Adaptive Soft Associative Memory

Anand Ramachandran, aramach4@illinois.edu

Abstract of the project
The project started by trying to solve a specific type of problem related to Computational Genomics namely, the error correction of short reads obtained from genome sequencing. This involves pattern-matching operations with patterns of higher recurrence replacing similar patterns of lower recurrence within a sample. Sparse Distributed Memory (SDM) [1] seemed a very good fit as a memory that could automatically effect this operation on stored data. How this can be achieved with SDM will be described later. However, SDM has very large memory requirements for a given vector size and provides only a fraction of that as storage capacity. Moreover, for correlated data, the uniformly distributed hard-locations in SDM should be replaced with trained locations [2][3], which incur additional burdens. When the simulation of SDM ran into such difficulties, a harder look was taken at the problem being solved. The subsequent analysis has resulted in the formulation of an associative memory model that is presented here. The memory has been modeled in Verilog and experimentally tested by implementing a channel decoder with it for hamming codes of distance equal to four between any two codes. This decoder doesn't initially have knowledge of the channel codes, but extracts them from noisy incoming data and once it has locked on the correct symbols, corrects the remaining noisy symbols for 1-bit errors. Though previous work [5] has implemented a decoder using associative memory, this implementation is not presented as another proposal for implementing decoders using associative memory, but only as a test of the memory itself, which is the main contribution of this work.

1. Introduction

1.1 Sparse Distributed Memory.

Kanerva introduced the Sparse Distributed Memory as an associative memory that responds to pattern queries similar to those stored in the memory.

The SDM has "hard-locations" which are random vectors chosen from the vector space specified by a binary vector of width equal to that of the data. The locations can be sparse in the vector space. For example, for data that is 1000-bits wide, the vector space contains 2^{1000} vectors, but the memory can be constructed using a sparse set of, say, 2^{20} vectors which are chosen randomly and uniformly from the vector space. Lets denote this set of hard-locations with the symbol N. Another parameter of the memory is the threshold distance d_{th}. When writing an item X into the memory, all locations within a hamming distance d_{th} of X in the sparse set, N, are selected and populated with X. Let this set of vectors be denoted by O(X), referred to as the circle of X. When reading back from the address X, locations in O(x) are read for their stored vectors. If x_{ij} is the i-th component of j-th vector in O(X), then x_i the i-th component of the result vector is given by

$$x_i = 1 \; if \; \sum_j x_{ij} > 0\,, and - 1 \; otherwise$$

x_{ij}'s are stored in the SDM using counters. If an incoming vector for storage has a value of 1 at position j, x_{ij} is incremented by 1, else decremented by 1. The counters saturate at a pre-specified value.

By the properties of this hyper-dimensional space, most of the space around a given vector is clustered around the normalized distance of ½ (the space should not be too sparse that this property doesn't hold). Meaningful values of d_{th} should be less than ½. If the vector space is too sparse, there is a good chance that many queried input data vectors will not have a hard-location within d_{th}. In addition, distances close to ½ may not be useful in many applications. For applications where data is more clustered around certain vectors and where matching operations need to meet more stringent criteria, the hard locations should be appropriately trained and localized and this requires huge computational overheads for any sizeable number of hard-locations.

1.2 Error Correction of Short Reads in Genomes [4]

For computational purposes, a genome is considered as a string of four alphabets (A, C, G, T). Genome sequencing cuts the genome into what are called "short-reads," which are the outputs of the sequencing machine. Thus, reads are short strings representing portions of the genome. There is a lot of redundancy among the short-reads and the overall "coverage" of the short-reads can be 40 (say) times the overall length of the genome. The genome is re-constructed from the short-reads based on their overlap with each other (overlap length is not fixed among all the short-reads).

A collection of reads can be represented by what is called a de Bruijn graph. The nodes of the de Bruijn graph are unambiguously contiguous portions of the genome reconstructed from short-read data. Ideally the reconstruction should give a linear graph without any branches. It doesn't for two reasons.

1. Repeats: A portion of the genome may occur multiple times through the length of the genome. See the figure below.

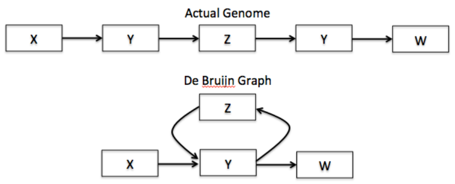

Figure 1 Repeats in Genome

2. Errors: The short-reads obtained from the genome can have errors. Therefore the same portion of the genome can be sequenced into two strings. In the figure below, Y' and Z' are erroneous reproductions of Y and Z.

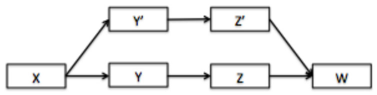

Figure 2 Errors in a genome

The initial proposal was to use SDM to filter out erroneous nodes. All the nodes of the graph can be stored in SDM using it as an auto-associative memory. For example node X in Figure 2 will be stored as SDM(X) = X.

If Y' is an erroneous duplication of Y, two assumptions can be made:

1. d(Y', Y) which is the distance between Y' and Y, is small.
2. Y occurs far more frequently in the genome sample than Y'.

Denoting by O(x) the circle of x, as described before:

$Y' \in O(Y)$

$Y \in O(Y')$

Assuming that we program the nodes in proportion to their frequency of occurrence, if Y has very large multiplicity compared to Y', reading at Y' will return Y or a value closer to Y than to Y'. Subsequent reads will be attracted towards Y and finally converge to it. For the case of repeats, if Y' is due to a repeat, then Y' will not lie inside O(Y) with a high probability, and hence querying at Y and Y' will return different results which do not interfere with each other. If the repeats are similar, the SDM can be slightly modified to have hysteresis to denote an inconclusive result when two candidate data items occur with similar likelihood.

In the complete algorithmic description, the nodes of the graph are programmed into the SDM while the edges are held in a hash table, referred to here just as Associative Memory (AM). AM[X] represents a set of nodes connected to node X. $AM_j[X]$ is the j-th element of the set. The algorithm is as follows:

1. Program all nodes into SDM (Auto-associative).
2. Program all the edges into associative memory (AM).
   For edge (X, Y), we have AM[X] = AM[X] U {Y}
3. For each source node i in AM, read the target nodes $AM_j[i]$ in the SDM.
4. Build a new graph from the read-back data in a second AM.
   $AM2[SDM[i]] = AM2[SDM[i]] \cup \{AM2[SDM[AM_j[i]]]\}$ for all j.

1.3 Limitations and initial workarounds.

Initial attempts at modeling the genomic error correction problem involved, generating a set of random nodes of a graph, which were 1024-wide bit strings and attaching erroneous nodes to existing nodes in the graph. An SDM model was written in Perl and also in Verilog to simulate the error correction algorithm given above. However simulation times and memory requirements were very large for sizeable number of hard-locations (1 million locations), and many nodes never hit a hard-location for tight values of 'd' for a sparser distribution of hard-locations.

Initial work-around involved dynamically allocating hard-locations on a need-basis when an item was programmed into the memory. The algorithm to dynamically allocate a location is as follows:

```
1.  If O(X) is empty go to step 2 else go to step 9.
2.  Create a hard location at X.
3.  for (distance = 1 to threshold) do steps 4 through 5
4.  numLocationsAtDistance = (density/sizeOfVectorSpace) * (bitWidth Choose distance)
5.  Repeat steps 6 and 7 numLocationsAtDistance times
6.  Create a random vector Y such that d(Y, X) = distance
7.  Store X at Y.
8.  End
9.  Store X in all locations in O(X).
10. End
```

Note: Words in bold indicate variables **sizeOfVectorSpace** = $2^{\mathbf{bitWidth}}$

Note: **density** is the total number of locations in a corresponding statically fixed SDM

Figure 3 is a graphical representation of the same.

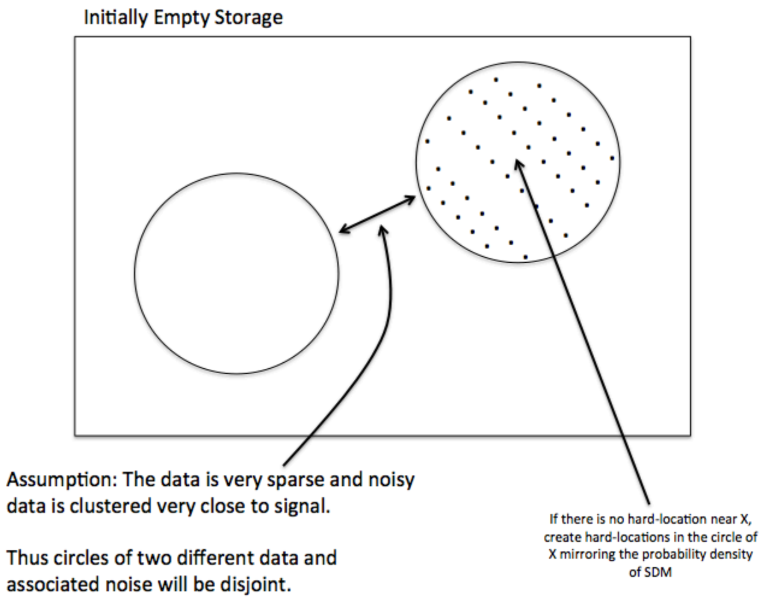
Initially Empty Storage
Assumption: The data is very sparse and noisy data is clustered very close to signal.
Thus circles of two different data and associated noise will be disjoint.
If there is no hard-location near X, create hard-locations in the circle of X mirroring the probability density of SDM

Figure 3 Dynamically Allocated SDM for Noise clustered with Data

2. Proposed memory model

The dynamic allocation scheme explained above led to the formulation of the following simplified associative memory model. The development of the model is described step-by-step.

2.1 Soft Associative Memory

In the proposed soft-associative memory, the locations are not preset. There is a large set of pairs of empty locations in the memory. Let such a pair be denoted by (x_i, y_i) where x_i is the i-th empty address location and y_i is the i-th empty data location. A user can choose any one of these locations, i, and populate x_i and y_i. Let the populated i-th location be represented as (X_i, Y_i). Once all the locations are populated for both the fields, subsequent read responses to an input query X will activate X_i such that $d(X_i, X) < d_{th}$. If more than one location matches the query or if no location matches the query, the response will be null. The response data will be Y_i that is paired up with the activated X_i. This is illustrated in Figure 4.

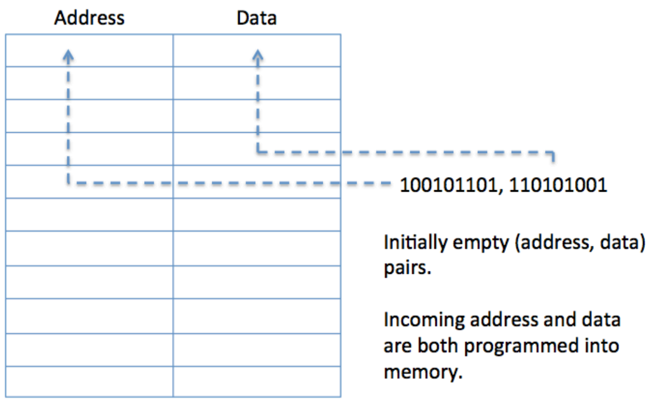
Address
Data
100101101, 110101001
Initially empty (address, data) pairs.
Incoming address and data are both programmed into memory.

Figure 4 Programming into soft associative memory involves programming both address and data

2.2 Counting and Adaptation

To add these two features – counting and adaptation - the bit-vector Y_i in the soft-associative memory described above, is replaced with a vector $\boldsymbol{\delta}_i$ whose j-th component is a counter. The counter is incremented every time a '1' occurs at position j in the write data, and is decremented every time a '-1' (or 0 depending on the representation) occurs at that position. When X_i is activated by an incoming query X, the response is R = (r_1, r_2, r_3, ... r_j ...) such that

$$r_j = 1 \; if \; \delta_{ij} > 0 \; or - 1 \; otherwise$$

where $\delta_{ij}s$ are the components of $\boldsymbol{\delta}_i$.

Thus, each location now holds a history of the previous writes. The value of each bit-position is determined by a majority vote on the past writes to the same bit-position.

Now assume the memory is used as an auto-associative memory that is, the write address and write data are the same. In this case, we can force the bits of the soft addresses in the memory to adapt to the majority-write data in the data field of that location. Majority-write data is considered as "signal" here and the spurious similar write data constitute "noise." The adaptation step can be described as:

$$x_{ij} = 1 \; if \; \delta_{ij} > T, -1 \; if \; \delta_{ij} < \; -T, unchanged \; otherwise$$

where x_{ij} is the j-th bit of the activated address, X_i, and T is a pre-defined threshold parameter.

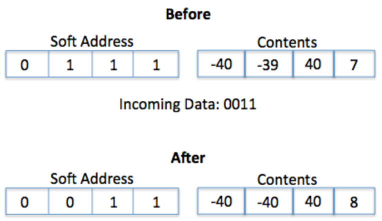

Figure 5 Saturation and Adaptation of address

Adaptation is needed in the case where the associative memory is used to correct errors in the incoming data. If an initial address/datum is noisy, then the subsequent writes to/reads from the memory will be clustered around this noisy address. The distance between a signal and two incoming noisy addresses maybe within the threshold, d_{th}, but the distance between two different noisy addresses may be greater than this value and hence some of the noisy addresses will not be matched to the correct signal address. Also if there are multiple noisy addresses in the memory, their circle of influence defined by d_{th} will overlap and hence there will be ambiguity in the response from the memory.

For example, in {0, 1} notation, consider a 4-bit wide signal to be 0000, and the incoming noisy version of the address to be 0001, which is first initialized into a soft address. Assume $d_{th} = 1$. Now the location 0001, may contain the data 0000 after a few programming cycles, but another noisy input address, 0010 will not be registered as a match because the distance between 0001 and 0010 is $2 > d_{th}$. However if the soft address adapts in the way described here, it will be updated to 0000 and 0010 will correctly activate the soft-location to read out the signal 0000.

2.3 Dynamic Allocation

The next feature added to the associative memory is dynamic allocation. When an incoming address is not matched to any of the existing soft addresses in the memory, the memory picks a free address location and initializes the address at that location to the incoming address and sets the counters at that location accordingly. In the implemented version of the memory, a priority scheme is used.

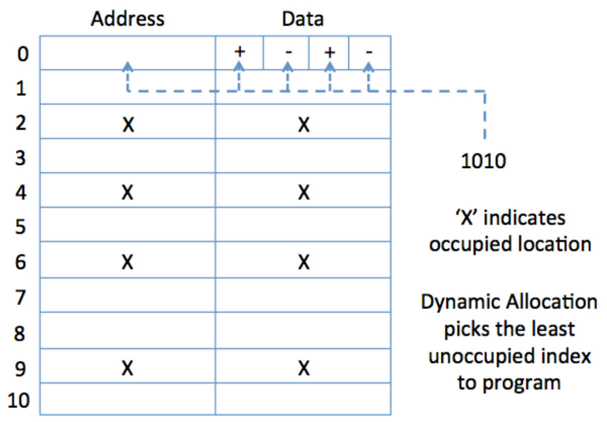
Address
Data
0
1
2
3
4
5
6
7
8
9
10
+
-
+
-
X
X
X
X
X
X
X
X
1010
'X' indicates occupied location
Dynamic Allocation picks the least unoccupied index to program

Figure 6 Dynamic Allocations

2.4 Strength and Unlearning

A memory location described by (X_i, δ_i) is said to be strong if for all components δ_{ij} of δ_i $|\delta_{ij}| > H$, a pre-defined hysteresis threshold value.

In case multiple locations are activated by an incoming address query, X, all the weak locations among these activated locations will be de-allocated. This maybe referred to as unlearning.

The feature of unlearning is added as yet another protection against noisy data getting initialized into the memory's soft locations. Counting and Adaptation described in section 2.1 are expected to work to correct minor refinements in the stored address, and unlearning is designed to cancel out large errors, which cause two different locations to be initialized to noisy address corresponding to the same signal address.

2.5 Adaptive Soft Associative Memory – Memory Cell Architecture

Figure 7 gives the overview of the architecture of a memory cell in the Adaptive Soft Associative Memory. Input address is compared with the soft-address with regards to hamming distance. The signal, hit indicates whether it is within the circle of the incoming address. If no other memory cell has a hit for the given address, the present memory cell is activated and the operations described in the previous sections are carried out.

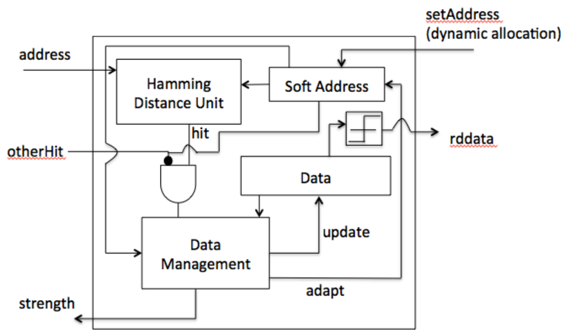

Figure 7 Architecture of a memory cell

3. Experimental setup and results

The adaptive soft-associative memory has been written in Verilog in fully parameterized form. The parameters include address width, number of locations, and the threshold values, d_{th}, T, and H. To experimentally test the associative memory it has been set up to learn and correct noise in transmitted symbols. The symbol set consists of the following four symbols. A quick examination shows that they have a hamming distance of 4 between them.

Symbols: 11111111, 11110000, 11001100, 10101010. Errors can be unambiguously corrected when there is a 1-bit error in the transmitted symbol. The associative memory is observed to learn the correct symbols from noisy symbols. For the experiments, symbols are randomly picked uniformly from the afore-mentioned set and noise is added to every bit with a probability of error following uniform distribution. For the given probability of error per bit, an ensemble of 64 experiments is carried out. Once the correct symbols are captured in the soft addresses, subsequent 1-bit errors in the input are corrected by the associative memory. The following table tabulates the results of the experiment.

Probability of error (per bit)	Successful learning (out of 64 trials)
0.03125	64
0.0625	64
0.09375	63*
0.1250 **(average 1-bit error per symbol)**	64
0.140625	64
0.15625	60
0.1875	42
0.21875	11
0.25	1

(*the failure disappears when the threshold H is increased from four to 12)

4. Future work

- Explore the theoretical framework of the memory in terms of machine learning formulation.
- Formulate hetero-associative memory from the same concepts.

References:
[1] P. Kanerva, “Sparse Distributed Memory”, The MIT Press, Cambridge Massachusetts, 1988
[2] Tim A. Hely, David J. Wilshaw, Gillian M. Hayes, “A New Approach to Kanerva’s Sparse Distributed Memory”, IEEE Transactions on Neural Networks, Vol 8., No. 3, May 1997
[3] S. W. Ryan and J.H Andrease, “Improving the performance of Kanerva’s associate memory,” IEEE Transactions on Neural Networks, vol. 6, pp. 125-130, Jan 1995
[4] Daniel R. Zerbino, Ewan Birney, “Velvet: Algorithms for de novo short read assembly using de Bruijn graphs”, Genome Research 2008, 18: 821-829
[5] L. Ionescu, C. Anton, I. Tutanescu, A. Mazare, G. Serban, “Error Correction and Detection System Based on Hopfield Networks,” The 7th International Conference for Internet Technology and Secured Transactions