CS358: Applied Algorithms

Assignment 3: Cuckoo Filters and Streaming (due 10/16/2025)

Instructor: Sam McCauley

Instructions

Code can be submitted by committing and pushing them to gitlab. I strongly suggest that you access `evolene.cs.williams.edu` through a web browser to make sure everything was uploaded as you expected. You may collaborate on code with your classmates, as well as the instructor and TA, but you may not use any LLM assistance. Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem 1 (Filters) Description

Let's call a word a "bilingual palindrome" if the reverse of that word is a word in another language. For example, "mures" is a bilingual palindrome, because "mures" is a word in French, and "serum" is a word in English. A palindrome may not be a bilingual palindrome: for example, "racecar" is not a bilingual palindrome unless "racecar" is a word in another language. Given a list of words in several languages, we can efficiently find all the bilingual palindromes by storing all the words in a hash table (along with what languages they appear in), and then going through each word and looking up its reverse in the hash table. If the reverse appears in a language different from the original word, then each is a bilingual palindrome. This assignment asks you to implement a cuckoo filter to speed up this lookup process. Before checking the whole hash table for the reverse, we first check the cuckoo filter. We only check the hash table if the cuckoo filter says that the reverse may be present. Even though we are dealing relatively small amounts of data (the original dictionaries have total size less than 10MB), a cuckoo filter speeds up the implementation by a factor of approximately 4. This effect would become much more significant for larger datasets.

The only coding requirement for this assignment is to implement a cuckoo filter. The hash table, searching, output, etc., are all already implemented in `test_filters.c`. You may change `test_filters.c` if you wish, but you'll only be graded on your cuckoo filter implementation.

The full problem description is given here to help with your understanding.

Input: `test_filter.out` is given five arguments. The first three are strings representing word lists. The fourth is an integer representing the total number of (unique) words across these lists. The fifth is an integer representing the number of bilingual palindromes in these lists (that is to say, the fifth number is the desired answer).

Each word list is assumed to consist of a sequence of words, one on each line. Blank lines are ignored. The words are assumed to not have punctuation.

Note that these word lists are stored in ASCII format. In particular, this means that accented characters[1] were (automatically and likely poorly) transliterated into English equivalents—so people familiar with some of these languages may notice that things are slightly off.[2]

The following command runs your program with the correct number of inserted words and expected answer:

```
./test_filter.out english.txt french.txt german.txt 916641 1977
```

You may also test your command with the much shorter dictionaryShort.txt:

```
./test_filter.out dictionaryShort.txt dictionaryShort.txt dictionaryShort.txt 48 12
```

Output: The testing program automatically checks if the number of bilingual palindromes found matches the desired number. In this assignment, the program automatically gives some extra output that may be helpful. Further output can be toggled using `#define` statements: set `VERBOSE` to 1 to output the bilingual palindromes that are found, or set `CHECK_CORRECTNESS` to 1 to check if the filter has any false negatives.[3]

Filter functions: I have given you a starting point for how to use the filter—the testing program already interfaces with the following filter functions. I've also given the outline of a `struct` which I found useful to store some of the filter metadata; it can be found in `filter.h`. It's quite possible that you will change this struct, or you may not even want to use it at all. You may also edit the functions any way you wish—i.e. changing the type of the arguments or how many there are (provided that you are still implementing cuckoo filter functionality).

To be clear, you can get a perfect grade on the coding portion of this assignment while only editing the inside of the functions (denoted by comments) in `filter.c`.

You may notice that there is a (global) array at the beginning of `filter.c`. Its entries are random. The intention is to use this array as the hash h which you apply to the fingerprint in partial-key cuckoo hashing.

Here is a list of the functions and how they are used:

```
void filter_instantiate(Filter* filter, int numWords)
```

This function is called before any other calls to the filter. You can think of it like a constructor. It should set constants and allocate memory. `filter` is a pointer to a struct that I found useful (you can change this to not use a struct if you wish); `numWords` is a guarantee on how many elements will ever be inserted into the cuckoo filter. (In other words, `numWords` is n.)

[1] Characters like é or ü or ß; really anything other than A–Z.

[2] I put a decent amount of effort into trying to get around this but C really really does not play well with non-ASCII characters.

[3] If this flag is set, the program will check the table regardless of the filter's output—this will give good information for checking correctness, but while this flag is on the filter will not speed up the program.

```
void filter_insert(char* word, int length, Filter* filter)
```

This function inserts a new word (given by `word`) into the filter. `length` is the length of the word, and `filter` is a pointer to the filter we want to insert to. After `word` is inserted, if we call `filter_lookup` on `word` and `filter` it should always return 1.[4]

`test_filter.c` will insert each *unique*[5] word in each word list into the filter by calling this function. These inserts will all occur before any call to `filter_lookup`.

```
int filter_lookup(char* word, int length, Filter* filter)
```

This function looks up a word (given by `word`) in the filter—it is equivalent to the "is $q \in S$?" query discussed in class. `length` is the length of the word, and `filter` is a pointer to the filter we want to insert to. This function returns 1 to represent the filter saying "$q \in S$", and 0 to represent the filter saying "$q \notin S$". Note that, in the starter code, this function always returns 1, thus satisfying the No False Negatives guarantee.

`test_filter.c` will call this function before querying the hash table for each reversed word; it will only query the hash table if this function returns 1.

PARAMETERS: Your cuckoo filter should implement the following parameters. Each fingerprint should be of length 8 bits. There should be 5%–10% space overhead; that is to say, the number of total slots should be around $1.10 * (\texttt{num\_words})$.[6] You should use partial-key cuckoo hashing, so you should have two hash functions h_1 and $h_2(x) = h_1(x) \wedge h(f(x))$. Each hash entry should consist of 4 slots, where each slot is large enough to store a fingerprint.[7]

On an insert, your filter should exit if the number of "cuckoo" iterations exceeds `max_iter`. I would suggest setting `max_iter`$= 4\log_2(\texttt{num\_words})$. If this causes frequent failures, feel free to make this number somewhat higher—but if you are running into consistent issues even with a large `max_iter`, there is likely an issue with your filter.

Implementation 1. Implement a cuckoo filter as described above.

Filter Analysis

Problem 1. Give an upper bound on the false positive rate ε of your filter. That is to say, analyze the (theoretical) false positive probability with bins of size 4, fingerprints of length 8, 1.05 fingerprint slots per element, and 2 hash functions.

Hint: Most of the work for this question was done in the lecture.

Solution. □

[4] You can test this with `CHECK_CORRECTNESS` in `test_filter.c`.

[5] You do not need to worry about duplicates; they will be removed before this function is called.

[6] 1.05 works fine for me, but it's OK if a little more space helps make the filter work.

[7] This conveniently means that each bin is of size 32 bits.

Problem 2. Let's say I take a Bloom filter (not a cuckoo filter!) for a set S_1 using a bit array A_1 and hash functions $h_1, \ldots h_k$, and a Bloom filter for a set S_2 using a bit array A_2 and the same hash functions $h_1, \ldots h_k$.

I create a bit array T using the bitwise OR of A_1 and A_2. (That is to say, I get a bit array T where $T[i] = A_1[i] \mid B_1[i]$.) Then T along with $h_1, \ldots, h_k$ is a Bloom filter—it is a bit array with an associated hash function.

(a) If I query T for an element q, and its bits are all set to 1, (i.e. if $T[h_i(q)] = 1$ for all i), what can I say about q? (For example: can we say that $q \in S_1$ for sure? Or that q is unlikely to be in S_2?)

(b) If I query T for an element q, and one of its bits is 0 (i.e. $T[h_i(q)] = 0$ for some i), what can I say about q?

Solution. □

Streaming: Problem Description

In this problem, we will be analyzing a very long novel: "In Search of Lost Time," by Marcel Proust.[8] This novel contains about a million words (including duplicates), and the text file given for this assignment is about 7MB. Nonetheless, we will be using very small streaming data structures to analyze this file with just a single pass over the data—the first using a handful of kilobytes of space, the second using just 32 bytes.

In this assignment, you will be building two data structures.

First, you will build a Count-Min Sketch data structure. All words in "In Search of Lost Time" will be inserted into the Count-Min Sketch. At the end, your data structure will be queried with some of the most common words in the novel: how many times does this word appear? The testing program will compare your output to the actual count of each word; your data structure should always overestimate the count, but give reasonably similar values.

Second, you will build a HyperLogLog data structure. Again, all words in "In Search of Lost Time" will be inserted into it. At the end, your data structure will be queried to find out approximately how many unique words occurred in the novel. HyperLogLog uses an incredibly small amount of space, so it is likely that your data structure will have some error. However, it should usually be reasonably close to the correct value.

INPUT: `test_streaming.out` is given three arguments. The first is a text document in ASCII format.[9] The second is a text document, where each line contains a word from the first text document, followed by a space, followed by the number of times that word appears in the first document. The final argument is an integer denoting the number of unique words in the original text document.

[8] This novel is, I understand, very popular and well-regarded in terms of literary content. However, it was chosen for this class mostly because it's long and its copyright has expired.

[9] As with last time, this means there are no accented characters.

To run your program on "In Search of Lost Time," you would use the following input:

```
./test_streaming.out proust.txt words.txt 36372
```

The following input may be useful for testing:

```
./test_streaming.out proustShort.txt wordsShort.txt 125
```

OUTPUT: This assignment is unique in this course in that a single answer is not usually marked as correct or incorrect.[10] Instead, the testing program will output, for each word in the second text file, the actual number of occurrences of the word in the text compared to the number output by your Count-Min Sketch. Furthermore, the testing program will output the number of unique words predicted by your HyperLogLog data structure, compared to the actual number of unique words.

INTERPRETING THE OUTPUT: For the large output, the CMS should generally answer most word queries within 1000 of the correct value. Almost all CMS answers should be within 1500 of the correct value. The HLL overall estimation of the number of words should almost always be between 25000 and 50000.

You should use several different seeds to check that your answers satisfy these bounds.

FUNCTIONS: This assignment is, broadly, structured much like the filters. The functions for both data structures already exist, and you must fill them in. The code in `test_streaming.c` will perform the above tests using the functions you provide.

`cms.c` and `cms.h` contain the code for the Count-Min Sketch data structure. `hll.c` and `hll.h` contain the code for the HyperLogLog data structure.
Here is a list of the functions and how they are used:

```
void cms_instantiate(Cms* cms)
```

This function is called before any other calls to the cms functions. You can think of it like a constructor. It should set constants and allocate memory. `cms` is a pointer to a struct that I found useful (you can change this to not use a struct if you wish). You do not need to edit this function if you don't want to; the version in the assignment is the version I used.

```
void cms_insert(char* word, int length, Cms* cms)
```

This function inserts a new word (given by `word`) into the filter. `length` is the length of the word, and `cms` is a pointer to the Count-Min Sketch we want to insert to.

`test_streaming.c` will insert each word in the first document into the Count-Min Sketch by calling this function. These inserts will all occur before any call to `filter_lookup`.

```
int cms_lookup(char* word, int length, Cms* cms)
```

[10] This choice is due to two concerns. First, we're using randomness, so some error is to be expected. (One could run the program multiple times and perform statistical tests, but that's very much contrary to the spirit of these structures.) Second, these structures are fairly inconsistent: for example, a cuckoo filter will almost always have approximately the same false positive rate on a large dataset; a CMS or HLL may not.

This function looks up a word (given by `word`) in the sketch pointed to by `cms`. It returns an estimate of how often `word` (which has length `length`) occurs in the first document.

```
void hll_instantiate(Hll* hll)
```

This function is called before any other calls to the hll functions. You can think of it like a constructor. It should set constants and allocate memory. `hll` is a pointer to a struct that I found useful[11] (you can change this to not use a struct if you wish). You do not need to edit this function if you don't want to; the version in the assignment is the version I used.

```
void hll_insert(char* word, int length, Hll* hll)
```

This function inserts a new word (given by `word`) into the HyperLogLog data structure `hll`. `length` is the length of the word, and `hll` is a pointer to the structure we want to insert to.

`test_streaming.c` will insert each word in the first document into the HyperLogLog data structure by calling this function.

```
int hll_estimate(Hll* hll)
```

This function asks for an estimate of how many unique words have been inserted into `hll`.

Count-Min Sketch Parameters: Your CMS should have 4 rows, each of 300 entries. Each entry should be of 32 bits.[12]

HyperLogLog Parameters: Your HyperLogLog data structure should keep track of 32 counters, each of length 8 bits. For 32 counters, the bias constant is .697. (The bias constants are included in `hll.h`.)

> **Implementation 2.** Implement a Count-Min Sketch and a HyperLogLog counter, each as described above.

CMS Analysis

Problems 3 and 4 both use the following setup.

Consider a situation where I have a stream of 1,001,000 elements. 1,000,000 of the elements $a_1, \ldots a_{1000000}$ only appear once, but one element, q, appears 1000 times throughout the stream.

For each of the following two problems (Problems 3 and 4), give the appropriate parameters[13] for a correct Count-Min Sketch with the smallest possible space. Be sure to answer:

[11] This is a bit wasteful, unlike the CMS and cuckoo filter. You could easily have the seed as a global constant and just pass around the table of counters rather than storing this struct.

[12] This is wasteful! Unfortunately, 16 bit integers are JUST small enough to barely overflow on this data. Using 18 or 19 bit entries would almost certainly be ideal, but would be a pain to maintain.

[13] Here I am asking about the parameters you should use for your *code*: number table entries, number of rows, etc. Determining ε and δ is likely an important part of your answer, but it is not the final solution.

- How many rows should I have? (`numRepetitions`)
- How many entries should I have in each row? (`numEntries`)
- How large should each entry be in bits?[14] (The size of each entry in `table`, in bits)

Problem 3. After inserting all stream elements into my CMS using the parameters above, I query an element a_i, and I receive an answer o_i, and I query q, and receive an answer o_q.

How should I set the parameters of my Count-Min Sketch so that at least 90% of the time, $o_i \leq o_q$? That is to say, how do I set my parameters so that 90% of the time, the CMS accurately returns that q was more common than a_i?

You do not need to give an optimized answer; giving parameters that work is OK even if slightly different parameters are more efficient. But, you should explain how you got your answer, and why it attains the required performance.

Hint: One way to do this is to come up with constants c_q and c_i such that for your parameters you can show that:

1. $c_i \leq c_q$, and
2. 90% of the time, $o_i \leq c_i$ and $c_q \leq o_q$.

Solution. □

Problem 4. After inserting all stream elements into my CMS, I query all elements $a_1, \ldots a_{1000000}$ to obtain answers $o_1, \ldots o_{1000000}$, as well as querying q to obtain an answer o_q.

How should I set the parameters of my Count-Min Sketch so that at least 90% of the time, $\max_i o_i \leq o_q$? That is to say, how do I set my parameters so that 90% of the time, the CMS accurately returns that q was the most common out of all elements queried?

Hint: You probably want to use the union bound here, combining it with your answer to Problem 3.

Solution. □

HyperLogLog Analysis

Problem 5. Let's say I create a HyperLogLog structure H_1 for a stream $a_1, \ldots a_n$ and a second HyperLogLog structure H_2 for a stream $b_1, \ldots b_n$. Assume that all a_i and b_i are in the same universe U, and assume that identical parameters and hash functions are used to create H_1 and H_2.

Describe how to use H_1 and H_2 to create a new HyperLogLog structure H_3 that

[14] Please give an exact answer, even if `C` does not support variables of this size.

> can estimate the number of unique items in the concatenation of the two streams: $a_1, \ldots a_n, b_1, \ldots, b_n$. (You should build H_3 using only H_1 and H_2, without seeing the stream again.)

Solution. □