Perfect Hashing

Preliminary notes

Overview:

Algorithm to generate perfect hash functions (PHF): Use a version of the Hash, Displace, Compress (CDH) algorithm to generate perfect hash functions. Overview. Paper.

Hash function to use: Bob Jenkins' integer hashing function modified to take a seed

Space/Time Complexity:

CHD:

The authors of CHD outline proofs in their paper for time/space complexity. Their findings:

Time: PHF generation is done in linear time

Space: from the authors: "For a load factor equal to the maximum one that is achieved by the BDZ algorithm (81 %), the resulting PHFs are stored in approximately 1.40 bits per key". For MPHFs, the result is "generate MPHFs that can be stored in approximately 2.07 bits per key". I am not choosing my table size based on load factor, rather the closest power of 2.

Table size: For our table, we will choose a table size by taking our number of keys and rounding up to the nearest power of 2. With a table size of a power of 2, instead of inserting at the regular hash(x) mod tableSize, we can use the bitwise and: hash(x) & tableSize to achieve the same result.

Commented [MB1]: this isn't designed to deal with huge numbers of keys, so we will gracefully fail if the number of keys causes overflow

Hash function:

O(1), 6 shift instructions.

```
/* Bob Jenkins' hash function modified to take a seed
   http://burtleburtle.net/bob/hash/integer.html */
uint hash(uint a, uint seed)
{
	a = (a + seed) + (a << 12);
	a = (a ^ 0xc761c23c) ^ (a >> 19);
	a = (a + 0x165667b1) + (a << 5);
	a = (a + 0xd3a2646c) ^ (a << 9);
	a = (a + 0xfd7046c5) + (a << 3);
	a = (a ^ 0xb55a4f09) ^ (a >> 16);
	return a;
}
```

CHD

What is it?

CHD uses two steps to generate a perfect hash function.

1. Place keys into buckets according to the first hash function

2. Process each bucket (largest first) and try to place each of the keys into empty spaces in the final hash table.

Why CHD?

The other algorithm I considered is BPZ, which is a scheme to generate PHFs and MPHFs based on constructing random r-partite hypergraphs. Both CHD and BPZ will need a lookup table. Like CHD, it's superior to most of the other algorithms currently on the market for this purpose. However, the authors of CHD show that their algorithm often beats BPZ in speed and bits per key. Also, I find CHD's double hashing scheme much simpler to understand than BPZ's graphs :)

Commented [Gu2]: I do agree that CHD is simpler to understand. The CHD paper, however, does indicate that BPZ is faster to evaluate (our primary scenario). Unfortunately, that data isn't necessarily relevant for our scenario which is about hashing 32-bit keys (rather than URL strings). So, I think going with CHD is OK.

Example

Let S be a set of keys {22, 26, 100}.

We eventually want to hash them with no collisions with CHD.

We have a hash function hash(x, d) which returns the hash of a key x based on some seed d. So, we hash every key value of S using hash(s, 0) & tableSize (which you'll remember is size 4 since |S| = 3 ; next power of 2 is 4.).

Let's say that:
hash(22, 0) & tableSize = 2
hash(26, 0) & tableSize = 2
hash(100, 0) & tableSize = 0.

We have collisions, so we group the keys that collide into "buckets".

100
22, 26

We sort the buckets by descending size…

Commented [MB3]: Design consideration: since we know that the number of full buckets is at most |S|, the size of our set (if there are no collisions), maybe using radix sort as opposed to quicksort would be an improvement.

22, 26
100

We know that our original seed *d* causes these collisions, so for each bucket starting with the largest one, we find a new value of *d* that doesn't result in any collisions. We need to remember the values of *d* for later, so we create a lookup table as well as our final hash table.

Start with the values in the {22, 26} bucket. We try d = 0…n until we see that they'll hash to unique slots. With a good hash function, this should not take many iterations. We find that d =4 works.

h(22, 4) & (tableSize -1)= 3
hash(26, 4) & (tableSize -1) = 2. No collisions with each other or anything in the table! So we put them in the table and save that bucket's value of d in the lookup table.

Next, we see that hash(100,0) = 0 still is fine, so we insert into the table there

	Before: hashing with collisions
0	100
1	
2	22, 26
3	

	Lookup (values of d)
0	0
1	
2	4
3	

	Final Hash Table
0	100
1	
2	26
3	22

Lookup:

Now we've constructed our function for each value of S. We have two objects left in memory – Lookup table as well as the actual final hash table.

To look up the item with key 26 in the final hash table, I find the value of d from the lookup table:
1. slot = hash(26, 0) & (tableSize - 1) // recall that this equals 2
2. Go to lookupTable[slot] to find that d = 4.
3. item = hash(26, 4) & (tableSize – 1)

Boom – now we have our object from a constant time lookup.

Why sort the buckets?

There's a proof in the CHD paper that this is what guarantees O(n) time complexity, but if you think about it: If you try the buckets with the most keys first, there are more spots available in the hash table. Therefore, it's easier to find a seed that works.

If you're going to a movie theater with a big group, it's a lot easier for you to find seats if you get there early. If you're going alone, then you can get there right as the movie's starting and still find a seat without much searching.

Seeds

Modified Bob Jenkins hash function to work with a seed