Bloom Filter

June 16, 2017

Student: Denis Ehorovici

Computers and Information Technology
(English Language)

Group C.E.N 1.1.b

1st Year

Introduction

A Bloom Filter can be seen as a probabilistic data structure similar to a set in which you can insert elements or search for one. It is very efficient in terms of time and space. Since it is probabilistic, when searching for an element the algorithm can only say if the element is surely not in the set or might be. The probability of failure depends on some variables: the expected maximum number of elements, the number of total bits necessary for representing the Bloom Filter as an array, the number of hash methods used. Hash methods usually receive as parameter the same single value (which is the element added or searched for) and return an integer which represents the index in the array which we will access and/or modify. The hashing algorithms should be uniformly distributed and ensure as less collisions as possible. One calls a colision the moment when all hashes return the same values for two different parameters.

For this project, the expected maximum number of elements is 10^5, the number of total bits is 10^6 and the number of hash methods used is 7. Only integer values of at most 8 digits are admitted. Each hash method used has the same structure, but different constants: an and operation with the value given, a xor operation and a modulo operation to ensure relatively uniform distribution and no accessing outside array bounds. The probability of failure in this case is about 1%.

Problem statement

Create a library for the basic utilities of a Bloom Filter: inserting an element and searching for an element.

Pseudocode

As mentioned earlier, we will use an array for representing the Bloom Filter and keeping track of the changes made. Also we will have a variable that holds count of the number of elements inserted so far. Keep in mind that multiple instances of the same value can be inserted.
We will use functions for inserting and searching in a Bloom Filter. We will also use some functions for utility, such as validating user's input.
Here are the important functions employed by the program:

```
bloomFilter_init(bloomFilter *BF)
1.      BF->nr_elements <- 0
2.      for i <- 1 to NR_BITS do
3.              BF->bloomVector[i] <- 0
```

```
bloomFilter_insert(bloomFilter *BF, int value)
1.      if (BF->nr_elements >= nrMax) do
2.              print ‘‘Error! Cannot add number... maximum set size is
    ‘‘ nrMax
3.      for each hash function do
4.              position <- get result of hash function for parameter
    value
5.              BF->bloomVector[position] <- 1
6.      BF->nr_elements <- BF->nr_elements + 1
```

```
bloomFilter_search(bloomFilter *BF, int value)
1.      for each hash function do
2.              position <- get result of hash function for parameter
    value
3.              if (BF->bloomVector[position] = 0) do
4.                      print ‘‘Element not found in Bloom Filter’’
5.      print ‘‘Element found in Bloom Filter’’
```

Application design

The library contains the headers ***bloomFilter.h, hashes.h, menuUtilities.h*** which have all the function prototypes to compute the required operations. These are all of them:
—void bloomFilter_init(bloomFilter *BF)
—void bloomFilter_insert(bloomFilter *BF, int value)
—void bloomFilter_search(bloomFilter *BF, int value)
—int hash1(int value)
—int hash2(int value)
—int hash3(int value)
—int hash4(int value)
—int hash5(int value)
—int hash6(int value)
—int hash7(int value)
—int checkNr(char *str)
—int buildNr(char *str)

The source file ***bloomFilter.c*** includes the implementations for functions in ***bloomFilter.h***:
—bloomFilter_init(bloomFilter *BF)
—bloomFilter_insert(bloomFilter *BF, int value)
—bloomFilter_search(bloomFilter *BF, int value).
Function bloomFilter_init(bloomFilter *BF) includes two steps, as follows:
1) Set the number of elements in the Bloom Filter to 0
2) Set all the values of the array to 0

Function bloomFilter_insert(bloomFilter *BF, int value) includes three steps, as follows:
1) Check if number of elements does not exceed the maximum amount possible
2) Send as parameter the value you want to insert to every hash function
3) Mark each resulted position from the hash function in the array as 1

Function bloomFilter_search(bloomFilter *BF, int value) includes four steps, as follows:
1) Send as parameter the value you want to search to every hash function
2) Check if any position resulted from the hash function is 0
3) If yes, print “Not found” and return. Else, continue.
4) If function reached the end, that means all positions are marked as 1, so print “Found”.

The bloomFilter structure is made of the following:
—unsigned short int bloomVector[NR_BITS] — which is our basic Bloom Filter representation
—unsigned int nr_elements — which holds count of the number of elements in the Bloom Filter at any point

The source file ***hashes.c*** includes the implementations of the hashing methods in ***hashes.h***:

Every hash method is implemented in the same manner, with the same structure, only different constants so that hashing results differ.
Each hash method consists of 3 main operations, in this order:
—And (bit-wise) operation— for which there is an array of random numbers: and[7] = {619067, 892701, 148840, 503544, 948857, 308270, 270728}
—Xor (bit-wise) operation— for which there is an array of random numbers: xor[7] = {2981, 17540, 877342, 796551, 11184810, 21453941, 778240}
—Modulo operation— for which there is an array of random prime numbers: prime[7] = {287233, 36277, 388099, 862991, 756253, 437219, 924197}

The source file ***menuUtilities.c*** includes the implementations of the methods in ***menuUtilities.h***:
—int checkNr(char *str)
—int buildNr(char *str)

Function checkNr(char *str) parses the string str and checks if there are only digits. If there are, it returns 1, else 0.

Function buildNr(char *str) converts the string str into an integer number which is then returned. It also accepts plus or minus sign before the digits.

The final source file, ***main.c*** itself, uses all of the functions discussed so far.
In the beginning, the user is presented the title of the project along with the name of the author. Then, the user will be presented the ***Main Menu***, which has four options:
—1) Initialize / Reset the Bloom Filter (set)— This is done at the start of the program, after the memory allocation, so there is no need to start with this option.
—2) Add an element to the current set— If chosen, it waits for the user's number to be typed.
—3) Search for an element in the current set— If chosen, it waits for the user's number to be typed.
—4) Exit program— If chosen, terminates the execution of the program, returning 0.
The ***Main Menu's*** input is built around the ***getch()*** function, which waits for a keypress, after which the program queries it and acts in conformity with the option chosen.

Source Code

```
//----------------------------bloomFilter.h------------------------------

/**
* @file bloomFilter.h
* @brief This header file contains the definition
* of a Bloom Filter with its basic utilities: Initialization, Insert,
    Search.
*
* @author Denis Ehorovici
*
* @date 06/15/2017
*/

#ifndef H_BLOOMFILTER
#define H_BLOOMFILTER

/// Maximum vector length of the bit vector for the Bloom Filter
#define NR_BITS 1000000

/// Structure definition for Bloom Filter
typedef struct bloomFilter {
       /// Unsigned short integer because we only need values of 0 and
           1 so we use as little memory as possible (around 2 bytes *
           NR_BITS)
       unsigned short int bloomVector[NR_BITS];
       /// Unsigned integer for positive number of elements in the
           Bloom Filter
       unsigned int nr_elements;
} bloomFilter;

/// Definition for initialization of Bloom Filter
void bloomFilter_init(bloomFilter *BF);

/// Definition for insertion in Bloom Filter
void bloomFilter_insert(bloomFilter *BF, int value);

/// Definition for searching in Bloom Filter
void bloomFilter_search(bloomFilter *BF, int value);

#endif // H_BLOOMFILTER
```

```
//----------------------------bloomFilter.c------------------------------

/**
* @file bloomFilter.c
* @brief This file contains all constant initializations
* and implementations for functions defined in the bloomFilter header.
*
* @author Denis Ehorovici
*
* @date 06/15/2017
*/

#include "bloomFilter.h"
#include "hashes.h"
#include <stdio.h>

/// Constant number representing maximum number of elements
const int nrMax = 100000;

/**
* This method will initialize / reset the Bloom Filter transmitted as
    parameter.
* @author Denis Ehorovici
* @param BF Pointer to bloomFilter structure
* @date 06/15/2017
*/
void bloomFilter_init(bloomFilter *BF) {
       /// Initialize / Reset the number of elements to 0
       BF->nr_elements = 0;

       /// Go through the whole array and set all values to 0
       int i;
       for (i = 0; i < NR_BITS; i++) {
              BF->bloomVector[i] = 0;
       }

       /// Signal success at the end of initialization
       printf("Set has been initialized successfully!\n");
}

/**
* This method will insert an element into the Bloom Filter transmitted
    as parameter.
* @author Denis Ehorovici
* @param BF Pointer to bloomFilter structure
* @param value The value to be inserted
* @date 06/15/2017
*/
void bloomFilter_insert(bloomFilter *BF, int value) {
       /// Check if the Bloom Filter is already at maximum capacity and
           display an error message if so
       if (BF->nr_elements >= nrMax) {
              printf("Error! Can’t add number... (maximum set size is
                  %d)\n", nrMax);
              return;
```

```
	}

	/// Go through each hash function and mark the position each one
	    returns as 1 in the Bloom Filter's array.
	int position = hash1(value);
	BF->bloomVector[position] = 1;

	position = hash2(value);
	BF->bloomVector[position] = 1;

	position = hash3(value);
	BF->bloomVector[position] = 1;

	position = hash4(value);
	BF->bloomVector[position] = 1;

	position = hash5(value);
	BF->bloomVector[position] = 1;

	position = hash6(value);
	BF->bloomVector[position] = 1;

	position = hash7(value);
	BF->bloomVector[position] = 1;

	/// Increment the number of elements by 1
	BF->nr_elements++;

	/// Signal success at the end of insertion
	printf("Value inserted!\n");
}

/**
* This method will search for an element in the Bloom Filter transmitted
    as parameter.
* @author Denis Ehorovici
* @param BF Pointer to bloomFilter structure
* @param value The value to be searched for
* @date 06/15/2017
*/
void bloomFilter_search(bloomFilter *BF, int value) {
	/// Go through each hash function and check if the position each
	    one returns is 0 in the Bloom Filter's array. If yes,
	    display the "not found" message and exit function.
	int position = hash1(value);
	if (BF->bloomVector[position] == 0) {
		printf("Value searched is surely not in the set!\n");
		return;
	}
```

```
        position = hash2(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        position = hash3(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        position = hash4(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        position = hash5(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        position = hash6(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        position = hash7(value);
        if (BF->bloomVector[position] == 0) {
                printf("Value searched is surely not in the set!\n");
                return;
        }

        /// If function reached this far, then the value might be in the
            Bloom Filter! Display the "found" message and exit program.
        printf("Value searched might be in the set!\n");
}
```

```
//-----------------------------hashes.h-------------------------------

/**
* @file hashes.h
* @brief This header file contains all hash methods
* definitions.
*
* @author Denis Ehorovici
```

```c
*
* @date 06/15/2017
*/

#ifndef H_HASHES
#define H_HASHES

/// Definition for each hash method
int hash1(int value);
int hash2(int value);
int hash3(int value);
int hash4(int value);
int hash5(int value);
int hash6(int value);
int hash7(int value);

#endif // H_HASHES
```

```c
//-----------------------------hashes.c-------------------------------

/**
* @file hashes.c
* @brief This file contains all the random constant values
* required for the implementations of the hashes defined
* in the hashes header.
*
* @author Denis Ehorovici
*
* @date 06/15/2017
*/

#include "hashes.h"

/// Maximum array length from bloomFilter.h (NR_BITS)
const int totalBits = 1000000;

/// Array of random prime numbers used for uniform distribution and
    limiting in each hash method
const int prime[7] = {287233, 36277, 388099, 862991, 756253, 437219,
    924197};

/// Array of random numbers used for and bit operation in each hash
    method
const int and[7] = {619067, 892701, 148840, 503544, 948857, 308270,
    270728};

/// Array of random numbers used for xor bit operation in each hash
    method
const int xor[7] = {2981, 17540, 877342, 796551, 11184810, 21453941,
    778240};

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash1(int value) {
       /// And operation for possible alteration of value’s bits
       value = (value & and[0]);

       /// Xor operation for further possible alteration / randomization
       value = (value ^ xor[0]);

       /// Final operation for making sure the hash method has uniform
           distribution and value won’t exceed maximum array length of
           Bloom Filter
       value = (value % prime[0]) + (totalBits - prime[0]);

       /// Return the final hashing result
       return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash2(int value) {
       /// And operation for possible alteration of value’s bits
       value = (value & and[1]);

       /// Xor operation for further possible alteration / randomization
       value = (value ^ xor[1]);

       /// Final operation for making sure the hash method has uniform
           distribution and value won’t exceed maximum array length of
           Bloom Filter
       value = (value % prime[1]) + (totalBits - prime[1]);

       /// Return the final hashing result
       return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
```

```
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash3(int value) {
       /// And operation for possible alteration of value's bits
       value = (value & and[2]);

       /// Xor operation for further possible alteration / randomization
       value = (value ^ xor[2]);

       /// Final operation for making sure the hash method has uniform
           distribution and value won't exceed maximum array length of
           Bloom Filter
       value = (value % prime[2]) + (totalBits - prime[2]);

       /// Return the final hashing result
       return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash4(int value) {
       /// And operation for possible alteration of value's bits
       value = (value & and[3]);

       /// Xor operation for further possible alteration / randomization
       value = (value ^ xor[3]);

       /// Final operation for making sure the hash method has uniform
           distribution and value won't exceed maximum array length of
           Bloom Filter
       value = (value % prime[3]) + (totalBits - prime[3]);

       /// Return the final hashing result
       return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
```

```
*/
int hash5(int value) {
        /// And operation for possible alteration of value's bits
        value = (value & and[4]);

        /// Xor operation for further possible alteration / randomization
        value = (value ^ xor[4]);

        /// Final operation for making sure the hash method has uniform
            distribution and value won't exceed maximum array length of
            Bloom Filter
        value = (value % prime[4]) + (totalBits - prime[4]);

        /// Return the final hashing result
        return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash6(int value) {
        /// And operation for possible alteration of value's bits
        value = (value & and[5]);

        /// Xor operation for further possible alteration / randomization
        value = (value ^ xor[5]);

        /// Final operation for making sure the hash method has uniform
            distribution and value won't exceed maximum array length of
            Bloom Filter
        value = (value % prime[5]) + (totalBits - prime[5]);

        /// Return the final hashing result
        return value;
}

/**
* This method will return a hash between 0 and 1000000 to the parameter
    value.
* @author Denis Ehorovici
* @param value The value to be hashed
* @date 06/15/2017
*/
int hash7(int value) {
        /// And operation for possible alteration of value's bits
        value = (value & and[6]);
        /// Xor operation for further possible alteration / randomization
        value = (value ^ xor[6]);

        /// Final operation for making sure the hash method has uniform
            distribution and value won't exceed maximum array length of
            Bloom Filter
        value = (value % prime[6]) + (totalBits - prime[6]);

        /// Return the final hashing result
        return value;
}
```

```
//-----------------------------menuUtilities.h------------------------------

/**
* @file menuUtilities.h
* @brief This header file contains the definition
* of the methods used in main menu to check and validate user's input
*
* @author Denis Ehorovici
*
* @date 06/16/2017
*/

#ifndef H_MENUUTILITIES
#define H_MENUUTILITIES

/// Definitions of methods checkNr and buildNr
int checkNr(char *str);
int buildNr(char *str);

#endif // H_MENUUTILITIES
```

```
//-----------------------------menuUtilities.c------------------------------

/**
* @file bloomFilter.c
* @brief This file contains all implementations for functions
* defined in the menuUtilities header.
*
* @author Denis Ehorovici
*
* @date 06/16/2017
*/

#include "menuUtilities.h"
```

```
/**
* This method will check if a string is an integer number and returns 1
    if check passed, else returns 0.
* @author Denis Ehorovici
* @param str Pointer to the string
* @date 06/15/2017
*/
int checkNr(char *str) {
       /// Ignore if string starts with plus or minus sign
       if (*str == '-' || *str == '+') {
              str++;
       }

       /// Check if string is only made of plus or minus sign, in which
           case the string is invalid
       if (*str == '\0') {
              return 0;
       }

       /// Parse the whole string and check if there exists any
           character that is not a digit and return 0 if so, else
           return 1
       while (*str != '\0') {
              if ('0' > *str || *str > '9') {
                     return 0;
              }
              str++;
       }
       return 1;
}

/**
* This method will convert the string str to an integer and return it.
* @author Denis Ehorovici
* @param str Pointer to the string
* @date 06/15/2017
*/
int buildNr(char *str) {
       /// Maintain a reminder for the sign of our value (default = 0
           which means positive number)
       int negative = 0;

       /// Variable for number built, defaults to 0
       int value = 0;

       /// Check sign and change "negative" variable if necessary
       if (*str == '-') {
              negative = 1;
              str++;
       }
```

```
        else if (*str == '+') {
                str++;
        }

        /// Parse the string and build the number
        while (*str != '\0') {
                value = value * 10 + (*str) - '0';
                str++;
        }

        /// Check if sign needs to be changed and change if necessary
        if (negative == 1) {
                value = -value;
        }

        /// Return number built
        return value;
}
```

```
//--------------------------------main.c--------------------------------


/**
* @file main.c
* @brief This file contains all the code regarding
* the Main Menu and memory allocation for the Bloom Filter.
*
* @author Denis Ehorovici
*
* @date 06/15/2017
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bloomFilter.h"
#include "menuUtilities.h"

/**
* This method is the main function of the program and is responsible for
    the Main Menu.
* @author Denis Ehorovici
* @date 06/15/2017
*/
int main() {
        /// Introduction
        printf("Project made by student Denis Ehorovici from University
            of Craiova, Faculty of Automatics and Electronics\n\n");
        printf("Title of the project: Bloom Filter\n\n");
        printf("Press any key for Main Menu: ...");
        getch();

        /// Allocate memory to the Bloom Filter that will be used along
            the program
        bloomFilter *BF = (bloomFilter*) malloc(sizeof(bloomFilter));

        /// Initialize the Bloom Filter
        bloomFilter_init(BF);
        while (1) {
               /// Clear screen for new menu phase
               system("cls");

               /// Show the Main Menu
               printf("Main Menu:\n\n");
               printf("Press the corresponding number for your
                   option:\n");
               printf("1. Initialize / Reset your bloom filter (set)\n");
               printf("2. Add an element to the current set\n");
               printf("3. Search for an element in the current set\n");
               printf("4. Exit program\n");

               /// Variable "key" holds user's input (single keyboard
                   press)
               int key = getch();

               /// Initialization option
               if (key == (int)'1') {
                      system("cls");

                      printf("Initializing...\n");
                      bloomFilter_init(BF);
                      printf("\nPress any key to return to Main
                          Menu...\n");
                      getch();
               }
               /// Inserting option
               else if (key == (int)'2') {
                      system("cls");

                      printf("Inserting value...\n");
                      printf("Insert an integer value of no more than 8
                          digits (7 if sign is typed before): ");

                      char str[1005];
                      gets(str);
                      while (strlen(str) < 1 || strlen(str) > 8 ||
                          checkNr(str) == 0) {
                             printf("Value has more than 8 digits or is
                                 not an integer, please try again: ");
```

```
                                gets(str);
                        }

                        int value = buildNr(str);

                        bloomFilter_insert(BF, value);
                        printf("\nPress any key to return to Main
                            Menu...\n");
                        getch();
                }
                /// Search option
                else if (key == (int)'3') {
                        system("cls");

                        printf("Searching value...\n");
                        printf("Search an integer value of no more than 8
                            digits (7 if sign is typed before): ");

                        char str[1005];
                        gets(str);
                        while (strlen(str) < 1 || strlen(str) > 8 ||
                            checkNr(str) == 0) {
                                printf("Value has more than 8 digits or is
                                    not an integer, please try again: ");
                                gets(str);
                        }

                        int value = buildNr(str);


                        bloomFilter_search(BF, value);
                        printf("\nPress any key to return to Main
                            Menu...\n");
                        getch();
                }
                /// Exit program option
                else if (key == (int)'4') {
                        break;
                }
                /// Invalid option, get input again
                else {
                        getch();
                }
        }

        /// End program
        return 0;
}
```

Experiments and results

```
Initalizing...

Adding element... 10
Output: Value inserted!

Adding element... asdf
Output: Value has more than 8 digits or is not an integer, please try
    again!
Adding element... -1234
Output: Value inserted!

Searching element... 10
Output: Value searched might be in the set!
Searching element... -1234
Output: Value searched might be in the set!
Searching element... -1
Output: Value searched is surely not in the set!
Searching element... -asdf
Output: Value has more than 8 digits or is not an integer, please try
    again!

Adding element... 123412341234
Output: Value has more than 8 digits or is not an integer, please try
    again!

Searching element... 123412341234
Output: Value has more than 8 digits or is not an integer, please try
    again!
```

Conclusions

Working on this project, I have learned a lot about bloom filters, their utility in big projects such as Chromium. One of the better uses of this probabilistic data structure is to get rid of extra data to be checked, for example: we have a bunch of mails and querying the bloom filter can tell you if one mail is surely spam or might not be. If it is spam, we already delete it, else we have to perform a deeper check on the mail with other methods.

References

1)https://en.wikipedia.org/wiki/Bloom_filter

2)https://llimllib.github.io/bloomfilter-tutorial/

3)http://www.cplusplus.com/