                                "type {}: {}".format(type(e),e))
                continue
        while True:
            try:
                kid2 = get_rand_codedict()
                break
            except Exception as e:
                logging.warning("complete_dict failed with Exception of " \
                                "type {}: {}".format(type(e),e))
                continue
        logging.debug("Kid 1:\n{}".format(kid1))
        logging.debug("Kid 2:\n{}".format(kid2))
        return pop, kid1, kid2

def get_next_pop(pop):
    """
    Function to take a population and from it produce the next population
    The DISCARD variable determines the portion of the population replaced in
    each generation
    Input: a population (list) of codedicts (CodeDictionary)
    Output: a population (list) of codedicts (CodeDictionary)
    """
    next_pop = pop.copy()

    # Get children through genetic reproduction
    while len(next_pop) < len(pop)+DISCARD: # make as many children as we need
        if len(pop) >= 2: # if there are valid parents, try and make children
            pop, kid1, kid2 = reproduce(pop)

            # Prevent duplicates from existing in the population
            for p in next_pop:
                if kid1 == p: # Make use of class's custom function
                    logging.info("kid1 is a duplicate; not keeping it")
                    break
            else: next_pop.append(kid1)

            for p in next_pop:
                if kid2 == p: # Make use of class's custom function
                    logging.info("kid2 is a duplicate; not keeping it")
                    break
            else: next_pop.append(kid2)

        else: # try and increase genetic diversity if necessary
            logging.info("Adding random codedict to children")
            while True:
                try:
                    temp = get_rand_codedict()
                    break
                except Exception as e:
                    logging.warning("complete_dict failed with Exception of " \
                                    "type {}: {}".format(type(e),e))
                    continue
            next_pop.append(temp)

    logging.debug("Children: {}".format(next_pop[POP_SIZE:]))
    # Mix the children into the population, according to fitness
    next_pop = sorted(next_pop, key=attrgetter('comp_ratio'), reverse=True)
    # Keep only the most fit of the population
    next_pop = next_pop[:POP_SIZE]

    return next_pop


def performance(func):
    """
    Wrapper function to capture runtime and statistics
    Input: a function (whose performance will be measured)
    Output: a function (returning run time, each generation's achieved
                        compression ratios, and the best result found by the
                        algorithm)
    """
```

```
    def wrapper():
        start_time = time.time()
        gen_comp_ratios, result = func()
        end_time = time.time()
        return end_time - start_time, gen_comp_ratios, result
    return wrapper


@performance
def algorithm():
    """
    Function to make all the generations according to a Genetic Algorithm
    Input: None
    Output: list of each generation's compression ratios (list[float]) for use in
            performance statistics of the algorithm, the codedict (CodeDictionary)
            with the best compression ratio at the completion of the algorithm
    """
    gen_comp_ratios = []
    for i in range(NUM_GENS):
        logging.info("Getting population {}...".format(i))
        if i==0: mypop = get_init_pop()
        else:    mypop = get_next_pop(mypop)

        mypop = sorted(mypop, key=attrgetter('comp_ratio'), reverse=True)
        if mylevel == logging.DEBUG or i%5==0:
            print("Population:")
            for codedict in mypop: print(codedict)
        else:
            print("Best of Population {}:\n{}".format(i,mypop[0]))

        gen_comp_ratios.append([m.comp_ratio for m in mypop])
        logging.info("Population {} has {} members.".format(i,len(mypop)))

    result = mypop[0]
    return gen_comp_ratios, result


def pack(result):
    """
    Function to take a CodeDictionary and construct the binary output package
    Input: (CodeDictionary)
    Output: compressed product to be written to disk, sent via network, etc.
            (bytes)
    """
    # Header Part 1: nsymd_bits
    nsymd_bits = MAX_SYM_LEN.bit_length()
    header = bitarray('0')*(4-nsymd_bits.bit_length()) + \
             bitarray(format(nsymd_bits,'b'))
    if MAX_SYM_LEN > 15: raise Exception("Header nsymd_bits problem")
    logging.debug("Header with nsymd_bits {}".format(header))

    # Header Part 2: ncode_bits
    ncode_bits = 0
    for s in result.codedict:
        ncode_bits = max(result.codedict[s].length(), ncode_bits)
    logging.debug("Maximum code bit size found: {}".format(ncode_bits))
    header += bitarray('0')*(4-ncode_bits.bit_length()) + \
              bitarray(format(ncode_bits,'b'))
    if ncode_bits > (2**4)-1: raise Exception("Header ncode_bits problem")
    logging.debug("Header with ncode_bits {}".format(header))

    # Header Part 3: nsyms
    nsyms = len(result.symbols)
    header += bitarray('0')*(10-nsyms.bit_length()) + \
              bitarray(format(nsyms,'b'))
    if nsyms > (2**10)-1: raise Exception("Header nsyms problem")
    logging.debug("Header with nsyms {}".format(header))

    # Header Part 4: npad_bits
    # ...defined and added after the body has been calculated
    logging.debug("Header npad_bits will be added later...")

    # Add the body: all nsymbs, symbols, ncodebs, and codes
```

```
    body = bitarray() # initialize
    for i in range(len(result.symbols)):

        # Body Part 1: nsymd
        nsymd = len(result.symbols[i]) # number of symbol digits
        body += bitarray('0')*(nsymd_bits-nsymd.bit_length()) + \
               bitarray(format(nsymd,'b'))

        # Body Part 2: the symbol
        for d in result.symbols[i]: # for the symbol
            body += bitarray('0')*(4-int(d).bit_length()) + \
                    bitarray(format(int(d),'b')) # each digit

        # Body Part 3: ncodeb
        ncodeb = result.codedict[result.symbols[i]].length() # number code bits
        body += bitarray('0')*(ncode_bits-ncodeb.bit_length()) + \
                bitarray(format(ncodeb,'b'))

        # Body Part 4: the code
        body += result.codedict[result.symbols[i]] # the code
    logging.debug("Body code:\n{}".format(body))

    # Add Header Part 4
    npad_bits = 8-((21+body.length()+result.outdata.length())%8)
    header += bitarray('0')*(3-npad_bits.bit_length()) + \
              bitarray(format(npad_bits,'b')) # number of trailing zeros
    logging.debug("Header with npad_bits {}".format(header))

    logging.debug("Header length in bits: {}".format(header.length()))
    logging.debug("Body length in bits: {}".format(body.length()))
    logging.debug("Outdata length in bits: {}".format(result.outdata.length()))
    logging.debug("Number of padding bits: {}".format(npad_bits))

    # Complete
    outbitarray = header + body + result.outdata + bitarray('0')*npad_bits
    outbytes = outbitarray.tobytes()

    logging.debug("Total output: {}".format(outbytes))
    logging.debug("Total length in bytes: {}".format(len(outbytes)))

    return outbytes


def compare_compress(result):
    """
    Function to compare the algorithm's compression to other popular compression
    methods
    Input: The best result CodeDictionary from the algorithm
    Output: Relevant statistics
    """
    logging.debug("JSONing dictionary and outdata...")
    result_serial = dict([(s,result.codedict[s].to01())
                          for s in result.codedict])
    # dict{str:str} -> ujson-str -> utf-8 bytes
    result_jsonb = str.encode(ujson.dumps((result_serial,result.outdata.to01())))
    logging.debug("JSON-bytes length: {}".format(len(result_jsonb)))

    stats = [len(zlib.compress(result_jsonb)),
             len(bz2.compress(result_jsonb)),
             len(lzma.compress(result_jsonb)),
             len(zlib.compress(str.encode(INSTR))),
             len(bz2.compress(str.encode(INSTR))),
             len(lzma.compress(str.encode(INSTR))),
             len(zlib.compress(INBYTES)),
             len(bz2.compress(INBYTES)),
             len(lzma.compress(INBYTES))]

    return stats



################################## MAIN PROGRAM ###################################

print("Welcome to Data Compression with a Genetic Algorithm")
```

```
# Specify desired logging statements
mylevel = logging.INFO
logging.basicConfig(level=mylevel, format='%(levelname)s: %(message)s')
random.seed(0)

# Define certain global variables
POP_SIZE = 20 # the number of members of the population
NUM_GENS = 20 # the number of generations each run should have
DISCARD = math.ceil(POP_SIZE/3) # replace ~1/3 of each generation's population
MAX_SYM_LEN = 8 # define maximum symbol length allowed

# Define the input integer
INDATA=2**1000
INDATA_NUM_DIGITS=len("%i"%INDATA)
INSTR = str(INDATA) # need string version to use digits
INBYTES = INDATA.to_bytes(math.ceil(INDATA.bit_length()/8),byteorder='big')
logging.info("INDATA is {}".format(INDATA))

# Execute the algorithm, with @performance capturing desired statistics
runtime, gen_comp_ratios, result = algorithm()
print("\nBest Result:\n{}".format(result))

#################### COMPRESSION COMPARISON AND STATISTICS ####################

logging.debug("Packing results...")

print("\nAlgorithm runtime: {:.6f} seconds".format(runtime))
print("Compression ratio achieved: {:.8f}".format(result.comp_ratio))

logging.info("")
logging.info("Original data size           : {}".format(len(INBYTES)))
logging.info("Package compressed custom    : {}".format(len(result.outbytes)))

stats = compare_compress(result)
logging.info("")
logging.info("Package compressed with zlib : {}".format(stats[0]))
logging.info("Package compressed with bz2  : {}".format(stats[1]))
logging.info("Package compressed with lzma : {}".format(stats[2]))

logging.info("")
logging.info("Original str data compressed with zlib : {}".format(stats[3]))
logging.info("Original str data compressed with bz2  : {}".format(stats[4]))
logging.info("Original str data compressed with lzma : {}".format(stats[5]))
logging.info("")
logging.info("Original int data compressed with zlib : {}".format(stats[6]))
logging.info("Original int data compressed with bz2  : {}".format(stats[7]))
logging.info("Original int data compressed with lzma : {}".format(stats[8]))


# Process the generational information for statistics
mins, maxs, avgs = [], [], []
for i in gen_comp_ratios:
    mins.append(min(i))
    maxs.append(max(i))
    avgs.append(sum(i)/len(i))

pltname = 'plt_'+datetime.now().strftime('%Y%m%d_%H%M%S')+'.png'

with open('stats.csv','a',newline='') as f:
    statwriter = csv.writer(f)
    statwriter.writerow([INDATA_NUM_DIGITS, MAX_SYM_LEN, POP_SIZE,
                         NUM_GENS, runtime, result.comp_ratio,
                         len(INBYTES),len(result.outbytes),pltname] + stats + \
                        [mins] + [maxs] + [avgs] + [INDATA, result.codedict])

################################## PLOTTING GRAPHS ##################################

logging.info("Plotting statistical data...")
plt.figure()
plt.xlabel('Generations')
plt.ylabel('Comp Ratios')
plt.title('GA Performance')
plt.plot(mins,'r') # Mininum, "worst" compression ratios plotted with red line
```

```
    plt.plot(maxs,'g') # Maximum, "best" compression ratios plotted with green line
    plt.plot(avgs,'b') # Average distances plotted with blue line
    plt.savefig('Images/'+pltname)
```