CS358: Applied Algorithms

Assignment 3: Cuckoo Filters (due 10/13/2021)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN."

Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem 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 2! 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.c. You may change `test.c` if you wish, but you'll only be graded on your cuckoo filter implementation.

There is no leaderboard this week—improving cuckoo filter performance is an interesting research area, but doesn't lend itself well to the kind of optimizations we've seen in this course. There won't be automatic testing either—as mentioned below, `test.c` gives you tools to check correctness.

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

INPUT: `test.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.out english.txt french.txt german.txt 916641 1977
```

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

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.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.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 $h_2(x) = h_1(x) \hat{} h(f(x))$. 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 higher.

Questions

Code (50 points). Implement a cuckoo filter as described above. You do not need to describe your implementation—but I left some space below in case there's something that you think would be helpful to state.

Solution.

Problem 1 (10 points)**.** 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.

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

Solution. We didn't get a chance to go over this analysis in lecture, so I'll include a correct answer here. Free 10 points for all!

When we perform a query q, we will look in all 8 slots corresponding to $h_1(q)$ and $h_2(q) = h_1(q)\hat{}h(f(q))$.

Consider one such slot s. This slot may store some fingerprint $f(x)$ for some $x \in S$. If $q \notin S$, then $q \neq x$, so if we assume that the output of MurmurHash is uniform random then the probability that $f(q) = f_s$ is $1/255$.

Query q is a false positive if its fingerprint matches any of the 8 slots. Using the union bound, the probability that q is a false positive can be upper bounded by $8/255$.

Bucket Sort

We've seen a few instances of problems where we need to sort data that's fairly randomly distributed. In this problem, you'll analyze a sorting algorithm that performs very well on random data.

This algorithm is called Bucket Sort.[8] Bucket sort works as follows on an array A of n 64-bit integers. Let's assume that n is a power of 2, so $n = 2^k$ for some integer $k \leq 64$.

1. Create 2^k empty integer[9] doubly-linked lists
2. For i from 0 to $n - 1$:
   (a) let $b = A[i]$ `>>` $(64 - k)$, where `>>` denotes right shift as in `C`. In other words, let b store the k most significant bits of $A[i]$.
   (b) Add $A[i]$ to the bth linked list.
3. For ℓ from 0 to $2^k - 1$, sort linked list ℓ using insertion sort.[10]

Problem 2 (10 points)**.** Analyze the worst-case running time of this algorithm on an array of n elements. (Note that this question does not use randomness.)

Solution.

Random Analysis Now, in the next few questions, we'll show that bucket sort is $O(n)$ if the elements of A are each chosen as random 64 bit integers. (So from now on, assume that elements of A are random.)

In the following questions, let $X_{i,j,\ell}$ be a random variable such that $X_{i,j,\ell} = 1$ if $A[i]$ and $A[j]$ are both stored in linked list ℓ by the bucket sort algorithm, and $X_{i,j,\ell} = 0$ otherwise.

[8] This is a well-known algorithm, and its analysis can be found online. But, I'd ask that you don't look it up. Furthermore, the analysis used here is a bit different from how it's generally done.

[9] I.e. each linked list node should be able to hold an integer.

[10] You can insertion sort doubly-linked lists! Insertion sorting a linked list of length k takes $O(k^2)$ time.

Problem 3 (5 points)**.** Show that the time required for bucket sort can be bounded by

$$O\left(n + \sum_{\ell=0}^{2^k-1} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_{i,j,\ell}\right) \tag{1}$$

Hint: Write the time to insertion sort in terms of the length of each list. Then, rewrite the time required to sort each list using a sum of $X_{i,j,\ell}$.

Solution.

Problem 4 (5 points)**.** Show that for any i, j, ℓ with $i \neq j$

$$\Pr\left[X_{i,j,\ell} = 1\right] \leq O(1/n^2). \tag{2}$$

Solution.

Problem 5 (10 points)**.** Combine Equations 1 and 2 to show that the expected running time of bucket sort on a random A is $O(n)$.

Hint: Use linearity of expectation.

Hint2: You'll probably want to handle all variables $X_{i,j,\ell}$ that have $i = j$ separately.

Solution.

Bloom Filters

Problem 6 (10 points)**.** Let's say I take a Bloom 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 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 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?

(c) (**Extra credit: 10 pts**) Assume that $k = \log_2(1/\varepsilon)$ and $|A_1| = |A_2| = 1.44n \log_2(1/\varepsilon)$ (where $n = |S_1| = |S_2|$). As we saw in class, we can analyze these filters with the (not-quite-accurate) assumption that each bit of A_1 and A_2 is 1 with probability 1/2, and that these events are independent. Under this assumption, what is the false positive rate of T?

Solution.