Compression for Wireless Sensor Networks

Daniel Shteremberg
Harvard University
dshterem@fas.harvard.edu

Jason Waterman
Harvard University
waterman@eecs.harvard.edu

Abstract

Abstract

1 Introduction

2 Related Work

3 LZ77

LZ77 [2] seemed like a promising algorithm for Wireless Sensor Network compression. Unlike LZW [1] and other dictionary building schemes, LZ77 maintains a constant size back-buffer with which it performs matches. Without an upper bound on the dictionary size, it is unreasonable to use dictionary based compression algorithms because it is very easy to run out of memory on motes. LZ77, on the other hand, requires a constant amount of memory, and using a pointer based circular buffer, it would be easy to implement LZ77 on a mote without having to copy large pieces of memory.

However, in practice, LZ77 does not perform as well. Further investigation of LZ77 implementations revealed that these implementations use large back-buffers, sometimes as big as 64k entries. Many times, pure LZ77 is not implemented, but a history dictionary is built in order to find matches more quickly. This growing dictionary makes LZ77 prohibitive on sensor motes. Without a history dictionary, matches can take a long time because the entire back buffer must be searched for each new element that comes in. In order to quantify how well LZ77 performs on our three data sets, we compressed our data using a standard implementation of LZ77.

Table 1 shows the compression ratios for our three data sets; high ratios mean bad compression. Note that LZ77 performs very poorly as the best compression ratio is 88.5%. We used a back-buffer size of 1024, which is a reasonable size for a mote implementation. LZ77 performs so poorly for several reasons. First, the back-buffer is not large enough to find a large amount of matches. Also, the noise in the data make matching very difficult. We are not compressing english text, where there is no noise. We are compressing sensor readings coming from an ADC, which have an inherent amount of noise.

Table 1 also shows the compression ratios of two other schemes: Run Length Encoding (RLE) and Huffman encoding. RLE works by encoding consecutive duplicates with two numbers: the element, and the length of the run. Due to the noise in our data, there are very few runs, which is why RLE does so poorly. Huffman coding does somewhat better, but it requires building a tree of prefix codes. This tree could be very large, depending on how many bits each sample has. For this reason, Huffman coding is not useful in a sensor network context.

These experiments have clearly shown that traditional techniques that work in traditional computing environments, are not always suitable in a sensor network context. We have not exhaustively tested all compression schemes, but we have tested a few that are commonly used. Moreover, complex algorithms are more difficult to implement on sensor motes due to their limited debugging capabilities and lack of mature development tools. We realize that the best compression scheme for Wireless Sensor Networks is one that requires little computation, is not very complex, and of course, has good compression ratios for the sensor data.

4 Variable Block Size Delta Encoding

Given the inadequacies of standard compression schemes in the sensor network context we developed a compression scheme that performs well with sensor data, has minimal computation and is easy to implement on motes. We decided to exploit the correlation

	LZ77	RLE	Huffman
Volcano	99.7%	100.1%	81.6%
SHIMMER	88.5% - 98.8%	100.1%	70.8%
Marmot chirps	98.5%	100.2%	70.2%

Table 1: Compression ratios using standard methods

	VBS Delta Encoding
Volcano	53.3%
SHIMMER	42.2% - 62.8%
Marmot chirps	33.1%

Table 2: Compression ratios using Variable Block Size Delta Encoding

between consequtive samples. In sensor data, a sample is very likely to be close to the previous sample. This characteristic lends itself nicely to delta encoding. Because compression is very tied to data transmission in sensor networks, our compression scheme must be designed for radio transmission as well as the data we are compression. Whereas data payload sizes in 802.11 packets are usually around 1500 bytes, data payloads in 802.15.4 packets, which is the protocol used by sensor motes, is between 30-100 bytes.

Given these requirements, we developed a variable block size delta encoding scheme. For clarity of explanation, we will assume we are working with 12 bit data and 30 byte packets. These characteristics are those of the SHIMMER platform from which we obtained the accelerometer data. However, this scheme works for higher order data and large packet sizes. All packets are fixed size, so we want to pack as much information into a packet as possible. In addition, we want to build some redundancy into the scheme. It might be possible to lose a packet in sensor networks. However, since we are using delta encoding, if we miss a packet, we still want to be able to decode the rest of the data.

Each packet contains a block of data. Each block has a header and a delta payload. The first sample in the block is stored in the header uncompressed. This is for redundancy so that if a packet is lost, only the data in that packet is lost and not any subsequent data. The delta is computed for every sample after the first with its previous sample. The number of bits required to encode the delta is calculated and stored. All of the deltas in the delta payload are encoded using the same number of bits. The number of bits to encode the deltas is therefore the number of bits needed to encode the largest delta. As each sample comes in, the size of the block is calculated. If the block still fits in the packet, the delta is added to the block. If the block would not fit in the packet, the block is encoded into the packet and dispatched. In the header we store the uncompressed first element (12 bits), the block size (8 bits), bits per delta (4 bits). The entire header fits into 3 bytes, leaving 27 bytes for the delta payload. There are two reasons why a new sample would not fit in the block. First, if the block already has many samples, a new delta might simply not fit in the block. Secondly, if a very large delta comes in, it would require that all the other deltas be encoded with many more bits, which would push the entire block over the packet size limit.

This compression scheme is extremely simple to implement and requires very little computation. For each sample, a delta must be calculated, it must be compared with the largest delta. When transmitting the block, the deltas must be packed into the payload. Compared to LZ77 or Huffman coding, this is very little computation. In the following section we will evaluate this compression scheme using our data sets and determine how well it performs. We will also quantify the effect of noise on our scheme.

5 Experiments and Results

We compressed our three data sets using our VBS Delta Encoding scheme. These results are presented in Table 2. Notice that the marmot chirp data set has the best compression ratio. This is mostly due to a lower noise floor than the other data sets. In addition, there is very little variablility in the data when there are no chirps, and the occasional chirp has a short duration. Volcano data has more noise and more frequent events. The SHIMMER data also has noise, but it records constant movement, so there are longer duration events with higher frequencies. However, there results outperform LZ77, RLE, and Huffman encoding.

5.0.1 Noise

We performed a simple experiment to quantify how well our compression scheme performs in the presence of noise. We took a constant data set with a value of 3000 and added random gaussian noise. We varied the standard deviation of the noise from 0 to 100. Figure 1 show a plot of four of these noise intensities. Figure 5.0.1 shows the compression raio as a function of the noise standard deviation. When there is no noise the compression ratio is 9.22%. This is the lower bound compression ratio for our scheme. Under perfect conditions (no noise and constant data), our scheme is capable of achieving this compression ratio. The maximum compression ratio with 100 standard devation noise is 69.08%. From Figure 5.0.1 we can see that adding a little noise has a significant effect on the performances. However, after a certain point, our scheme is robust to noise. A standard deviation of 100 for 12 bit data is quite high, and we don't expect data sets to have anywhere near as much noise. For example, the marmot chirp data could not have noise of any more than 4 standard deviation, otherwise it would not have been compressed to 33.1%.

6 Future Work

7 Conclusion

References

[1] T. A. Welch. A technique for high performance data compression. In *IEEE Computer*, volume 17,6, pages 8–19, 1984.

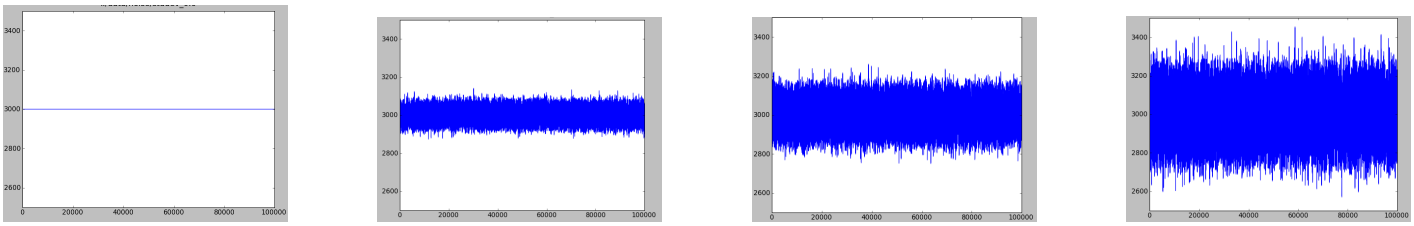

Figure 1: Noise data set with standard deviation of 0, 30, 60 and 100

[2] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

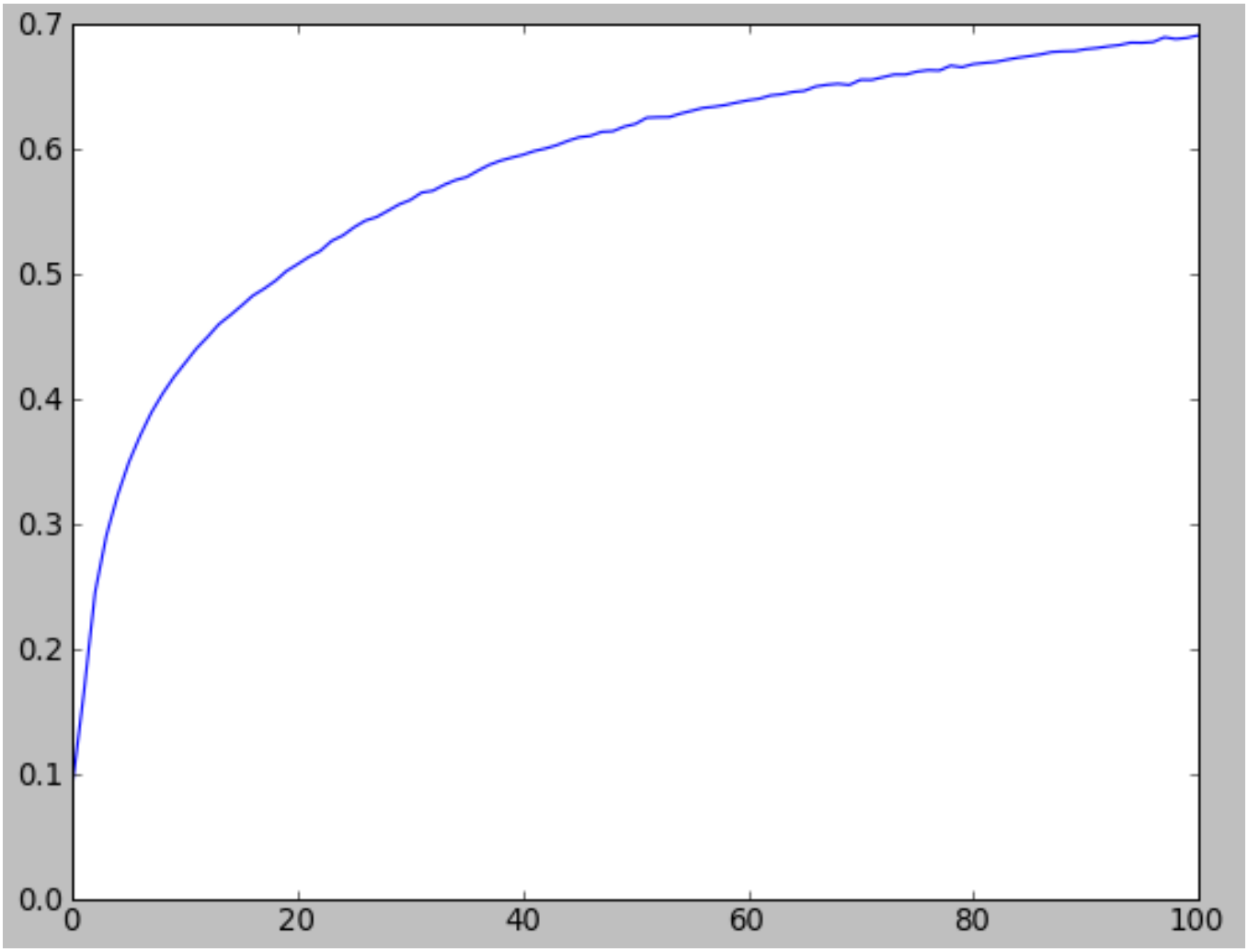

Figure 2: Compression ratio vs. standard deviation of noise