504.420 Algorithms for Bioinformatics

William McElhenney

JHU ID: wmcelhe1

Project 2 Analysis

Due Date: 11/5/2019

Turned In: 11/6/2019

Project 2 Analysis

Analysis of hash tables

The code for the hash tables is contained in *hashing.py*. This file provides functions for the initialization of the hash table, the probing strategies, and addition of elements to the hash table.

The hash table itself is built of basic python lists. In the case that we are not chaining the hash table will consists of either *Nones* or integers (or lists of None and integers if *bucket_size > 1*). In the case that we are chaining, the table will consist of *links*, which are an inner class to the *hash_table* to hold the stored value and an integer representing the next link in the chain (free-space is maintained as a list, which can act as a stack in Python).

The space complexity of the hash table then depends on the data structure we are storing. For non-chained tables, our space complexity *O(n)* where *n* is the maximum number of items stored in the table (120 for all in this project). In the case of chaining, it is closer to *O(2n)* as the *link* structure is two integers.

Time complexity for adding to the table without need for probing is *O(1)*. When we include quadratic or linear probing our complexity goes up to *O(n)*, where *n* is the number of buckets in the table. If we using chaining for "probing" our time complexity is technically *O(m)*, where *m* is the number of links in the chain of the original hash, as we will traverse the whole chain before popping space from the free-space stack (which occurs in *O(1)*). If we were to pull from the table, the time complexities should be approximately the same as adding to the table (pulling was not necessary for this project).

Hash tables are useful in a bioinformatics context in that they allow quick lookup of hashed values, which can be useful for storage of sequence data (nucleotide or amino acid sequences).

What I Learned

I do not feel like I learned much from this project having already implemented a hash table like this before.

What I Might Do Differently Next Time

The *add()* function for adding to the able could be much neater. It needs a series of branches to make sure that the different storage options as described in Analysis, but I think I could have done it in a cleaner (less convoluted) way.

The package *argparse* is probably less headache than what I went through for the input parameters in this project, so I'd probably use that next time.

I also need to read the handout better.

Design Justification

The program made of three files *hashing.py*, *fileIO.py*, and *main.py*.

Starting with *main.py* we have the necessary functions for command-line interaction with the hash tables. That is to say that this file provides functions for the parsing of command-line arguments. The command-line arguments may be entered either positionally, or by name (following the convention *[parameter]=[value]*). Argument methods may not be mixed. The inclusion of the named parameter method seemed necessary as there was no logical way to remove the positional elements from runs that did not need it (the student hash method has no need for *mod* for instance). If an argument has invalid parameter data it will be replaced with a default in order to continue running the program (this safety measure is also mirrored in *hashing.py*). If a default parameter is used the user is notified in terminal. This method also provides a help menu that may be invoked via the command *python3 project2.py --help* or by simply entering *python3 project2.py*. This was included after getting frustrated trying to remember all the parts necessary to configure a table when testing the code. Named parameter parsing is achieved with regular expressions.

fileIO.py provides two functions for the input and output of data, respectively. The input method *read_input()* at lines 18 to 29 uses regular expressions to control the input to only integers followed by some whitespace (0 or more whitespaces) and a newline. This prevents multiple numbers from erroneously existing on the same line in the input file and ensures that only integers are obtained from said input. The output function *write_outp()* at lines 38 to 54 takes in a hash table and invokes its *to_string()* method in order to print the table to the file. It also polls the hash table for the statistics gathered over the course of running.

hashing.py provides all the necessary functions for the storage and maintenance of the hash table. The hash table itself is designed as a class with several default parameters set (input file is not hardcoded, but there is an option for a default output filename, but it is expected that the input parameters provide a non-default one). Unlike what was suggested *linear()* and *quadratic()* are separate (lines 113 – 118 and lines 136 – 153, respectively) as I found when I implemented it as one function the code was very messy (not that it's still not messy but it's better). Both of these are generator functions providing an iterator like function in order to iterate over possible slots from the original (collided) hash. *chaining()* (lines 161 – 183) is not a generator function as there is no need to iterate over possible spaces because free space is maintained as a stack. This function simply takes the original (collided) hash, traverses the chain, and then adds it to whatever space pops off the free space stack. The *add()* function (lines 193 – 262) adds elements to the table taking care of trying the original hash and then coordinating the correct probing strategy if a collision occurs. This function also takes care of maintaining the statistics of the table, including collision count, unplaced elements, and the fill ratio of the table. c*lass_hash* and *my_hash* are the two possible hashing strategies (lines 269 – 270 and 278 – 310 respectively). *class_hash* is a simple modulo divisor method, but my method (*my_hash*) is a riff on the middle-square method, which Wikipedia states is a form of multiplicative hashing. The method takes the middle-ish three digits of the square of the element to be entered and uses that as the hash (because our table only has 120 slots, not 1000). Hashes that would be too big for the table are reduced to fit the table with a modulo of the table size (I hope this meets requirements, as it is not a direct division hash). The final function of this file is the *to_string()* function, which formats the table as a string for output.

Issues of Efficiency

I do not think there are any issues of efficiency to really be considered here.

Certainly, the storage amount does differ with when chaining, but I don’t think this is really that big of an issue and it is necessary for the proper function of the program.

I think the time complexity should be approximately what is expected from hash tables.

General Issues

The handout specifies mod 41 for the *bucket size > 1* scenarios, which to the best of my understanding will not work because we only have 40 buckets (containing 3 slots). It’s a little late for me to ask at the time of this writing, so to deal with this I use the specified mod value but restrict the hashes that can be produced to the table size (maximum of 40 in that case then). This provides additional support for errors users might make (i.e. mod value too big), so I think this is an appropriate (and more importantly repeatable) solution.

Enhancements

In addition to the required statistics, I also calculate the fill ratio (full slots to total slots) for each of the tables.

I also tried varying the *c* values for the quadratics (with labeled outputs of course) outside of the specifications ([0.5, 0.5]).