Multidimensional Incremental Parsing: Current Status and Future Directions

Anas Youssef Boubas and Saad Harous

***Abstract*—Multidimensional Incremental Parsing (MDIP) is a source-coding scheme based on Lempel-Ziv incremental parsing. It becomes especially useful when there is a need to do source analysis based on pattern-occurrences. Since in incremental parsing the source is inherently coded by extracting codewords of high frequencies, a big portion of the pattern analysis process will have been already done by the coding scheme. MDIP, however, is very recent and did not receive as yet lot of attention from the research community, since it is less efficient in terms of compression ratio. We argue here that the space and time savings during the pattern-analysis process far outweigh the decrease in compression efficiency. We aim with this work to spur interest in the research community to further investigate MDIP. We present here a critical review of the MDIP scheme, identifying many open problems and areas where there is room to improve.**

***Index Terms*—Patch-based analysis, data coding, incremental parsing, semantic retrieval, pattern matching, pattern analysis, pattern recognition.**

I. Introduction

In incremental parsing coders such as LZ78 [1], and its multidimensional variant MDIP threshold [2]-[5], no prior knowledge of the source and its statistical descriptors is assumed. This is particularly useful in data streaming applications, where the source itself is altogether unavailable in its entirety while coding. MDIP consists of three main stages: a pattern-matching scheme termed maximum decimation matching (MDM), codeword assignment, and dictionary augmentation. In incremental parsing coders in general, the encoded stream consists of pointers to codewords, each suffixed by a single symbol. The codewords themselves come from a dictionary that is incrementally constructed in real-time during the encoding of the source, and incrementally reconstructed (also in real-time) as an exact replica at the decoder side. Each time a codeword-suffix pair is transmitted/received, the encoder/decoder add the pair as a new codeword to the dictionary. In MDIP multiple suffixes could be transmitted with each codeword pointer. For a source of m dimensions, each of these suffixes represents codeword growth in one of the m dimensions. The main idea behind incremental parsing is to transmit small pointers in place of huge codewords. The bit-rate of pointers adapt to the current size of the dictionary, which is a known parameter at both the encoding and decoding ends.

Manuscript received February 5, 2014; revised April 18, 2014.

A. Y. Boubas and S. Harous are with the College of Information Technology, United Arab Emirates University, Al Ain, United Arab Emirates (e-mail: ab@uaeu.ac.ae, harous@uaeu.ac.ae).

II. Source and Codeword Definitions

Considering the source S as a 3-dimensional cuboid (such as a video stream), or in general an m-dimensional closed convex paralleloid of discrete source symbols $S_{s_1,\dots,s_m}$ taken from an alphabet $\mathcal{A}$:

$$\boldsymbol{S} = \left\{ S_{s_1,\dots,s_m} : \begin{array}{c} 0 \le s_i < n_i, \\ i = 1,\dots,m, \\ (s_1,\dots,s_m) \in \mathbb{Z}^m \end{array} \right\} \quad (1)$$

where n_i is the length across the i^{th} axis y_i. As the coder progresses through the source, it will be extracting subsets $S(\vec{s};\vec{a})$ of S as new codewords:

$$\boldsymbol{S}(\vec{s};\vec{a}) = \left\{ S_{\bar{s}_1,\dots,\bar{s}_m} : \begin{array}{c} s_i \le \bar{s}_i < s_i + a_i, \\ i = 1,\dots,m \end{array} \right\} \quad (2)$$

where $\vec{s}$ is the multidimensional location vector of the codeword within S, and $\vec{a}$ is an integer vector with a_i denoting the length of the codeword across y_i. The element in S at location $\vec{s}$ is denoted by $S(\vec{s})$ and termed the anchor point of the codeword. There are actually various ways to define anchor points in the literature, as some use the centroid [6] and others use one of the corners of the paralleloid. For the sake of easing later the definition of codeword suffixing we defined the anchor point as the location of the element at $\vec{0}$ local to the codeword (i.e. the first most element).

III. Suffixing

A new codeword can be constructed by suffixing an existing codeword with one or more additional symbols along one of the axes y_i. To prevent running into known issue of suffix ambiguity, suffixing order must be defined. In this paper, we use the same suffixing order used in [5]. Let C_1 be the original codeword, and let $\vec{a}$ be its size vector. Starting with C_1, we first append a suffix of the size $1 \times a_2 \times \dots \times a_m$ in the direction of y_1. This is repeated, each time resulting in a new codeword, until a codeword of the size $n_1 \times a_2 \times \dots \times a_m$ is produced. Now a suffix of size $a_1 \times 1 \times a_3 \times \dots \times a_m$ is appended to C_1 in the direction of y_2, let the resulting codeword be C_2. Starting with C_2, suffixes of size $1 \times (a_2 + 1) \times \dots \times a_m$ are appended repeatedly in the direction of y_1, each time resulting in a new codeword, until we reach a codeword of size $n_1 \times (a_2 + 1) \times \dots \times a_m$. The above process is repeated in the direction of y_2 for $a_2 + 2, a_2 + 3, \dots, n_2$ until we reach codeword size $n_1 \times n_2 \times \dots \times a_m$. Then repeated similarly for all other axes until we reach the full size of $n_1 \times \dots \times n_m$. This is merely the

definition of the *order* of suffixing to achieve a codeword of desired dimensions, but not all the aforementioned codewords are actually produced at the runtime.

It should be noted that this order might not be the most efficient one. Therefore, it might be more efficient to think of other ways to resolve suffix ambiguity, especially in the absence of a comprehensive method for moving the coding point (as will be discussed later). However, this is beyond the scope of this work, and we present it here as an **open problem** for the research community.

IV. Dictionary

The set of existing codewords, the codebook, is sometimes referred to as a dictionary $\mathbb{D}$, and the codewords are then described as the vocabulary. To begin coding a source S, the coder is initialized with $\mathbb{D} = \emptyset$. The following operations are defined for a dictionary $\mathbb{D}$:

- Augmentation: $\mathbb{D} \Leftarrow \boldsymbol{S}(\vec{s}; \vec{a})$
- Codewords count: $|\mathbb{D}|$
- Symbols count of the i^{th} codeword: $|\mathbb{D}_i|$
- The size vector $\vec{a}$ of of the i^{th} codeword: $[\mathbb{D}_i]$

Augmentation is the process of adding to $\mathbb{D}$ a new codeword of size $\vec{a}$ and extracting it at an anchor point $\vec{s}$ in source S. There are several issues to consider in order to perform this process efficiently. We will briefly discuss some of these in later subsections. Although $|\mathbb{D}|$ represents the size of the codebook in terms of number of codewords it contains, it does not reflect the effective size of $\mathbb{D}$, since codewords are of varying sizes. The full size can be calculated by multiplying $|\mathbb{D}|$ by the average codeword size (i.e. average $|\mathbb{D}_i|$).

V. Exact Matching

At each coding epoch the coder performs three main steps: pattern matching to find the best codeword match based on a set evaluation criterion, codeword assignment, and dictionary augmentation. Since we consider multidimensional coding here, maximum decimation matching (MDM) is used for pattern matching. We found the way MDM was discussed in [4] a bit confusing, as the authors relied on *mismatches* for its definition. Since the original LZ78 was based on pattern matching and not pattern mismatching, we redefined the same MDM scheme here using matches rather than mismatches.

A match function is generally defined as $\rho: \mathcal{A} \times \mathcal{A} \rightarrow (-\infty, 1]$, and for our specific case here we will implement it as $\rho(q, w) = 1$ if $q = w$ and 0 otherwise. Now for a given codeword C we define codeword match as:

$$\rho(\boldsymbol{C}, \boldsymbol{S}) = \frac{1}{|\boldsymbol{C}|} \sum_{\vec{s} \in [\boldsymbol{C}]} \rho\big(\boldsymbol{C}(\vec{s}), \boldsymbol{S}(\vec{s})\big) \tag{3}$$

We then obtain M defined as the set of all matching codewords anchored at a specified coding point Δ as follows:

$$\boldsymbol{M} = \big\{\boldsymbol{C} | \rho\big(\boldsymbol{C}, \boldsymbol{S}(\Delta, [\boldsymbol{C}])\big) = |\boldsymbol{C}| \; \forall \; \boldsymbol{C} \in \mathbb{D}\big\} \tag{4}$$

VI. Minimizing Match Overlap

Let E be the set of all loci vectors $\vec{s}$ at which a match was previously encoded. A binary decimation field $F(\vec{s})$ is defined as 0 if $\vec{s} \in E$ and 1 otherwise. Also

$$\boldsymbol{F}(\vec{s}; \vec{a}) = \Big\{F_{\bar{s}_1, \ldots, \bar{s}_m} : \quad \begin{matrix} s_i \leq \bar{s}_i < s_i + a_i, \\ i = 1, \ldots, m \end{matrix} \Big\} \tag{5}$$

The decimation level is basically indicating the number of symbols that can be encoded by a given match and were not encoded already by a previous match. This scheme obviously allows match overlap, yet it could be desirable to minimize such overlap. Hence, the codeword $\mathrm{C}_{\max}$ of the MDM is picked by a decimation maximization function:

$$\boldsymbol{C}_{max} = \underset{\mathbb{D}_i \in \boldsymbol{M}}{arg\,max} \big\{\textstyle\sum_{\vec{s} \in \boldsymbol{F}(\Delta; [\mathbb{D}_i])} \boldsymbol{F}(\vec{s})\big\} \tag{6}$$

We argue here that minimizing the overlap is desirable only from a compression viewpoint. Allowing overlap could be beneficial for pattern analysis in some cases. For instance, consider two codewords that contain no edge data, but each occurring on either side of the boundary of an edge. The edge data was not captured by the codeword, therefore no codeword represents the edge occurrence. Slightly shifting (by redefining F) either of the codewords to overlap with the other could be desirable in this case. We leave this as an **open problem** here, and perhaps some solutions tapping a balance between compression and analysis could be inspired from work in [7].

VII. Coding Point

At the beginning of every coding epoch, the coding point Δ is determined. There are two methods for determining Δ, *absolute* and *approximate* MDM. In absolute MDM, determining Δ is considered by itself a search problem. It was noted in [4] that searching for the best Δ would involve in that case very high computational complexity, but will require the least number of coding epochs. In fact, the problem is even more complicated, since the optimal Δ at the current coding epoch might result eventually in less optimal Δ's at later coding epochs. Therefore we identify this as a *shortest path* problem (as defined in the field of graph theory), and a solution can be perhaps inspired from graph theory or optimization algorithms such as Genetic Algorithms or Particle Swarm Optimization [8-9]. We leave this here as an **open problem** for further research. Approximate MDM is a less optimal solution, but more computationally feasible. In approximate MDM a raster is used for determining Δ. The raster has to be known to both the encoder and the decoder to avoid passing Δ for each coding epoch.

Another issue that we present here as an **open problem**, is that the above definition of MDM does not include support for source streaming, which is considered one of the main applications of the original LZ78. In streaming applications, the complete source might not be pre-known to the encoder.

VIII. Codeword Assignment

Once $C_{\max}$ is obtained via MDM for a desired Δ, the dictionary is augmented with new codewords produced by suffixing to $C_{\max}$ in accordance to the suffixing order defined earlier. It is possible to have multiple new codewords from one $C_{\max}$, each growing by 1 unit in the direction of the available axes y_i. It is also possible that the growth in the direction of one of the axes y_i is prohibited by the suffixing order. For example, consider the case of $[C_{\max}] = \vec{a}_{\max} = [2 \quad 2]$. Only one new codeword is possible in that case $[C_{\text{new}}] = \vec{a}_{\text{new}} = [2 \quad 3]$. The other codeword $[C_{\text{new}}] = \vec{a}_{\text{new}} = [3 \quad 2]$ is prohibited by the suffixing order, since originally $C_{\max}$ was arrived to by growing in the direction of y_1. In order to arrive to a desired size of 3×2, one would need to grow codewords in the following sequence of dimension vectors $\vec{a}$: $[1 \quad 1] \rightarrow [2 \quad 1] \rightarrow [3 \quad 1] \rightarrow [3 \quad 2]$. The dictionary is augmented with all possible codewords that can grow from $C_{\max}$ along any possible y_i without violating the suffixing order.

It can be noticed that the suffix can grow very large, especially in high dimensionalities. At each epoch, the encoder would need to transmit all the suffixes used for creating new codewords, and the suffixes themselves would need to be coded if compression ratio is to be enhanced. For more details on this and on dictionary augmentation methods and algorithms, reader is referred to [4].

IX. Lossy Coding

In order to accommodate lossy coding, approximate matches are to be allowed. As discussed earlier, there are two important factors for defining an approximate match. The first one is the deviation distance function that given two paralleloid, it computes the amount of deviation or distortion between them. This can be modeled statistically for universal coding, or with an application specific model for specialized coding. The second factor is the allowed distortion threshold. This can be defined as being globally constant over the source, or as a function of the location within the source. The later would incorporate defining a measure of local sensitivity to error, and implementing such sensitivity.

An approximate match function is first defined as $\rho_{\text{apx}}: \mathcal{A} \times \mathcal{A} \rightarrow [0, \infty)$. Although in [4] $\rho_{apx}(\text{sym1}, \text{sym2})$ was defined as $|\text{sym1} - \text{sym2}|$, the exact implementation of ρ_{apx} is left out here, and in fact can be application specific. Now for a given codeword C match approximation can be found by:

$$\rho_p(\boldsymbol{C}, \widehat{\boldsymbol{C}}) = \frac{1}{|\boldsymbol{C}|}\left(\sum_{\vec{s} \in [\boldsymbol{C}]} \left| \frac{max\left\{\rho_{apx}\left(\boldsymbol{C}(\vec{s}), \widehat{\boldsymbol{C}}(\vec{s})\right) - \tau(\vec{s}), 0\right\}}{\tau(\vec{s})} \right|^p \right)^{1/p} \tag{7}$$

which is basically disregarding any distortion below $\tau(\vec{s})$ as nonexistent. It can be seen that this reduces to a weighted l_p norm. Traditionally, l_p norms measure the deviation (i.e. mismatch) amount, with 0 denoting prefect match, and anything larger than 0 as a deviation (considering positive norms here for illustration). To rather measure the amount of mismatch using the above one needs to flip the domain of the inner parts of the l_p norm, and then shift it to end with 1. We leave it here at this point.

Two norms were experimented with in [4], the Euclidean and the Chebyshev norms. In order to redefine the set M_{apx} of approximately matching codewords at a specified coding point Δ, we first define ε as the lower-bound of the acceptable ρ_p. M_{apx} will then be:

$$\boldsymbol{M}_{apx} = \left\{\boldsymbol{C} | \rho\left(\boldsymbol{C}, \boldsymbol{S}(\Delta, [\boldsymbol{C}])\right) \geq \varepsilon \; \forall \; \boldsymbol{C} \in \mathbb{D}\right\} \tag{8}$$

where ε is chosen such that $0 \leq \varepsilon \leq |C|$. It should be noted that this is slightly different from the definition in [4], since we are using here matches for the definition rather than mismatches.

While taking into consideration local sensitivity to error is a good enhancement over using constant error threshold, we argue here that local sensitivity to error is not independent of neighborhood distortion. Therefore, in addition to being function of location, the local sensitivity to error should also be a function of the current distortion map of the source. This will obviously raise the problem of *error intra-dependency*: allowing the introduction of an error at location $\vec{s}$ could mean that an existing sub-threshold error at another location $\vec{d}$ now becomes supra-threshold. Hence, finding the optimal distortion map that does not break the threshold becomes challenging. Solution to this problem could be either computational or adaptive (like hill-climbing). It is understood that computational solutions are most likely going to be application specific. On the other hand, a universal adaptive framework could be built, which would only require replacing the distortion measure with an application specific one. This is an interesting open problem to be considered in future work.

X. Suffix Compression

The suffix mechanism was noted earlier in Suffixing section. In this section we visit the issue of storing and utilizing the suffixes.

At each coding iteration the extracted patch is grown in different directions by suffixing to it some symbols to form new patches. Since those new patches are a composition of an existing patch and a suffix, it makes sense to only store the new part and a reference to the existing patch. For this, a suffix tree can be constructed. Each node in the tree represents a suffix, with the root node being the empty symbol. By traversing the tree, one can reconstruct all the patches, appending one suffix at a time.

Suffix tree helps in saving bits for the construction of patches, but it does not help in compressing suffixes themselves. This has two impacts. First, suffixes could be encoded via the incremental parsing scheme, leading to higher compression ratio. Specifically, significantly higher compression ratios could be achieved in high dimensionality sources. Second, it could be interesting to analyze suffixes, in addition to analysis of patch occurrences. This could be interesting in many cases, such as high-energy textures. Therefore, suffix compression is an interesting open problem that might require a datastructure that is more involved than suffix trees.

XI. Conclusion

This work was a critical review of a recently developed multidimensional incremental parsing (MDIP) framework. To that end, we came to the following conclusions:

1) MDIP is a new and very promising coding approach that is still in the early stages of development.
2) MDIP source-coding framework is especially useful for use in analysis applications that are based on pattern occurrence analysis.
3) Source compression is also a desirable effect that MDIP promises, though it comes second to source analysis.
4) We found that in both promises MDIP still requires further work, and we presented here a handful of directions for further development.

References

[1] A. Lempel and J. Ziv, "Compression of individual sequences via variable-rate coding. *Information Theory," IEEE Transactions on 24,* vol. 5, pp. 530–536, 1978

[2] A. Boubas and S. Harous, "Parallel abstraction of images with applications in image retrieval," in *Proc. Information Science, Signal Processing and their Applications (ISSPA), 11th International Conference on,* pp. 170–174, 2012.

[3] A. Y. Boubas, S Harous, and B. Belkhouche, "Image abstraction for improved semantic retrieval accuracy and reduced space-time complexities," in *Proc. 4th International Congress on Image and Signal Processing (CISP)*, Oct. 2011, vol. 3, pp. 1363 – 1367.

[4] S. H. Bae and B.-H. Juang, "Multidimensional incremental parsing for universal source coding," *Image Processing, IEEE Transactions on 17,* vol. 10, pp. 1837 –1848, 2008.

[5] S. H. Bae and B.-H. Juang, "Ipsilon: Incremental parsing for semantic indexing of latent concepts," *IEEE Transactions on Image Processing 19,* vol. 7, pp. 1933–1947, 2010.

[6] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "Patchmatch: a randomized correspondence algorithm for structural image editing," *ACM Trans. Graph*, vol. 28, no. 3, pp. 24:1–24:11, 2009.

[7] S. Avidan and A. Shamir, "Seam carving for content- aware image resizing," *ACM Trans. Graph.*, vol. 26, 2007.

[8] D. Goldberg and J. Holland, "Genetic algorithms and machine learning," *Machine Learning 3*, vol. 2-3, pp. 95–99, 1988.

[9] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence 1,* vol. 1, pp. 33–57, 2007.

Anas Y. Boubas was ranked the top student of the College in B.Sc. from University of Sharjah, United Arab Emirates, in 2007 before receiving M.Sc. in computing, with distinction in Research Project, from Imperial College London, United Kingdom in 2010. He is currently working towards his PhD at the College of IT, UAE University, UAE. His research interests include educational technologies, applications of artificial intelligence, image processing and compression. He has also received many awards, including the London University Outstanding Achievement Medal, Golden and Silver Medals of Geneva International Exhibition of Inventions, Kuwait International Award, Prize for the Best Research Project and Prize for the Best Undergraduate Project from the University of Sharjah, the Tanmiyat Investment Group Award and the Sharjah Executive Council Award.

Saad Harous obtained his PhD in computer science from Case Western Reserve University, Cleveland, OH, USA in 1991. He has more than 20 years of experience in teaching and research in 3 different countries: USA, Oman and UAE. He is currently an associate professor at the Faculty of Information Technology, in United Arab Emirates University. His teaching interests include programming, data structures, design and analysis of algorithms, operating systems and network. His research interests include parallel and distributed computing, and the use of computers in education and processing Arabic language. He has published more than 80 journals and conference papers. He is an IEEE senior member.