Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

Programming Project 2 - Bloom Filters

Project Goal

The goal of this project is to build a Bloom Filter in python and assess its accuracy and performance. Two text documents were provided with which to test this performance, *rockyou.ISO-8859-1.txt* (referred to as *rockyou.txt*) and *dictionary.txt*. Statistics are recorded during testing and the results are printed out at the end of the program.

Program Procedure

The program starts by initializing an instance of the `BloomFilter` class. This class is designed to take a maximum number of elements *n*, and a target probability of false positives *p* as initial inputs. The `BloomFilter` class then calculates the optimal bitmap size *m*, and number of hash functions *k* based on formulas provided by this online calculator. The formulas for *m* and *k* are shown in Figure 1.

```
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

Figure 1. Formulas for *m* and *k* based on *n* and *p*.

After these features have been determined, the program then inserts the words from *rockyou.txt* into the Bloom Filter. The insertion method works by taking one word at a time and hashing the string using a hash method from the `pycryptodome` library. The insert method then creates a list of *k* numbers by taking this hash and multiplying it by and adding it to an index value from 0 to *k*, then finally taking the modulus on the bitmap size *m*. The result is a list of *k* addresses in the bitmap that are flipped from 0 to 1. A code snippet of the functions that perform this operation is displayed in Figure 2.

Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

```python
def _convert_string_to_hashed_int(self, element:str):
    element = str(element)
    return int(self.hash_method.new(element.encode('utf-8')).hexdigest(), 16)

def determine_addrs(self, element) -> list:
    """
    Creates a list of bitmap addresses based on the bitmap size and
    the pre-determined number of hash functions k (self.num_hash_funcs)
    """
    int_val = self._convert_string_to_hashed_int(element)
    func = lambda k : (k * int_val + k) % self.bitmap_size

    return [func(k) for k in range(1, self.num_hash_funcs + 1)]

def insert(self, element) -> None:
    """
    Inserts an element into the Bloom Filter.

    Hashes the elemen value and flips the corresponding bits.
    """
    addrs = self.determine_addrs(element)
    for addr in addrs:
        self.bitmap[addr] = 1
```

Figure 2. The BloomFilter insertion methods.

After all words in *rockyou.txt* are inserted into the Bloom Filter, the performance checking begins. The `StatisticsTracker` class takes the `BloomFilter` instance as well as the list of words from *rockyou* and *dictionary*. At this point, a design decision was made to improve the performance. The words from *dictionary.txt* are stored in a list, but the words from *rockyou.txt* are stored in a set. The reasoning for this is further explained in the Discussion section below.

As the `StatisticsTracker` iterates through each word in *dictionary*, it checks the Bloom Filter by hashing the word and verifying that all *k* bits are set to 1. If any of the bits are set to 0, it means that the word is not in the Bloom Filter (a true negative). However, if all of the bits are set to 1, the word may either be a true positive or a false positive.

To distinguish between the true and false positives, the `StatisticsTracker` then checks the word from *dictionary* against *rockyou*. If the word is in the Bloom Filter but not in *rockyou*, then it is marked as a false positive. If the word is in both the Bloom Filter and *rockyou*, it is a true positive. Because of the nature of how a Bloom Filter works, a false negative should never occur if everything is working properly. However, a theoretical false negative would be a situation where the word returns a 0 bit from the Bloom Filter, indicating a negative, but it is actually in *rockyou*. This situation could occur from corrupted data or cosmic bit flipping.

Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

Once the `StatisticsTracker` has finished checking the words of *dictionary* against the Bloom Filter and *rockyou* it then tallies the totals, calculates the percentages, and prints the results. The program then terminates after printing out the total runtime.

Results

An example of a results printout from the program is shown in Figure 3.

```
(venv) (base) sweis@10-116-13-155 BloomFilters % python bloomFilter.py
Loading words from text files. This may take a minute...
Words in rockyou: 14344391
Total time to load files: 5.9855 seconds
Loading rockyou into Bloom Filter using 4 variations of hash method MD5 with target collision probability p = 0.075 (1 in 13)...
All words in rockyou loaded to Bloom Filter
40511275 of 77334941 bits are used. (52.38%)
Total time to fill Bloom Filter: 207.9942 seconds
Checking dictionary words in Bloom Filter...
Finished checking dictionary words in Bloom Filter
Total time checking dictionary: 9.7582 seconds
Results
--------
True Positives: 119151 words; 19.11%
True Negatives: 396501 words; 63.59%
False Positives: 107866 words; 17.30%
False Negatives: 0 words; 0.00%
Ratio of True to False Positives: 1.10
Total words in dictionary.txt: 623518
Total time running program: 225.9340 seconds
(venv) (base) sweis@10-116-13-155 BloomFilters %
```

Figure 3. Example of results printout.

Because *rockyou.txt* has 14,344,391 words in it (*n*) and the target probability (*p*) is 0.075, the `BloomFilter` class determines that it needs a bitmap of 77,334,941 bits long (*m*) and 4 variations of the hash algorithm (*k*). In this example, the hash method MD5 is being used.

The bitmap is 52.38% filled after inserting all words from *rockyou*, and took about 3.5 minutes to fill (the majority of the total program runtime). Checking all 623,518 words in *dictionary* against both the Bloom Filter and *rockyou* took just under 10 seconds. Design decisions to optimize this step are explained in the Discussion section below.

The `StatisticsTracker` returned the following results:

- 119,151 True Positives, 19.11% of the words in *dictionary*
- 396,501 True Negatives, 63.59% of the words in *dictionary*
- 107,866 False Positives, 17.3% of the words in *dictionary*
- 0 False Negatives
- A TP/FP ratio of 1.1

The total project runtime for this example was just over 3.75 minutes.

Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

Discussion

As mentioned above, design decisions can affect the performance of the program. One decision that significantly impacts the program runtime is the data structure that the words from *rockyou.txt* are stored in. When the words are stored in a `list`, the lookups take agonizingly long (on the order of over-night). However, when the words are stored in a `set`, lookups can be done in constant time O(1), significantly increasing the performance of the `StatisticsTracker`'s `check_validity` method.

Another design consideration affecting the performance is deciding what inputs to allow for parameter creation in the Bloom Filter. The online calculator was used to find optimal sizes based on the number of words in *rockyou.txt* and different false positive probabilities. Since the number of elements in the Bloom Filter *n* was predetermined by *rockyou.txt*, it felt natural to use this as a starting parameter. Since the desired probability of false positives *p* can shift depending on the application, this was a safe second choice to include as an input parameter. From these two starting points, the calculator was used to assess the remaining two parameters, bitmap size *m* and number of hash algorithms *k*. To limit the amount of parameter tweaking, it was decided in this program to calculate *m* and *k* based on the formulas provided by the calculator.

Figures 4 and 5 show the graphical relationship between *p* and *k*, and Figure 6 shows the output from the program when changing the value of *p* from 0.075 (1 in 13) to 0.01 (1 in 100). Notice how the optimal number of hashes *k* shifts from 4 when *p* is 0.075 in Figure 4 to 7 when *p* is 0.01 in Figure 4.

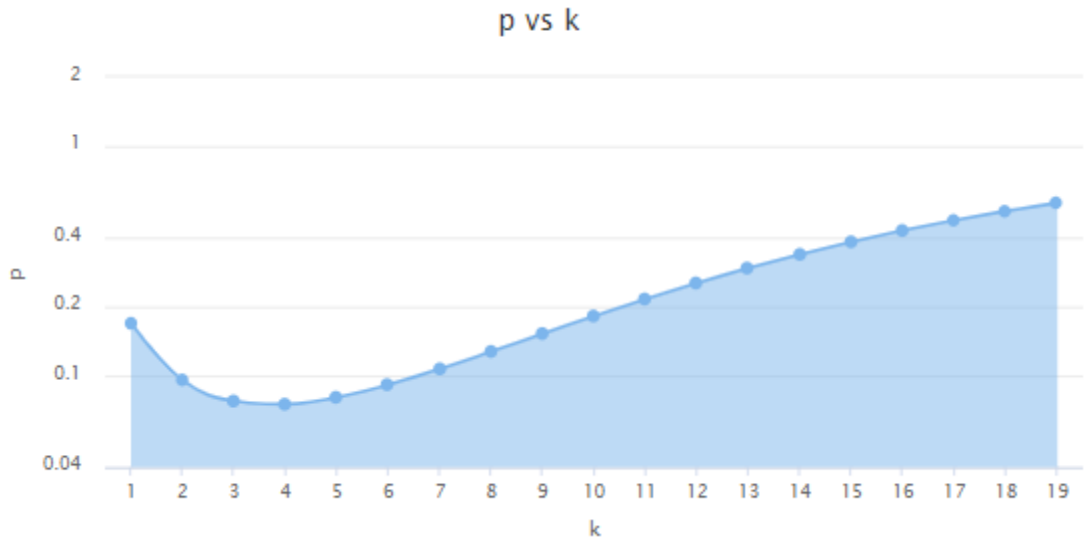

Figure 4. Graphical representation of relationship between *p* and *k* when *p* = 0.075

Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

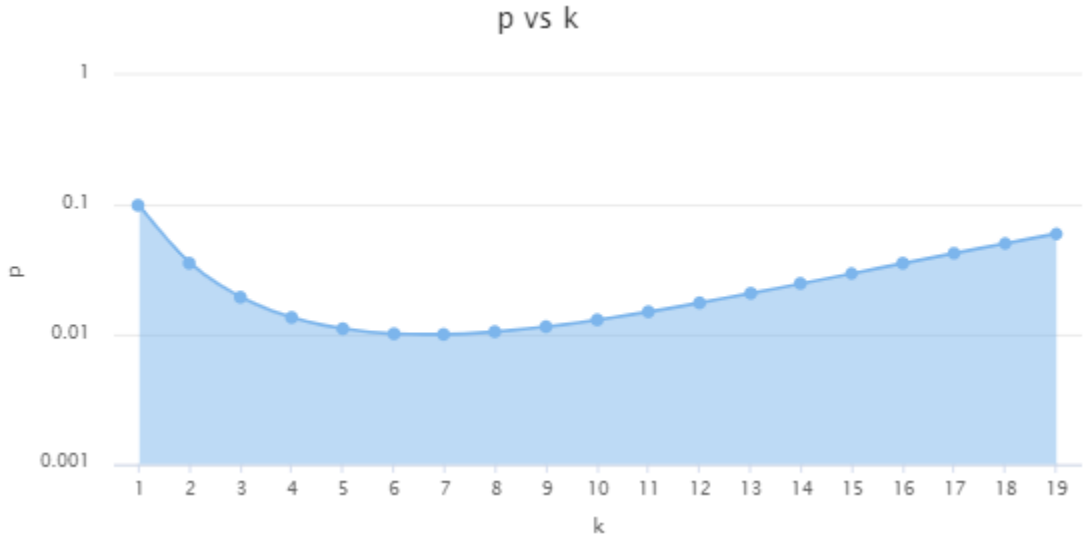

Figure 5. Graphical representation of relationship between *p* and *k* when *p* = 0.01

```
(venv) (base) sweis@10-116-13-155 BloomFilters % python bloomFilter.py
Loading words from text files. This may take a minute...
Words in rockyou: 14344391
Total time to load files: 6.1827 seconds
Loading rockyou into Bloom Filter using 7 variations of hash method MD5 with target collision probability p = 0.01 (1 in 100)...
All words in rockyou loaded to Bloom Filter
67982184 of 137491826 bits are used. (49.44%)
Total time to fill Bloom Filter: 217.3497 seconds
Checking dictionary words in Bloom Filter...
Finished checking dictionary words in Bloom Filter
Total time checking dictionary: 11.3574 seconds
Results
--------
True Positives: 119151 words; 19.11%
True Negatives: 441283 words; 70.77%
False Positives: 63084 words; 10.12%
False Negatives: 0 words; 0.00%
Ratio of True to False Positives: 1.89
Total words in dictionary.txt: 623518
Total time running program: 238.7119 seconds
(venv) (base) sweis@10-116-13-155 BloomFilters %
```

Figure 6. Example of results with *p* = 0.01

```
Loading words from text files. This may take a minute...
Words in rockyou: 14344391
Total time to load files: 9.8151 seconds
Loading rockyou into Bloom Filter using 10 variations of hash method MD5 with target collision probability p = 0.001 (1 in 1000)...
All words in rockyou loaded to Bloom Filter
95661017 of 206237738 bits are used. (0.46%)
Total time to fill Bloom Filter: 238.5485 seconds
Checking dictionary words in Bloom Filter...
Finished checking dictionary words in Bloom Filter
Results
--------
True Positives: 119151 words; 19.11%
True Negatives: 460165 words; 73.80%
False Positives: 44202 words; 7.09%
False Negatives: 0 words; 0.00%
Ratio of True to False Positives: 2.70
Total words in dictionary.txt: 623518
Total time checking dictionary: 31.8393 seconds
Total time running program: 285.9811 seconds
```

Figure 7. Example of results with *p* = 0.001

Seth Weiss
weissse@oregonstate.edu
CS 370 - Intro to Security
Programming Project 2 - Bloom Filters
Fall 2023

In the first example (Figure 3), the projected false positive rate (p = 0.075) was only 7.5% yet the actual false positives composed 17.3% of the words from *dictionary.txt*. When decreasing the projected rate to 1% (p = 0.01), the actual results reduce to 10.12%. In Figure 7, it is shown that decreasing the project false positive rate to 0.1% (p = 0.001) further decreases the actual false positive rate to 7.09%. There is a consistent overshooting of false positives from the respective targets. However, it does seem to trend in a desired direction. Notice also how with each decrease in p, the value of k increases from 4 to 7 to 10.

The comparison of these three examples demonstrates how the choice of p can significantly affect the performance of the Bloom Filter. However, as mentioned earlier, this decision can be made based on the application. While it primarily affects the proportions of true negatives and false positives, adjusting p also affects the value of k. This in turn slightly affects the runtime since the insertion method runs with a time complexity of of $O(n * k)$. Some situations might allow more collisions than others and thus justify this slight increase in runtime performance at the cost of an increase in collisions (false positives).