**CMSC858D Homework #2: Minimum Perfect Hash Function Implementation**

Report by: Sanna Madan

**Short description of implementation:**

This implementation of BBHash works by creating a cascading set of bitvectors for a set number of levels, before ending in a terminal hash table. At each step, the hashes of each key are computed (modulus N\*gamma). Then a helper function ArrayFill checks to see if there are any collisions by creating a bitvector and a temporary corresponding bitvector. If there is a collision, the colliding keys are passed on to the next iteration of BBHash and a new bitvector is created with a new hash function (by changing the seed). The lookup function then finds which level of bitvector that the index = hash(key) equals 1. The previous levels’ weights are summed up along with the rank of the index at the current level, or if this doesn’t occur then the terminal hash table result is returned.

**Most difficult part of implementing the project:**

I had several difficulties, the major ones just being C++ environment issues and debugging. I initially struggled with importing the xxHash library, which took me a disproportionately long amount of time, probably due to my lack of C++ expertise. This ended up requiring me to change how I was compiling my code (in a Makefile instead of with CMakeLists.txt in CLion or g++ in terminal directly). While understanding the algorithmic implementation itself wasn’t too difficult, I ran into many bugs in the way and had to spend a lot of time debugging (hence my code is inundated with print statements). I would often get segmentation faults, and ultimately the errors ended up being small fixes for the most part that just took a really long time to trace and solve solve. For example, the “last” variable was messing things up. I realized I had to iterate up to but not including “last”, which was unlike what the algorithm we cited did. There were subtle nuances that I had to thus sort out. I had print statements to track whether certain loops and methods were entered and therefore executing, and if they weren’t that was a huge clue as to what went wrong. Creating the terminal hash table and lookup functions were also tricky as I was getting completely unreasonable values, and I then realized one of the major errors was I was looping over bit\_arrays.size() instead of bit\_arrays[i].size() for each key set. Overall, though the algorithm was straightforward and elegant to understand (thanks to the pseudocode especially), debugging took up by far most of the time.

**Plots and analyses**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **Hash algorithm** | **Gamma** | **Speed (microseconds)** | **Size** |
| 10 | mphf | 1 | 10 | 31 |
| 10 | mphf | 2 | 11 | 46 |
| 10 | mphf | 5 | 46 | 51 |
| 10 | Basic hash table | n/a | 7 | 10 |
| 100 | mphf | 1 | 6 | 254 |
| 100 | mphf | 2 | 6 | 372 |
| 100 | mphf | 5 | 6 | 697 |
| 100 | Basic hash table | n/a | 42 | 100 |
| 1000 | mphf | 1 | 8 | 3427 |
| 1000 | mphf | 2 | 8 | 5933 |
| 1000 | mphf | 5 | 8 | 13484 |
| 1000 | Basic hash table | n/a | 496 | 1000 |
| 10000 | mphf | 1 | 10 | 40001 |
| 10000 | mphf | 2 | 10 | 70001 |
| 10000 | mphf | 5 | 9 | 160001 |
| 1000 | Basic hash table | n/a | 5432 | 10000 |

*Speed of access*

Ultimately, the time to run is essentially constant for all values of n and gamma for the minimum perfect hash function. However, for the basic hash map, it increases exponentially with n. The average lookup time for the minimum perfect hash function was ~8 microseconds, for all values of n and gamma. For basic hash function it exceeded 5000 microseconds at n=10,000, from only 7 microseconds at n=10.

*Storage space*

The amount of storage increases rather exponentially as you increase gamma. The standard hash map storage increases linearly with n, so it will be more space efficient understandably. Increasing gamma understandably also increases storage space by the factor of gamma. All however have relatively slow growth until n=1,000, at which point all the functions’ slopes rise up (perhaps not all exponentially like mphf, gamma=5 but linearly for sure).

**Alien keys**

In my case, the lookup function returned a value that immediately let me determine if the key was not present in my original set 100% of the time. It simply returned zero because the lookup function resorts to the terminal hash table if it cannot find a value for a key, and if that key isn’t even present in that table then the value returned is zero. So, for all my experiments with varying values of n and gamma with alien keys, it always returned a lookup value of zero. From my empirical observations, I do not believe that there is a relationship between n, gamma, and the rate of detecting alien keys.

Citation:

1. Limasset, A., Rizk, G., Chikhi, R., & Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. arXiv preprint arXiv:1702.03154.