Toward Feature Algebras in Visual Databases: The Case for a Histogram Algebra

Amarnath Gupta*and Simone Santini†

University of California San Diego

Abstract

Searching and managing large archives of visual data, such as images and video, is made hard by the lack of proper integration between the visual aspects of the problem (image processing, motion estimation, feature extraction...) and its database aspects (defining visual data as data types in a database). In this paper, we argue that image database languages can be built based on *feature algebras*, and demonstrate how such a feature algebra can be built in the case of one of the most popular image characterization techniques: histograms.

1 Introduction

A primary capability of any database system is to provide a user the means to create, query and manipulate data in a natural, meaningful and expressive way. In traditional database systems [1], this is accomplished by using well-formed mathematical structures (such as sets or trees), and designing a language to create, constrain and manipulate data sets associated with these structures. The language, based on an algebra or calculus, is designed to express most queries that a user is likely to formulate on the mathematical structure for the domains of application. The implicit assumption behind the choice of the mathematical structure and the operators of the language is that they fit naturally to the way a user would think about the data.

Unfortunately, in query and data manipulation languages designed for visual (image and video) database systems, a balance between meaningfulness and expressive power has been hard to achieve. We believe that one primary reason for this difference is that the visual scientist's focus is often targeted toward problem-specific data transformation, while the database scientist's focus is on generic methods to formalize relationships between data, and to access data from well-structured collections. This difference in focus, we contend, has led to an "expressiveness gap" in database languages for visual information systems. We illustrate the problem in the following paragraphs.

*gupta@sdsc.edu

†ssantini@ece.ucsd.edu

1.1 The Expressiveness Gap

Consider a hypothetical database system created by assimilating "best practices" from current visual database research. Let us assume that the system supports both image and video retrieval.

We also start with the assumption that each image or video item is identified by an *id* and $\mathbf{Id}$ denotes the type of all identifiers in the system. Being a characteristic system, it will typically have a set of *global visual features* $\{g_i\}$, computed by a series of image transforms, reducing the original image to a collection of numbers, the so-called *feature vectors* [2]. These feature vectors represent different perceptual or cognitive properties of an image such as color, texture, or camera motion properties of videos, and can be designed using sophisticated techniques to attain properties like invariance to affine transformations and illumination disparities. However, despite the fact that each feature vector is a *collection*, the user does not usually have access to any value "inside" this collection, and has to treat the collection as the instance of an opaque data type $\mathbf{T}_i^g$. In fact, most often the user does not even have access to the *value* of the feature. The type has only one binary operation $\circ_i$: $\mathbf{T}_i^g \times \mathbf{T}_i^g \rightarrow \mathbf{R}$, producing a *distance* between two instances of type $\mathbf{T}_i^g$. Depending on $\mathbf{T}_i^g$, $\circ_i$ may be commutative (e.g., when $\circ_i$ is a *norm* such as L_2), but it need not be (see [3]), and the associativity of the operator is not considered important. Since the distance operation produces a *real* number called *score*, the database system often compares one example image with others in the database and produces $[\mathbf{Id} \times \mathbf{R}]$, a rank-ordered list of $\{id, score\}$ pairs as the result type of a search.

As the system supports n independent global image features, we can assume without loss of generality that these features form a relation. So the collection of images and their properties can be viewed as a relational database with tuples having the type $\mathbf{T}_1^g \times \mathbf{T}_2^g \ldots \mathbf{T}_n^g \times \mathbf{Id}$. The system offers a composite distance between two images by computing a tuple-wise difference. This tuple uses a combination function (Fagin [4, 5] calls it a scoring rule) over the tuple of $\{id, score\}_i$ pairs obtained from each of the n features. Thanks to the wide body of recent research on rules and formal techniques to express and compute this combination function [4, 5, 6, 7] our hypothetical system will have a rich collection of ways to use aggregate ranking functions.

Thus, given the lack of access to the value or the structure of a feature, the system treats individual features as a "black box" with very poor support. However, it provides a wide variety of utilities when the features are put in a relation and when the tuple distance function produces a list of scores.

The situation improves somewhat for *local visual features*. A local visual feature type $\mathbf{T}_i^l$ is defined for an image as a composite type $\mathbf{T}_i^f \times \mathbf{R}^2$, where the first component arises from image transformations and the second component localizes the feature in a region of the image. For local video feature, the composite type is given by $\mathbf{T}_i^f \times \mathbf{R}^2 \times \tau$, where τ stands for the time when the feature occurs in the video[1]. We call these locality components the *spatial support* and the *temporal support* of the feature respectively. Given the composite nature of local visual features, the "support" component of the feature can be projected out. This allows our hypothetical system to characterize the spatial support region into known spatial data types, such as regions, lines and points. Now, the system can provide a rich set of query operations defined on spatial and temporal intervals and on binary topological relations that can be derived from spatial support data. [9, 10, 11]. Again we have the same problem that the *feature* part

[1] Some video operations such as joint color and motion-based object segmentation can provide features with both spatial and temporal support.

of the data is significantly weak and unexpressive compared to the structured portion of the data.

1.2 Contribution of the paper

In this paper, we attempt to show that it is possible to reduce this expressiveness gap by defining mathematical structures for several feature classes in a data and process independent fashion. To this end, we first identify a set of properties that such *feature algebra* will need to satisfy. Then we develop a special case of feature algebra by treating histograms as a generic mathematical structure. We illustrate that this allows us to perform manipulations and express query classes on visual data that could not be expressed unless a feature algebra were defined.

2 Some properties of a Feature Algebra

It is well-recognized that features are typically classified into *global*, *local* and *relational*. A local feature is characterized either by an explicit spatial or temporal support, or by an implicit encoding of spatial (or temporal) information in a globally computed feature (e.g., color correlograms [12]. Relational features are often represented as graphs having atomic attributes or computed features at the nodes and edges. In this section we discuss the structural representation of a generic feature that may be global, the non-support component of a local property, or embedded in the node of a relational attribute.

Definition 2.1 *A* feature *is a collection C of values, containing arbitrarily nested subcollections $S(C)$ indexed by an algebraic structure* $\mathtt{M}$ *defined over* $\mathtt{N}$*, the set of natural numbers.*

Example 1 Structurally, a *vector feature* V can be considered to be a one-dimensional array, and defined as a set of *real* values, indexed by a list of natural numbers, such that $V[i]$ is the i-th element of V.

Example 2 Many complex features such as texture are often represented by a hierarchical bank of filters applied to an image. For example, [13] describes a dual-tree complex wavelet feature tree for texture determination. In this case the $S_i(C)$ is formed by the coefficients of each individual filter bank $C = \cup_i S_i(C)$. $\mathtt{M}$ represents an extended tree-structure such that the node of the tree indexes a filter bank, and, a list within a node specifies a single coefficient within a filter bank.

While the operators in a specific feature algebra must depend on the exact nature of C and $\mathtt{M}$, we can identify a core set of properties that most feature algebras will satisfy. One set of operators is governed by the domain of feature values (e.g., integer, real), and is outside the algebra itself. The algebra would include operators such as the following:

selectCollection This is a *select* operation to choose a specific (sub)collection from a database. This can be accomplished by a predicate on some attribute of the collection such as its name, or cardinality.

pickCollection In case of collections containing subcollections, this operation selects a subcollection a path-expression from the "root" of the collection.

nextSubCollection Applicable for nested collections, this operation relies on the premise that the structure $\mathtt{M}$ provides a *traversal* functionality. Thus, given a subcollection this operator selects the *next* subcollection in the traversal order prescribed by $\mathtt{M}$.

selectElement This is a classical *select* operation of a database system.

pickElement This is a selection by path expression starting from the root of a subcollection containing the element. Typically, the path expression will be prefixed by the path expression leading up to the subcollection.

nextElement Similar to the *nextSubCollection* operation, this operator traverses from one element to the following element.

getElementValue Given an element, this operator returns its value. In some cases this value may be complex. For example, for a 3D object it may return the two 3-vector principal curvature directions of a surface.

compareElement This corresponds to the $\circ_i$: $\mathbf{T}_i^g \times \mathbf{T}_i^g \rightarrow \mathtt{R}$ mentioned before and produces an element distance.

makeCollection The operation creates a new (sub)collection from zero or more existing (sub)collections, by possibly applying a function f to them. An example of such a function could be creating a new histogram by computing a termwise difference of two given histograms.

placeSubCollection This operation positions a newly constructed subcollection into a specific point in the structure of a larger collection. It is based on the premise that the structure $\mathtt{M}$ allows a systematic traverse and insert functionality.

From this generic set of operations we now illustrate a concrete instance of a feature algebra, applied to the case of histograms.

3 The Histogram Algebra

A histogram is a frequency distribution of one or more variables over a set of observations. Without loss of generality, we may state that given any measurement function $f : R \rightarrow \Re^n$, where R is the domain of definition of an image (usually $R \subset \Re^2$) a histogram represents the probability distribution of the values of f. The goal of our algebra is to preserve this probabilistic character of the histogram, instead of treating them like an array [14, 15], where no correlation between different cells of an array can be assumed.

Before describing the histogram data type and the operations on it, we will need a few accessory functions and definitions. The data types *boolean*, *integer*, *real*, as well as arrays of these types are assumed. Integers can be used to form *ranges*, such as $1 : n$. If a is an array, $a[i]$ (or, indifferently, a_i) is its ith element, and $a[1 : n]$ (or $a_{1:n}$) is an array composed of its first n components. Ranges can be k-dimensional, and a range can be assigned to variables of type *range*, but no other operations are

defined on them. The k-dimensional range $\{d_{1i} \leq x_i \leq d_{2i}, i = 1, \ldots k\}$ will be represented using the notation $[d_{1i}, d_{2i}]^k$ where d_{1i} and d_{2i} are one-dimensional arrays.

A probability distribution function is a mapping from real numbers to the interval (0,1) obeying the laws of probability.

The *generation function* $g_p([d_{1i}, d_{2i}]^k, n^k, p_i)$ generates n real numbers in the k-dimensional range $[d_{1i}, d_{2i}]^k$ according to the pdf given by p_i. The number of buckets in each dimension is specified by $n^k \in N^k$.

A *bucketing function* $b_{n,D,\Delta} : \mathbb{R}^k \rightarrow [1 : n]^k$, where D is a range $[d_{1i}, d_{2i}]^k$, and Δ is a k-dimensional array of positive numbers, is a function that maps the k-dimensional range $[d_{1i}, d_{2i}]^k$ into the integer range $[1 : n]^k$. An element of this range is called a *bucket*. The semantics of the bucketing function is as follow. Let $\mathbf{X}$ be a k-dimensional array, and $x_i \in \mathbf{X}$ such that

$$\lfloor \frac{x_i - d_{1i}}{\Delta_i} \rfloor + 1 = h_i \tag{1}$$

then

$$b_{n,D,\Delta}(\mathbf{X}) = [h_1, h_2, \ldots, h_k] \tag{2}$$

Definition 3.1 *A* histogram *is a mapping* $H : [1 : n_i]^k \rightarrow \mathbb{R}^m \cup null$*, where* k *is the* dimension *of the histogram,* n_i *is the* bucket size *along the* i*-th dimension of the histogram, and* m *is the* codomain dimension *of the histogram.*

While by definition the codomain of a histogram is always 1, if two distinct histograms $H1$ and $H2$ have exactly the same dimensions, domain, and bucket sizes, we represent them in a compressed histogram with a codomain 2. To see where such a histogram may be used, consider a 2D edge-orientation histogram with the two dimensions representing the direction of the orientation and the strength of the edge respectively. We may now want to perform a smoothing operation along each of the dimensions, thereby computing at each cell two values. We represent the k-th value in the ij-th cell of the composite 2D histogram H' as $H'[i][j][k]$ as if it had an additional dimension. For a simple (non-composite) 2D histogram, $H'[i][j][k]$ would produce the value err.

An important decision in our algebra is that we strictly enforce dimensionality of histograms. In linear algebra it is quite common to identify a column vector with a matrix with only one column, or a row vector with a matrix with only one row. For that matter, it is possible to see a vector as an array of any dimensionality in which all dimensions but one have only one element. The same identification is possible with histograms: a single dimensional histogram can be seen as a two (or three, or four...)-dimensional histogram with only one bucket in the second direction. This identification is explicitly prohibited in our model. The dimensionality of a histogram is a well-defined attribute irrespective of the number of buckets along any dimension. Two histograms are called *isomorphic* iff they have the same dimensionality, codomain dimensionality, bucket dimensionality along every dimension, and if their domains coincide.

Constructors The following operators build histograms starting from other data types.

Let $\mathbf{A}$ be a k-dimensional array. The operator $build(\mathbf{A})$ constructs a k-dimensional histogram with codomain dimension equal 1 such that $H(i_1, \ldots, i_n) = \mathbf{A}[i_1, \ldots, i_n]$. The operator $build(v, i_1, \ldots, i_k)$

builds a k-dimensional histogram with co-domain dimension equal to the dimension of the array v, and i_i buckets along the ith dimension. The histogram is initialized to the map

$$H(j_1, \ldots, j_k) = \begin{cases} v & \text{if } j_1 = i_1 \wedge j_2 = i_2 \wedge \cdots \wedge j_k = i_k \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

We will also consider two special operators. The first, G_p, constructs a histogram based on a given function. Its semantics is

$$G_p([d_1, d_2]^k, f, n^k) = build(g_p([d_1, d_2]^k, n^k, f)) \tag{4}$$

The operator $null(n_1, \ldots, n_k)$ builds a k-dimensional histogram with n_i buckets along the ith dimension implementing the null mapping.

Histogram functions The function $dim(H)$ returns the dimension of the histogram.
The function $dom_i(H)$ returns the domain of the histogram in the i-th dimension.
The macro $Dom(H)$ is a program that returns the complete domain of the histogram.
The function $size_i(H)$ returns the number of buckets in the histogram for the i-th dimension.
The macro $Size(H)$ is a program that returns an array $[n_1, \ldots, n_k]$ denoting the size of all buckets in the histogram.
The function $val(H[i1, \ldots ik])$ returns the value of the bucket at the specified index. For conveneience, we also define the predicate $isval(H[i1, \ldots ik] = const)$, which evaluates to $true$ iff $val(H[i1, \ldots ik]) = const$
The macro $TotCount(H)$ computes the sum of $val(H[i1, ...ik])$ over all buckets.
The macro $bounds(H[i1, ...ik])$ returns a k-tuple of pairs $[[u1, l1], \ldots [uk, lk]]$ where $[uj, lj]$ are the upper and lower bounds on the domain of the bucket $[i1, ...ik]$ for the j-th dimension.
The macro $norm(H)$ normalizes a histogram so that, if $G = norm(H)$

$$\sum_{i_1, \ldots, i_k} H(i_1, \ldots i_k) = 1 \tag{5}$$

and, for all the values for which the operation is defined

$$\frac{G(i_1, \ldots, i_k)}{G(j_1, \ldots, j_k)} = \frac{H(i_1, \ldots, i_k)}{H(j_1, \ldots, j_k)} \tag{6}$$

Same size operators The following operators combine two histograms of the same dimension, codomain dimension and bucket size. The operators are undefined when applied to histograms that differ on any of these dimensions.

The operator $+$ denotes addition of two histograms. $H_1 + H_2$ has the following semantics:

$$(H_1 + H_2)(i_1, \ldots, i_k) = H_1(i_1, \ldots, i_k) + H_2(i_1, \ldots, i_k) \tag{7}$$

The operator $-$ denotes absolute value difference of two histograms. $H_1 - H_2$ has the following semantics:

$$(H_1 + H_2)(i_1, \ldots, i_k) = |H_1(i_1, \ldots, i_k) H_2(i_1, \ldots, i_k)| \tag{8}$$

These operators are special cases of the general *element-wise combination* operators $\langle\cdot\rangle$ defined as follows. Let $f : \mathbb{R}^m \times \mathbb{R}^m \to (\mathbb{R}^+)^m$ be a symmetric and associative operator. Then, for histograms H_1 and H_2 with m-dimensional co-domain, $H_1\langle f\rangle H_2$ has the following semantics:

$$(H_1\langle f\rangle H_2)(i_1,\ldots,i_k) = f(H_1(i_1,\ldots,i_k), H_2(i_1,\ldots,i_k)) \tag{9}$$

The operator = denotes assignment and has the usual semantics that $H1[i1,\ldots,ik] = C$, where C is a constant, means the predicate $isval(H1[i1,\ldots,ik] = C) = true$

Field operators These operators combine a histogram and a real number. The unary operator $\cdot$ denotes scalar multiplication by a positive scalar constant. The operator is not defined for $c < 0$. Thus $c \cdot H$ (abbreviated cH) has the semantics:

$$(c \cdot H)(i_1,\ldots,i_k) = cH(i_1,\ldots,i_k) \tag{10}$$

Similarly to the previous case, this operator is a special case of the general operator $\langle\cdot\rangle$, defined as follows. Let $f : \mathbb{R} \times \mathbb{R}^m \to (\mathbb{R}^+)^m$ a function, then the semantics of $c\langle f\rangle H$ is

$$(c\langle f\rangle H)(i_1,\ldots,i_k) = f(c, H(i_1,\ldots,i_k)) \tag{11}$$

Cross Dimensional Operators These operators change the dimensionality or the number of buckets of a histogram.

The *expand* operator increases the dimensionality of a histogram. Its signature is $G = expand(H, n, a)$, where H is a k-dimensional histogram, a is a k-dimensional vector of integers such that $a_i \in [1, n]$ and $i \neq j \Rightarrow a_i \neq a_j$, and the result is an n-dimensional histogram.

The formal specification of the operator (see below) is rather involved, but its semantics is actually quite simple. For example, consider a two dimensional histogram H. Then the operation $G = expand(H, 3, [3, 1])$ builds a three dimensional histogram such that the first dimension of H becomes the third dimension of G, the second dimension of H becomes the first dimension of G, and all other dimensions of G (in this case the second dimension) have only one bin. In this case, the relation between G and H is:

$$size_1(G) = size_2(H) \tag{12}$$
$$size_2(G) = 1 \tag{13}$$
$$size_3(G) = size_1(H) \tag{14}$$
$$G(i, 1, j) = H(j, i) \tag{15}$$

In order to define the behavior of the operator in more general circumstances, we need a few auxiliary definitions. Let $I = [i_1,\ldots i_k] \in N^k$ be a k-dimensional index, and $J = [j_1,\ldots,j_n] \in N^n$ an n-dimensional index. Let $u_a : N^k \to N^n$ be the transformation defined as

$$[u_a(J)]_i = \begin{cases} j_h & \text{if } a_h = i \\ 1 & \text{otherwise} \end{cases} \tag{16}$$

i	1	2	3	4	5	6	7	8
$f(i)$	(1,1)	(1,2)	(2,1)	(3,1)	(2,2)	(1,3)	(1,4)	(2,3)
i	9	10	11	12	13	14	15	16
$f(i)$	(3,2)	(1,4)	(2,4)	(3,3)	(4,2)	(4,3)	(3,4)	(4,4)

Table 1:

Consider now the set of n-indices that have a "1" in the locations not covered by the vector a: $\mathcal{I}_a = \{[j_1, \ldots, j_n] \in N^n | \ \nexists h : a_h = q \Rightarrow j_q = 1\}$. Then the transformation u_a is an isomorphism between N^k and $\mathcal{I}_a$ and therefore invertible on this domain: $u^{-1} : \mathcal{I}_a \rightarrow N^k$. The relation between the histograms H and G can then be defined as follows:

$$\forall I \in \mathcal{I}_a \ \ G(I) = H(u_a^{-1}(I)) \tag{17}$$

The *embed* operator substitutes part of a larger histogram with values from a smaller histogram with the same dimensionality, starting at a location $I = [i_1, \ldots, i_n]$ in the larger histogram. The operator signature is $Q = \varepsilon_I(H, G)$, where H is embedded into G. If for some j we have $i_j \leq 0$ or $i_j + size_j(H) > size_j(G)$, the histogram H will be clipped, and only the values with indices within the legal range of G will be used. The semantics of the operator is

$$(\varepsilon_I(H, G))([j_1, \ldots, j_n]) = \begin{cases} H([j_1 - i_1, \ldots, j_n - j_n]) & \text{if } \forall h \ 0 < j_h - i_h \leq size_h(H) \\ G([j_1, \ldots, j_n]) & \text{otherwise} \end{cases} \tag{18}$$

The *select* operator $\sigma_\theta(H)$, selects those bins of histogram H, that satisfy the predicate θ. Its semantics is given by:

$$\sigma_\theta(H) = \begin{cases} H[i] & \text{if } \theta(H[i]) = true \\ null & \text{otherwise} \end{cases} \tag{19}$$

The *project* operator $\pi_{h,\oplus}(H)$ takes a symmetric associative operator $\oplus : \mathbb{R}^m \times \mathbb{R}^m \rightarrow (\mathbb{R}^+)^m$ and uses it to compress the hth dimension of the histogram H. The semantics of the operator is the following

$$(\pi_{h,\oplus}(H))(i_1, \ldots, i_{h-1}, i_{h+1}, \ldots, i_n) = \bigoplus_{j=1}^{size_h(H)} H(i_1, \ldots, i_{h-1}, j, i_{h+1}, \ldots, i_n) \tag{20}$$

Next, the *traversal* operator $T_f(H)$ takes as an argument an index transform $f : N^k \rightarrow N^n$ ($n \geq k$) and uses it to reduce the dimensionality of the histogram H and transform its indices

$$(T_f(H))(I) = H(f(I)) \tag{21}$$

where $I \in N^k$ is an index of the new histogram. As an example, consider the two dimensional histogram of Fig. 1 and the index transform defined by Table 1, corresponding to the traversal of Fig. 1.b. The traversed histogram is shown in Fig. 1.c.

Finally, the *rebucket* operator changes the number of buckets along all the dimensions of a histogram re-distributing the data inside a bucket according to a given probablity density. Consider, for instance,

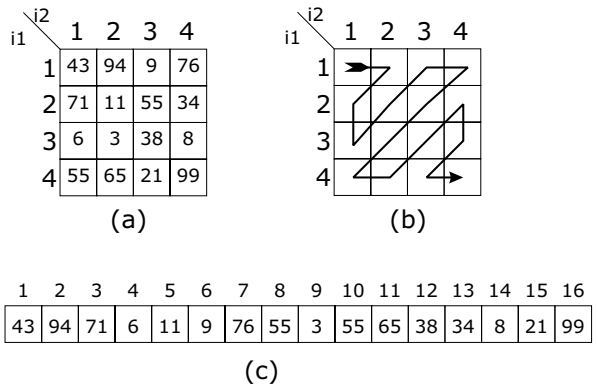

Figure 1: The traversal operator.

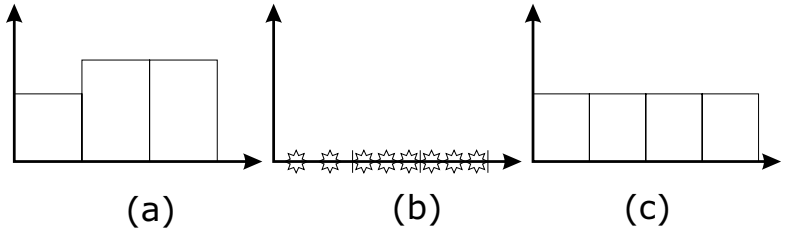

Figure 2: An example of re-bucketing

the one-dimensional histogram in Fig. 2, The histogram has three buckets, and we want to expand it to four using a uniform underlying probability. The rebucket operator works as follows:

1. Transform the histogram into a statistical sampling using the existing buckets and the given probability. In other words: the histogram was obtained by sampling a probability distribution and, according to the result, there were two samples in the interval covered by the first bucket, three samples in the interval covered by the second bucket, and three samples in the interval covered by the third bucket. Inside the buckets, the samples are distributed according to the given distribution (uniformly, in this case).

2. The statistical distribution is re-sampled with the new number of buckets, four in this case.

The operator has the form $b(H, q, p)$, where H is a n-dimensional histogram, $q \in N^n$ specifies the number of buckets in the final histogram, and p is a probability density function $p : \mathcal{H} \times N^n \times \Re^n \rightarrow [0, 1]$ that specifies the distribution of the samples inside a bucket. Note that p can depend on the histogram and the index of the bucket that we are expanding.

4 Computing with the Histogram Algebra

We now illustrate how the algebra can be used to compute useful operations.

Sum of squares of an histogram

$$sq(H) = S(H\langle\cdot\rangle H) \tag{22}$$

where

$$S(H) = \begin{cases} \pi_{1,+}(H) & \text{if } dim(H) = 1 \\ S(\pi_{1,+}(H)) & \text{otherwise} \end{cases} \tag{23}$$

The recursion is well defined since for all histograms H $dim(\pi_{h,\oplus}(H)) = dim(H) - 1$.

Computation of the L_2 distance between histograms

$$L_2(H_1, H_2) = \sqrt{\text{sq}(H_1 - H_2)} \tag{24}$$

Computation of the Hue histogram from the RGB histogram We assume that the functions $h(r, g, b)$, $s(r, g, b)$ $v(r, g, b)$ transform an r, g, b color into the corresponding hsv color. From this function, given a histogram with n bins on each color axis, a index for the corresponding hsv color can be computed by the function

$$\iota[i, j, k] = n \cdot \left[h\left(\frac{i}{n}, \frac{j}{n}, \frac{k}{n}\right), s\left(\frac{i}{n}, \frac{j}{n}, \frac{k}{n}\right), v\left(\frac{i}{n}, \frac{j}{n}, \frac{k}{n}\right) \right] \tag{25}$$

The hues histgram can then be computed as

$$hue(H) = \pi_{3,+}(\pi_{2,+}(T_\iota(H))) \tag{26}$$

Answering queries The algebra provides a powerful tool for the specification of queries in interactive systems. Due to the greater complexity of these examples, we will assume that the histogram algebra is expressed in a suitable programming language. In particular, we will write the functions using the ML programming language [16]. The reasons why ML was chosen are its powerful handling of functions, including support for *currying*. In ML, a function f that takes an integer and returns an integer (e.g. `fun f(x:int) = x`) is a first class object of type `int -> int`. On the other hand, a function defined as `fun x y = x*y`, where all the variables are integers, is a curried function of type `int -> (int -> int)` that is, the function takes an integer value (`x`) and produces a function that takes the value `y` and returns an integer. In other words, the expression `f(x)` is a function of type `int -> int`.

The rest of the ML syntax used in the following is rather intuitive, and should make the examples understandable also to readers not familiar with ML.

Example 1."Find all the histograms that behave like the function f in the interval $[I, J]$". A similarity measure for this query is given by

```
- fun D(f, F, I, J) =
   let
      hist1 = σ_{I≤x≤J} ( G_p (Dom(H), f, Size(H)));
      hist2 = σ_{I≤x≤J} (H);
   in
      L_2 (hist1 - hist2)
   end;
```

Example 2. "Find all peaks of a histogram". The peaks are to be determined using the following rules: first the histogram is filtered with a given kernel $K = [w_{-m}, \ldots, w_0, \ldots, w_m]$, then the maxima are detected. Points adjacents to a peak are not considered peaks. For the sake of simplicity, we will only consider one-dimensional histograms.

We begin with the definition of two functions that shift a histogram by an amount I, where I is an index. The first function pads the shifted portion with zeros:

```
- fun shft1(H, I) =
   ε_I (null(Size(H)), H)
```

The second function rotates the histogram, as if it were a periodic function

```
- fun shft2(H, I) =
   ε_I (null(Size(H)), H) + ε_{-I mod Size(H)} (null(Size(H)), H).
```

The filter operator, with kernek K is defined as follows:

```
- fun F(K, m, H) =
   let
      fun Ft(K, m, H) =
         if m = 1
            k[1] * H
         else
            (k[1] * H) + Ft(k[2:m], m-1, shft2(H, 1))
   in
      Ft(K, 2*m+1, shft2(H, -m))
   end;
```

Note that in this case we use the second shift operator that is, we consider the histogram as a periodic function. If this is undesired, the histogram can be embedded into a null histogram of size $Size(H)+2m$ and the first shift operator can be used without losing data.

The peaks will be returned in the form of a histogram that is zero everywhere except in the peak locations, where it has value 1. We will use the indicator function $\delta(a, b)$ whose value is 1 if $a = b$ and

zero otherwise. The following operator takes a histogram H and returns a histogram with all bins set to zero except where H attains its maximum:

```
- fun M(H) =
   let
      fun f(H)(x) = π_{1,max}(H)
   in
      H ⟨δ⟩ G_p (Dom(H), f(H), Size(H))
   end;
```

The following auxiliary function takes an histogram H and a histogram G with the same dimension and bucket dimension as H. For every bin in G with a nonzero value creates a three bins in H with zero value. In other words, the function creates "holes" in H of size 3 corresponding to the values in G.

```
- fun Q(H, G) =
   let
      fun f(x) = 1;
      val h1 = G_p (Dom(H), f, Size(H)) - G;
      val h2 = G_p (Dom(H), f, Size(H)) - shft1(G, 1);
      val h3 = G_p (Dom(H), f, Size(H)) - shft(G, -1);
   in
      H * (h1 + h2 + h3)
   end;
```

Finally, the function P, finds the n highest peaks in the histogram

```
- fun P(H, n, K, m) =
   let
      fun Pt(H, n) =
         if n = 1
            M(H)
         else
            M(H) + P(Q(H, M(H)), n-1)
   in
      Pt(F(K, m, H), n)
   end;
```

Example 3. "Find whether, at a specific point in time t of a video sequence, the dominant motion in the image is reversed (change of approximately 180 degrees) in less than three frames". We are looking for the event that the motion is reversed sometimes between t and $t + 3$.

The data are stored in a relational database whose schema comprises just one table:

$$MOTION(t : int, h : histogram) \tag{27}$$

where t is the time at which the motion is considered, and h is a histogram counting how many points are moving in a given direction.

We begin by writing a function that determines whether, given two histograms H_1 and H_2, there is a inversion of motion between the two. The function works as follows (see Fig. 3). We first take, for

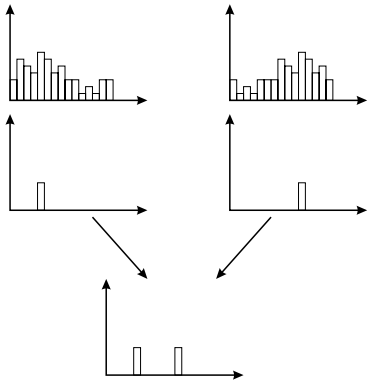

Figure 3: First steps of the determination of motion inversion between two histograms.

each histogram, the highest peak, which is representative of the dominant motion, then we add the two histograms to obtain a single histogram with two peaks. Using the function P defined above, we can write this histogram as

$$P(H_1, 1) + P(H_2, 1) \tag{28}$$

We then proceed as illustrated in Fig. 4. The histogram of the two major peaks, represented in

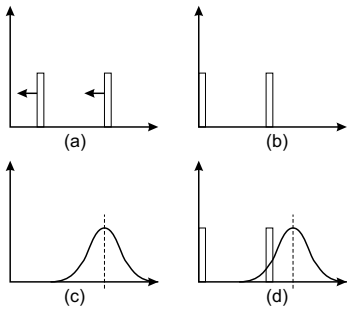

Figure 4: Determination of the distance of two peaks of a histogram

Fig. 4.a is shifted until the lowest peak is on the first bin (Fig. 4.b). Then, a "gauge" histogram is created using a Gaussian function centered at a bin corresponding to the distance at which we want to check the presence of a peak (the Gaussian allows us to tolerate slight misplacements of the second peak, and to control the extent to whhich these displacements can be accomodated), as in Fig. 4.c. Finally, the

Gaussian histogram and the peak histogram are multiplied, giving us a measure of the presence of a second peak in the desired position. The shift function Sh uses the function $shft2$ defined above:

```
- fun Sh (H) =
   let
      fun f (x) = if x = 0 then 1 else 0;
      val cond = proj(1, max, H * G( Dom(H), f, Size(H))) != 0;
   in
      if cond then H else Sh( shft2(H) )
   end;
```

The following function checks if the histogram has a peak around the position y:

```
- fun Pn(H, y) =
   let
      fun f y x = Gauss(x - y, sigma);
      val hist = H * Gp(Dom(H), f(y), Size(H))
   in
      π1,max (hist)
   end;
```

The function returns a value between 0 and 1 representing the confidence that the histogram has a peak in the given position. A threshold can be applied if a hard decision is needed. With these functions, we can write the function `MInv(H1, H2)` that returns a number between 0 and 1 representing the confidence that a motion inversion takes place between the histograms `H1` and `H2`:

```
- fun MInv(H1, H2) =
   let
      val H = Sh(P(H1, 1) + P(H2, 1));
      val y = (dom(H, 1).up - dom(H, 1).lo)/2;
   in
      Pn(H, y)
   end;
```

With this function, it is possible to formulate the query in the database as:

```
SELECT  r, s
FROM    MOTION
WHERE ABS(r.t - s.t) <= 3 AND MInv(r.H, s.H) > 0.5
```

5 Summary and Outlook

In this paper, we have argued that image database languages can be built based upon feature algebras and demonstrated how data manipulations and queries can be executed with a feature algebra in terms of histograms. While in this paper we have not formalized the exact query classes expressible in terms of this algebra, the peak detection and motion inversion examples, suggest that when embedded in a functional language, the algebra can express more complex queries than possible with current systems. We plan to investigate the expressiveness properties of histogram algebra in the future, and expect that it will provide us some insight into the desired properties of a more general feature algebra.

References

[1] S. Abiteboul and R. Hull and V. Vianu, *Foundations of Databases*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1995.

[2] A. Gupta, "Visual Information Retrieval: a Virage Perspective", *Technical Report*, Virage, Inc., 1995.

[3] S. Santini and R. Jain, "Similarity Measures," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Sept. 1999

[4] R. Fagin and E.L. Wimmers, "A Formula for Incoporating Weights into Scoring Rules", *Theoretical Computer Science* (to appear). Modified from "Incorporating user preferences in multimedia queries", in *Proc. 6th International Conference on Database Theory*, Delphi, Jan. 1997, Springer-Verlag Lecture Notes in Computer Science 1186, ed. F. Afrati and Ph. Kolaitis, 247-261.

[5] R. Fagin, "Combining Fuzzy Information from Multiple Systems", *J. Computer and System Sciences*, 58, 1999, 83-99.

[6] S. Adali and P. Bonatti and M.L. Sapino and V. Subrahmanian, "A Multi-Similarity Algebra", in *Proc. ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington*, 402-413.

[7] S. Nepal and M.V. Ramakrishna, "Query Processing Issues in Image (Multimedia) Databases", in *Proc. 15th International Conference on Data Engineering*, 23-26 March 1999, Sydney, Austrialia, 22-29.

[8] R.M. Haralick and L.G. Shapiro *Computer and Robot Vision, Volume I*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.

[9] J.Z. Li and M.T. Ozsu and D. Szafron and V. Oria, "MOQL: a multimedia object query language". In Proc. *Third International Workshop on Multimedia Information Systems*, Como, Italy, September 1997, 19-28.

[10] W.-S. Li and K.S. Candan and K. Hirata and Y. Hara, "Hierarchical Image Modeling for Object Based Media Retrieval", *Data and Knowledge Engineering*, 27, 1998, 139-176.

[11] Y.F. Day and Al-Khatib Wasfi and R.Paul and A. Ghafoor, “Specification of a query language for multimedia database systems”, in *Proceedings International Workshop on Multimedia Software Engineering*, Kyoto, Japan, 20-21 April 1998, 111-118.

[12] J. Huang and S. Ravi Kumar and M. Mitra and W-J. Zhu and R. Zabih, “Image Indexing Using Color Correlograms”, In. *Proc. IEEE Computer Vision and Pattern Recognition Conference*. San Juan, Puerto Rico, June 1997.

[13] S. Hatipoglu and S.K. Mitra and N. Kingsbury, “Texture classification using dual-tree complex wavelet transform”, In Proc. 7th International Congress on Image Processing and its Applications, Manchester, UK, July 1999, 344-347.

[14] Leonid Libkin and Rona Machlin and Limsoon Wong, “A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques,” *ACM SIGMOD, 1997* 228-239

[15] Arunprasad P. Marathe and Kenneth Salem, “A Language for Manipulating Arrays” *VLDB 1997* 46-55

[16] Jeffrey D. Ullman, *Elements of ML Programming*, 2nd Edition. Prentice-Hall, 1997.