Computer Science 222: Succincter

Tuesday, December 11, 2012

Professor Mitzenmacher

Dan Bradley (dbradley@college), Saagar Deshpande (sdeshpande@college)

Final Project

****

1 Abstract

Using Mihai Patrascu's 2008 paper "Succincter" [1], we implement a way to store trits (trinary values) within 1.01% of the ideal space of $n * log_2(3)$ while having lookup in $O(t)$ time, where t is the depth of our data structure. We find that this is both a fast and space efficient data structure with room for extension past simply storing trits.

2 Introduction

There are few effective methods for storing trits. In this paper, we focus on three of them: the naive method, arithmetic coding, and our succincter implementation.

The naive method is simple. As a trit has more information than a bit, but less than two bits, store each trit using two bits. Obviously this is not the most space-efficient method, you could encode $\frac{4}{3}$ as much information in the same space, so it is clearly wasteful. It is however, very fast, both on encode and decode. Encode is strictly linear, with very small constants. The encoder we used took 3.9 milliseconds to encode 500000 entries, and 13.7 milliseconds to decode 500000 entries.

Arithmetic coding is much more effective at reducing size. By reducing the entries to one number, we can encode in exactly $\lceil \frac{n*log_2(3)}{8} \rceil$ bytes. Decoding, however, ideally takes linear time, as to reach any original entry, you must go through every entry prior, there is no way to randomly access any element. Furthermore, the nature of arithmetic coding lends itself to manipulation of large numbers, which is time consuming.

The succincter data structure is a recursive approach to the problem, and attempts to approach the ideal space usage without sacrificing on decoding time, especially for individual entries. Essentially, succinter takes an array of entries, and breaks them into blocks. These blocks are each treated as one large number, base K (in the first level of the structure when storing trits K is 3, but K changes by level). These large number are then divided by 2^M where M is a user-defined variable. The remainder is stored for transmission, and the quotient gets passed up to the next level of the data structure as a new array. The quotient will be some number $\{0, 1, 2, ..., K-1\}$, and so we can repeat the process on the new array by simply finding new blocks and treating them as numbers base K.

Mihai Patrascu outlined the idea of succinter data structures in [1], but we took some liberties with our implementation, as his approaches to finding block size and K proved less useful in practice than they might have been in theory.

3 Implementation

To encode, we start with an array with fixed size of trits with pseudorandomly assigned values. For the sake of simplicity, and the fact that the values of the trits do not particularly matter for our inquiry, we used the

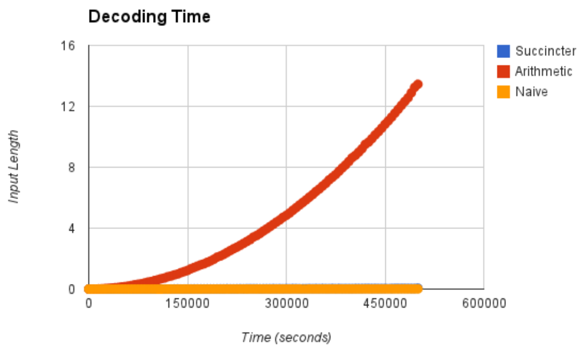
c standard library rand() function seeded by the time of running.

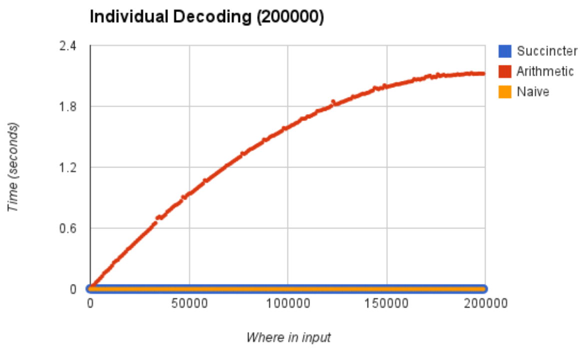
We also decided that all remainders would be up to 32 bits long, which for the sake of discussion we will refer to as M.

With the size of the array and the base 3, we construct the necessary headers to be used by the decoder, specifically the size of the array at that level, the base at that level, and the block size at that level.

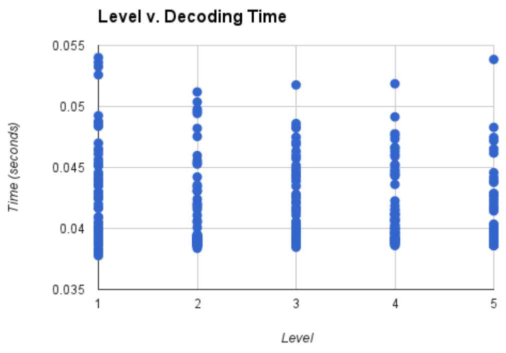
Given the size and base, the header of all levels are constructable. We find an n such that our new $K = \lceil \frac{3^n}{2^M} \rceil > 2$. n is the block size for the level, the base for the level is the old K, and the size for the level is the old size. The variables then update, with old K being set to new K, size decreasing by a factor chunksize, and we start back where we started, just a level higher up the headers.

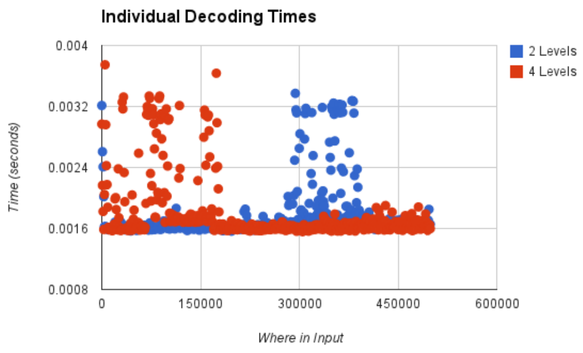
4 Results and Analysis





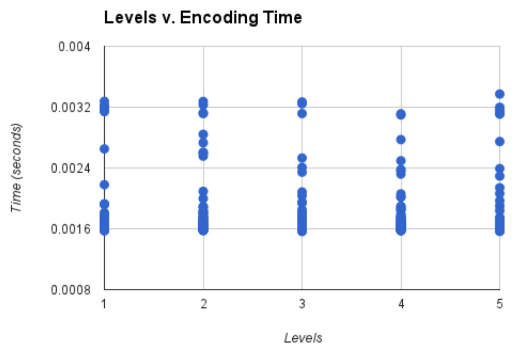
asdfaksdjf;aksldfjadsf

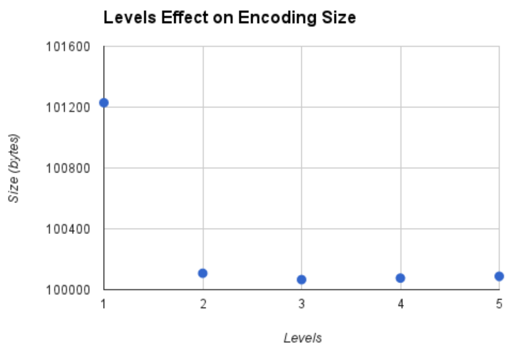




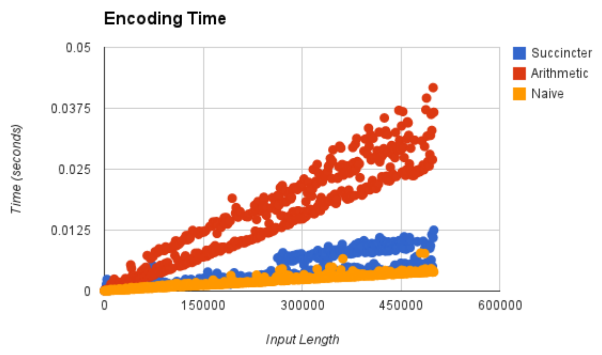
asdfaksdjf;aksldfjadsf
asdfaksdjf;aksldfjadsf
asdfaksdjf;aksldfjadsf
asdfaksdjf;aksldfjadsf

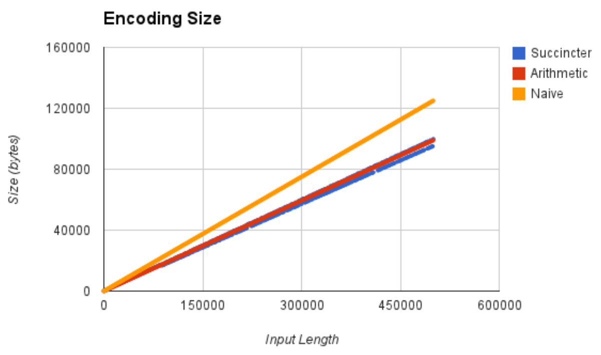
asdfaksdjf;aksldfjadsf





asdfaksdjf;aksldfjadsf





asdfaksdjf;aksldfjadsf

5 Conclusion

****

Appendix