CS358: Applied Algorithms

Mini-Midterm 2: Robin Hood Hashing (due 11/3/21)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github, as with assignments. 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." The mini-midterm will not have a leaderboard or any automated testing (beyond the testing available in the starter code).

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

This is a mini-midterm (as on the syllabus). This means that **all work must be done alone.** Please do not discuss solutions with any other students, even at a high level. You should not look up answers, hints, or even code libraries on the internet. This midterm was designed to be completed with no external resources, beyond those explicitly linked in the midterm. (Class resources, such as slides, notes, and your previous assignment submissions, are of course acceptable, as are basic resources such as looking up debugging information.)

You may assume the running times of any algorithms and data structures you learned about in CSCI 256 or this class. For example, you may assume a dictionary implemented with a hash table that requires $O(1)$ lookup time and $O(n)$ space,[1] or a sort algorithm taking $O(n \log n)$ time and $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses.

A comment on grading: remember that we have four of these over the course of the semester, and (while important) this midterm is only worth 20% of your final grade.

[1]As we'll see in the next part of the course, $O(1)$ is only expected, but for simplicity you may assume it is worst case.

1 Robin Hood Hashing

For this mini-midterm you will build a hash table—this is likely a fairly familiar data structure in terms of its most basic workings. However, we'll be implementing a number of very popular optimizations to improve hash table efficiency, in terms of both time and space.

The data we're using is moderately sized, so it's very much possible to get a fast (or possibly much faster) implementation without some of these optimizations. However, I'm asking you to implement all for the sake of the assignment.

Assignment Parameters

THE DATA: For this assignment we'll be using DNA data. In fact, we'll be using a publicly-available sequence of (an example of) the first chromosome of human DNA; this is available in `chromosome1.txt`.

This data has been fairly thoroughly cleaned. All strings you'll be dealing with will consist of exactly 80 characters, where every character is `'A'`, `'C'`, `'G'`, or `'T'`; there may also be some space characters `' '` (space characters were appended to any line with fewer than 80 characters).

In addition to `chromosome1.txt`, there are two files that we'll use for queries: `posQueries.txt` is a permuted version of `chromosome1.txt`, and `negQueries.txt` consists of 80-character strings where each character (from `'A'`, `'C'`, `'G'`, or `'T'`) is chosen with probability 1/4.

As in Assignment 5, all of this data is stored as a set of uncompressed text files.[2] These files are each too large for github, so they have been slightly compressed into zip files. Each should be unzipped before being used.

Finally, there are also two smaller files for help with testing: `shortChromosome1.txt` and `shortPosQueries.txt`.

FUNCTIONS: You will be implementing five functions in total. (As usual, you are encouraged to create your own helper functions, but the following are the ones that will be called by `test.c` directly.)

- `void hashtable_instantiate(int numEntries, Hashtable* theHashTable)`: this function is to set up your hash table. This function will be called once before any other function is called. `numEntries` is an upper bound on the number of `hashtable_insert` calls made for the hash table.

- `void hashtable_insert(char* sequence, Hashtable* theHashTable)`: the insertion function. The 80-character string pointed to by `sequence` should be stored in `theHashTable`.

- `int hashtable_lookup(char* sequence, Hashtable* theHashTable)`: the lookup function. Should return 1 if the 80-character string pointed to by `sequence` is stored in `theHashTable`, and 0 otherwise.

[2]That's not a particularly good way to store these files! It sure would be nice if a motivated group of Williams students could implement an effective way to make them smaller in a later assignment...

- `int manual_compare(char* str1, char* str2)`: a function to compare two strings of at most 80 characters. This function should return 1 if the strings are different, and 0 if the strings are the same.[3]
- `simd_compare(char* str1, char* str2)`: in Problem 1, you will create a function to compare two strings using SIMD instructions. See Problem 1 for details.

Testing Your Code: The test code takes four arguments: a file of items to insert, a file of queries (all positive—i.e. the items in this file must all be in the previous file), and a file of all negative queries, followed by the number of items to use from each file.

You can test your code by running

```
./test.out chromosome1.txt posQueries.txt negQueries.txt 2881018
```

A smaller test can be run with

```
./test.out shortChromosome1.txt shortPosQueries.txt negQueries.txt 5
```

If there are any errors, the testing file will output a line beginning with "Incorrect lookup". If you see no such line (just timing data about each operation), then there were no errors found on that run.

Problem Description

Likely the most popular method to resolve collisions is with linear probing. In linear probing, if a slot is full, the algorithm finds a subsequent slot in which to store the element.

In this lab, you will be implementing hashing with linear probing, with the following optimizations. Make sure you remember to implement each in your code.

1. **A circular hash table:** in linear probing, what happens if items "fall off" the end of the table?[4] One classic option is to make the hash table circular: if the last slot in the table is full, the next slot to be checked is the first slot (i.e. slot 0).

2. **Robin hood hashing:** Robin hood hashing is a way to ensure that items are not stored too far away from their intended slot. For each item in our hash slot, we store the *distance* of the item from the slot it originally hashed to. When inserting an item, we don't let its distance grow larger than the surrounding items: when we come across a slot with smaller distance, we insert the new item there; any item in the way must be shunted down to make room (and their distances increased). This allows us to speed up lookups.

   On a lookup, we keep track of the distance of the current slot from the intended slot of the query. We can save time using distances in two ways. First, we can save time

[3] This largely replicates the functionality of `strcmp()`. But it's likely that your implementation will be faster since we're assuming that the strings have fixed length. Plus, this is a good warmup for the SIMD version later—and we'll be modifying it for an experiment later on.

[4] For example, let's say an item hashes to the second to last slot in the table—but both the last and second-to-last slots are full.

by only comparing to elements in slots whose stored distance matches the distance of the query element. Second, if the distance stored in the current slot is less than the current distance of the query element, then we can safely state that the query is not in the table. (No further elements in that row will hash to the same slot as the query)

Make sure that your distances stay correct while wrapping around the table. This means that you cannot calculate the distance simply by subtracting the element's original slot from the current slot. In my opinion, the easiest way to maintain the distance is to keep it as a variable, incrementing it each time we move to the next slot in the table.

3. **Item signatures:** When performing a lookup, the query needs to be compared against a number of items in the hash table with the same stored distance. These comparisons are expensive for strings. Therefore, let's store a short hash for each string; we can call this a *signature*.[5] We'll compare the signature of the query string to the signature of the element stored in each slot (so long as the robin hood distances match). We only compare the strings for elements whose signatures match.

   You need a way to distinguish empty slots in your hash table. Maintain the invariant that any slot with both signature and distance equal to 0 is empty. (This allows you to check for emptiness quickly by checking if the 32-bit slot has value 0). This means that you should ensure that no element has a signature of 0.[6]

Code Parameters

Please use the following parameters in your code.

- Your hash table should be 95% full; that is to say, the number of slots should be the number of stored entries divided by .95.
- Your implementation should be able to handle using *any* number of bits to store distances—this value should be a variable that can be easily changed. (That said, 8 is likely a good value for testing.)
- Your algorithm should fail if, at any point, it attempts to store a distance that cannot be stored in the given number of bits. (That is to say: it's OK if your hash table doesn't have a rebuild mechanism. But it should check for this case and fail gracefully, rather than potentially giving an incorrect answer.)
- Each slot in your hash table should be 32 total bits, where the given number of bits (see discussion above) is used for distances, and the rest of the bits are used for signatures
- You should have a second array to store the original strings themselves. This array should mirror the hash table: a shift in the hash table should result in a shift in this array.

[5] Recall that MurmurHash gives several outputs for each hashed element; often this is stored as an array of integers. One good way to implement signatures is to use the first element in this array to find the slot for an element, and to use the second element in the array for the signature.

[6] You may use mod for this if you want. Or, you can set elements with signature 0 to have signature 1.

- You should not use the modulo operator to implement circular hashing; you should use if statements. (This is both for efficiency and to help avoid bugs.) (You may use modulo to calculate the slot of an element—something like `slot = hash % numBuckets` is fine. Modulo for calculating signatures is also OK.)

Questions

Code (50 points). Implement the hash table described above.

Problem 1 (10 points). Create an efficient function `simd_compare` to compare two strings using SIMD instructions (i.e. using AVX2 intrinsics). Set your code so that if `USE_SIMD` is 1 (this variable is in `hashtable.h`), your code will use `simd_compare` (rather than `manual_compare`) as its function to compare strings.

Your strings have 80 characters. Your function should use two 256-bit SIMD compare instructions, each of which compares 32 characters at a time. (Of course, you'll need to load these values into memory, and interpret the result of the comparison—these will each require further calls to intrinsic functions.) You should compare the remaining 16 characters using two 64-bit comparisons.

If `USE_SIMD` is set, `test.c` will compare the speed of your two compare functions. If implemented properly, the SIMD version of the compare function will be very noticeably faster in these tests.

Solution.

Problem 2 (10 points). Despite the SIMD compare function being faster, your implementation likely requires a similar amount of time with and without SIMD on.

Perform the following experiment. You should have `USE_SIMD` set to 0 for both of these cases. You also probably want to use a fixed seed for your hash to ensure consistency between runs.

- With `USE_SIMD` set to 0: Compare the time required for all positive queries[7] for your algorithm, with the time required for all positive queries where you test ONLY signatures (you don't compare to the original strings). What is the running time in each case?
- With `USE_SIMD` set to 0: Now, modify your `manual_compare` function so that it only compares the first character of each string—if the first character matches it should return 0, otherwise 1. What is the running time after this modification?

In my experiments, testing only the first character (rather than all 80) took around twice as long as only checking the signatures. That is to say, testing just the first character took approximately as long as doing the full comparison (whether or not SIMD is on). (Hopefully your results were fairly similar—but if not, please answer the rest of this question on the basis of my results.)

Explain what is causing this discrepancy—why is even comparing a single character of each string so costly (minimizing the impact of SIMD optimizations)?

[7]You just need to do positive query times, though other times are welcome. Because we use signatures, only positive queries are significantly affected by string comparisons.

Solution.

Problem 3 (10 points)**.** In a Robin Hood hash table with linear probing and load factor α (load factor is the percent of full entries—we had $\alpha = .95$ in this assignment), each query must be compared to $O(\alpha)$ entries in the table in expectation.

Assume we use randomly[8] generated signatures of s bits. We insert n items into the table (the table has n/α slots), and perform n negative queries to items not in the table.

Let's say our table's answer is generated solely based on signatures—it does not compare the query to the original string. In the above set of queries, how many mistakes are made in expectation? In other words, how many times (in expectation) is one of the negative queries compared to an element that it shares a signature with?

Solution.

Problem 4 (10 points)**.** For this question, assume there are no spaces in the data (no `' '` characters).

To create the negative queries data for this assignment, I created 2881018 randomly-generated strings of length 80, where each letter in the string was randomly selected from `'A'`, `'C'`, `'G'`, or `'T'`.

Each of these queries was assumed to be negative—I assumed that none of the randomly-generated strings happened to be the same as one of the 2881018 strings in the actual dataset.

Give a good[9] upper bound[10] on the probability that this assumption is correct. Give your result as a negative power of[11] 10. Be sure to show how you got your answer.

Solution.

Problem 5 (10 points)**.** In class we saw how locality-sensitive hashing can help find similar bit vectors. Can we do the same for strings?

Let's say that I define the similarity of two strings s_1 and s_2 that have the same length ℓ as:

$$(\text{Number of characters } s_1 \text{ and } s_2 \text{ have in common})/\ell.$$

This similarity measure seems a bit like Jaccard similarity, though it is not the same.

Consider the following hash: the hash consists of[12] a number $h \in \{1, \ldots \ell\}$. Then for a string s, the hash of s is the hth character of s.

[8] You may assume signatures are ideal: each bit is independent and uniform random.

[9] By "good" I am only intending to rule out upper bounds like 1 or 1/10, which are correct but don't give much information.

[10] Note that I'm asking for an upper bound. That almost certainly means that I'm looking for you to use a technique we learned in class—a technique to upper bound probability—to keep things reasonably simple.

[11] Wolfram alpha (wolframalpha.com) is one free, accessible resource that can handle these kinds of calculations—of course, you can use your favorite at-home calculator instead.

[12] This is similar to the permutation underlying a given minHash.

For example, let's say we have two strings: $s_1 = HOP$ and $s_2 = POP$. Note that the similarity of these two strings is $2/3$. If $h = 1$, then s_1 hashes to H and s_2 hashes to P. If $h = 2$, then s_1 and s_2 hash to O. If $h = 3$, then both s_1 and s_2 hash to P.

Show that this hash is locality-sensitive according to the definition given in class. (Using r, cr, p_1, and p_2.)

(Note that, like Minhash, this is not a particularly effective hash on its own—we need to concatenate $\Theta(\log n)$ independent copies for it to work well for similarity search. But we'll leave that to another time.)

Solution.