FeedbackBypass: A New Approach to Interactive Similarity Query Processing

Ilaria Bartolini Paolo Ciaccia

DEIS – CSITE-CNR, University of Bologna
Bologna, Italy

{*ibartolini,pciaccia*}*@deis.unibo.it*

Florian Waas

Microsoft Corp.
Redmond, USA

florianw@microsoft.com

Abstract

In recent years, several methods have been proposed for implementing interactive similarity queries on multimedia databases. Common to all these methods is the idea to exploit user feedback in order to progressively adjust the query parameters and to eventually converge to an "optimal" parameter setting. However, all these methods also share the drawback to "forget" user preferences across multiple query sessions, thus requiring the feedback loop to be restarted for every new query, i.e. using default parameter values. Not only is this proceeding frustrating from the user's point of view but it also constitutes a significant waste of system resources.

In this paper we present FeedbackBypass, a new approach to interactive similarity query processing. It complements the role of relevance feedback engines by storing and maintaining the query parameters determined with feedback loops over time, using a wavelet-based data structure (the Simplex Tree). For each query, a favorable set of query parameters can be determined and used to either "bypass" the feedback loop completely for already-seen queries, or to start the search process from a near-optimal configuration.

FeedbackBypass can be combined well with all state-of-the-art relevance feedback techniques working in high-dimensional vector spaces. Its storage requirements scale linearly with the dimensionality of the query space, thus making even sophisticated query spaces amenable. Experimental results demonstrate both the effectiveness and efficiency of our technique.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

1 Introduction

Similarity and distance-based queries are a powerful way to retrieve interesting information from large multimedia repositories [Fal96]. However, the very nature of multimedia objects often complicates the user's task of choosing an appropriate query and a suitable distance criterion to retrieve from the database the objects which best match his/her needs [SK97]. This can be due both to limitation of the query interface and to the objective difficulty, from the user's point of view, to properly understand how the retrieval process works in high-dimensional spaces, which typically are used to represent the relevant *features* of the multimedia objects. For instance, the user of an image retrieval system will hardly be able to predict the effects that the modification of a single parameter of the distance function used to compare the individual objects can have on the result of a query.

To obviate this unpleasant situation, several multimedia systems now incorporate some *feedback* mechanisms so as to allow users to provide an evaluation of the *relevance* of the result objects. By analyzing such relevance judgments, the system can then generate a new, refined query, which will likely improve the quality of the result, as experimental evidence confirms [RHOM98]. This interactive retrieval process, which can be iterated several times until the user is satisfied with the results, gives rise to a *feedback loop* during which the default parameters used by the query engine are gradually adjusted to fit the user's needs (see e.g. [ORC$^+$97]).

Although relevance feedback has been recognized as a highly effective tool, its applicability suffers from two major problems:

1. Depending on the query, numerous iterations might occur before an acceptable result is found, thus convergence can be slow.

2. Once the feedback loop of a query is terminated, no information about this particular query is retained

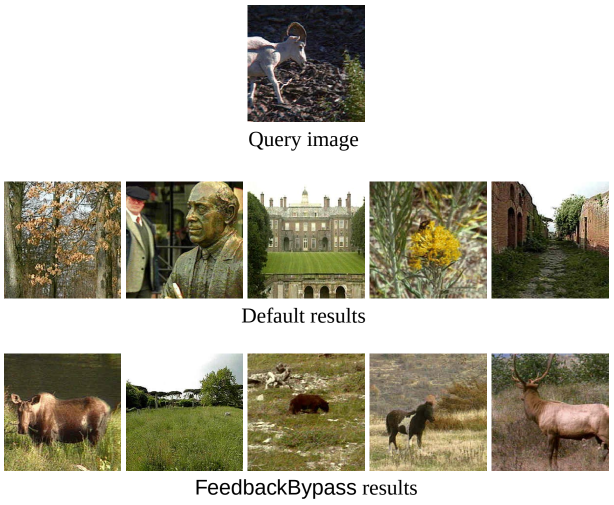


Figure 1: FeedbackBypass in action. The middle line shows the 5 best matches computed using default parameters for the query image on top. The bottom line shows the results obtained for the same query when the parameters suggested by FeedbackBypass are used

for re-use in further processing. Rather, for further queries, the feedback process is started anew with default values. Even in the case where a query object has already been used in an earlier feedback loop, all iterations have to be repeated.

Note that both problems concern the *efficiency* of the feedback process, whereas the *effectiveness* of retrieval will depend on the specific feedback mechanisms used by the system, on the similarity model, and on the features used to represent the objects.

This paper presents FeedbackBypass, a new approach to interactive similarity query processing, which complements the role of current relevance feedback engines. FeedbackBypass is based on the idea that by storing and maintaining the information on query parameters gathered from past feedback loops it is possible to either "bypass" the feedback loop completely for already-seen queries or to "predict" near-optimal parameters for new queries. In both cases, as an overall effect, the number of feedback and database search iterations is greatly reduced, thus resulting in a significant speed-up of the interactive search process.

Figure 1 shows a query image together with the 5 best results obtained from searching with default parameters a data set of about 10,000 color images. No result image belongs to the same semantic category "Mammal" of the query image (see Section 5 for a description of image categories). The bottom line of the figure shows the 5 best matches obtained for the same query when FeedbackBypass is applied and the system uses the predicted query parameters. With FeedbackBypass, we obtain 4 relevant images (i.e. 4 mammals) among the top 5 query results.

The implementation of FeedbackBypass utilizes a novel wavelet-based data structure, called *Simplex Tree*, whose storage overhead is linear in the dimensionality of the query space, thus makes even sophisticated query spaces amenable. Its resource requirements are *independent* of the number of processed queries but depend only on the complexity of the query parameter function, which guarantees proper scalability and performance levels. Furthermore, storage requirements can be easily traded-off for the accuracy of the prediction. Experimental results presented below demonstrate both the effectiveness and efficiency of our technique.

The rest of the paper is organized as follows. Section 2 provides the background on relevance feedback mechanisms and on related work. In Section 3 we describe our approach and state the basic requirements for an effective implementation of FeedbackBypass. Section 4 provides a thorough description of the Simplex Tree and of related implementation issues. Experimental results on a real-world image data set are presented in Section 5. Section 6 concludes the paper.

2 Background and Related Work

We frame our discussion within the context of *vector space* similarity models, for which a multimedia object is represented by a D-dimensional vector (i.e. a point in $\Re^D$) of *features*, $\mathbf{p} = (p_1, \ldots, p_D)$. The similarity of two points $\mathbf{p}$ and $\mathbf{q}$ is measured by means of some *distance function* on such space. Relevant examples of distance functions include L_p norms (L_1 is the Manhattan distance, L_2 is the Euclidean distance) and their weighted versions. For instance, the weighted Euclidean distance is

$$L_{2W}(\mathbf{p}, \mathbf{q}; \mathbf{W}) = \left(\sum_{i=1}^{D} w_i \, (p_i - q_i)^2 \right)^{1/2} \qquad (1)$$

Also *quadratic* distances can be used to capture correlations between different coordinates of the feature vectors. The well-known Mahalanobis distance is defined as

$$d^2_{Mahalanobis}(\mathbf{p}, \mathbf{q}; \mathbf{W}) = \sum_{i=1}^{D} \sum_{j=1}^{D} w_{i,j} (p_i - q_i)(p_j - q_j)$$

and leads to arbitrarily-oriented ellipsoidal iso-distant surfaces in feature space [SK97]. Note that this distance is indeed a "rotated" weighted Euclidean norm.

The typical interaction with a multimedia retrieval system that implements relevance feedback mechanisms can be summarized as follows [Sal88]:

Query formulation. The user submits an initial query $Q = (\mathbf{q}, k)$, where $\mathbf{q}$ is called the *query point* and k is a limit on the number of results to be returned by the system.

Query processing. The query point $\mathbf{q}$ is compared with the database objects by using a (default) distance function d. Then, the k objects which are closest to $\mathbf{q}$ according to d, $Result(Q, d) = \{\mathbf{p}_1, \ldots, \mathbf{p}_k\}$, are returned to the user.

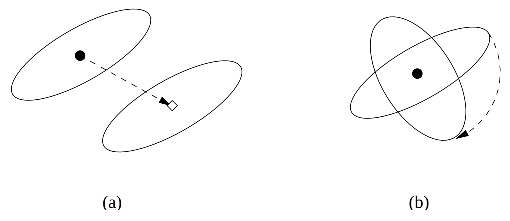


Figure 2: The "query point movement" (a) and the "re-weighting" (b) feedback strategies

Feedback loop. The user evaluates the relevance of the objects in $Result(Q, d)$ by assigning to each of them a *relevance score*, $Score(\mathbf{p}_j)$. On the basis of such scores a new query, $Q' = (\mathbf{q}', k)$, and a new distance function, d', are computed and then used to determine the second round of results.

Termination. After a certain number of iterations, the loop ends the final result being $Result(Q_{opt}, d_{opt})$, where $Q_{opt} = (\mathbf{q_{opt}}, k)$ is the "optimal" query the user had in mind, and d_{opt} the "optimal" distance function to retrieve relevant objects for Q_{opt}.

Every interactive retrieval system provides a specific implementation for each of the above steps. For instance, the choice of the initial query point depends on the system interface and, also considering the nature of the multimedia objects, can include a *query-by-sketch* facility, the choice from a random sample of objects, the upload of the query point from a user's file, etc. A number of options is also available for implementing the actual query processing step, which typically exploits index structures for high-dimensional data, such as X-trees [BKK96] and M-trees [CPZ97].

More relevant to the present discussion are the issues concerning the feedback loop. The use of *binary* relevance scores is the simplest one, even from the user's point of view. In this case the user can mark a result object either as "good" or "bad", and implicitly assigns a neutral ("no-opinion") score to non-marked objects. Graded, and even continuous, score levels have also been used to allow for a finer tuning of user's preferences [RHOM98].

The two basic strategies for implementing the feedback loop concern the computation of a new query point (*query point movement*) and the change of the distance function, which can be accomplished by modifying the weights (importance) of the feature components (*re-weighting*).

Query point movement. The idea behind this strategy is to try to move the query point towards the "good" matches (as evaluated by the user), as well as to move it far away from the "bad" result points (see Figure 2 (a)). A well-known implementation of this idea dates back to Rocchio's formula [Sal88], which has been successfully deployed in the context of document retrieval. More recently, query point movement has been applied by several image retrieval systems, such as the MARS system [RHOM98]. Ishikawa et al. [ISF98] have proved that, when using *positive* feedback (scores) and the Mahalanobis distance, the "optimal" query point (based on the available set of results) is a weighted average of the good results, i.e.:

$$\mathbf{q}' = \frac{\sum_j Score(\mathbf{p}_j) \times \mathbf{p}_j}{\sum_j Score(\mathbf{p}_j)} \tag{2}$$

Re-weighting. The idea of re-weighting stems from the observation that user feedback can help identify feature components that are more important than others in determining whether a result point is "good" or not. Consequently, such components should be given a higher relevance. For simplicity of exposition, let us consider a retrieval model based on weighted Euclidean (see Equation 1) and also refer to Figure 2 (b). In order to assess the relative importance of the i-th feature vector component, the distribution of the "good" $p_{j,i}$ values, i.e. the values of the good matches along the i-th coordinate, is analyzed. In an earlier version of the MARS system [RHOM98], it was proposed to assign to the i-th coordinate a weight w_i computed as the inverse of the standard deviation of the $p_{j,i}$ values, i.e. $w_i = 1/\sigma_i$. Later on, it was proved in [ISF98] that the "optimal" choice of weights is to have $w_i \propto 1/\sigma_i^2$. Similar results have been proven for quadratic distance functions [ISF98], as well as for the case where the number of good matches is less than the dimensionality of the feature space [RH00].

In a recent paper [RH00] Rui and Huang have extended the re-weighting strategy to a "hierarchical model" of similarity, where above strategy is first separately applied to each individual feature, and then each feature (rather than each feature component) is assigned a weight which takes the overall distance into account that good matches have from the query point by considering only that very feature. Note that for F features this amounts to define the distance between objects $\mathbf{p}$ and $\mathbf{q}$ as a weighted sum of feature distances, each of which the authors assume to have a quadratic form [RH00].

3 The FeedbackBypass Approach

The basic idea of our approach is to "bypass", or at least to reduce, the loop iterations to be performed by an interactive similarity retrieval system by trying to "guess" what the user is actually looking for, based only on the initial query he/she submits to the system.

If we abstract from the specific differences existing between the systems and concentrate on what all such systems share, two important observations can be made:

1. All systems assume that the user has in mind an "optimal" query point as well as an "optimal" distance function for that query.

2. Each time a new distance function is computed, this is taken from a *parameterized class* of functions (e.g. the class of weighted Euclidean distances), by appropriately setting the values of the class parameters.

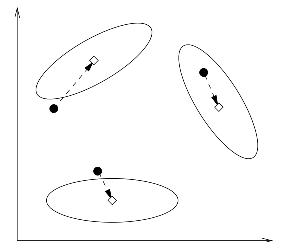


Figure 3: The optimal query mapping for 3 sample query points, assuming Mahalanobis distance

This general state of things can be synthetically represented as a mapping:

$$\mathbf{q} \mapsto (\mathbf{q_{opt}}, d_{opt}) \equiv (\mathbf{\Delta_{opt}}, \mathbf{W_{opt}}) \tag{3}$$

which assigns to the initial query point $\mathbf{q}$ an optimal query point, $\mathbf{q_{opt}}$, and an optimal distance function, d_{opt}. The equivalence just highlights that d_{opt} is the distance function obtained when the parameters are set to $\mathbf{W_{opt}}$, and that $\mathbf{q_{opt}}$ can be obtained from the initial query point by adding to it the "optimal offset" $\mathbf{\Delta_{opt}} = \mathbf{q_{opt}} - \mathbf{q}$. In the following we refer to the pair $(\mathbf{\Delta_{opt}}, \mathbf{W_{opt}})$ as the *optimal query parameters*, OQPs, of query $\mathbf{q}$. Figure 3 provides an intuitive graphical representation of the above mapping for three 2-dimensional query points.

FeedbackBypass is based on the observation that, as more and more query points are added, an "optimal" *query mapping*, M_{opt}, from query points to query points and distance functions, will take shape, and that "learning" such mapping can indeed lead to "bypass" the feedback loop.

Let $\mathcal{Q} \subseteq \Re^D$ be the domain of query points and let $\mathcal{W} \subseteq \Re^P$ be the set of possible parameter choices, where each $\mathbf{W} \in \mathcal{W}$ corresponds to a distance function in the considered class and P is the number of independent parameters that characterize a distance function. Then, the problem faced by FeedbackBypass can be precisely formulated as follows:

Problem 1 *Given the $\mathcal{Q}$ query domain and a class of distance functions with set of parameters $\mathcal{W}$, "learn" the query mapping $M_{opt} : \mathcal{Q} \rightarrow \Re^D \times \mathcal{W}$ which associates to each query point $\mathbf{q} \in \mathcal{Q}$ the optimal query parameters $(\mathbf{\Delta_{opt}}, \mathbf{W_{opt}}) = M_{opt}(\mathbf{q})$.*

In other terms, the problem can be described as that of learning the optimal way to map (query) points of $\Re^D$ into points of $\Re^{D+P}$. It should be remarked that when query points are normalized, the dimensionality of both the input (feature) and the output space of M_{opt} can be reduced by 1.

Of course, statistical techniques for *dimensionality reduction* could be applied to lower the dimensionality of both the input and the output space. We do not consider dimensionality reduction in this paper, and leave it as an interesting follow-up of our research.

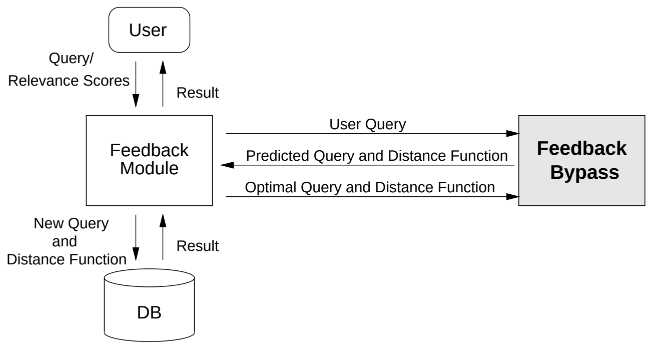

User
Query/
Relevance Scores
Result
Feedback Module
User Query
Predicted Query and Distance Function
Optimal Query and Distance Function
Feedback Bypass
New Query and Distance Function
Result
DB

Figure 4: An interactive retrieval system enriched with the FeedbackBypass module

Example 1 Assume that objects are color images, which are represented by using a 32-bins color histogram, and that similarity is measured by the weighted Euclidean distance. Since the sum of the color bins is constant (it equals the number of pixels in the image) and one of the weights of the distance function can be set to a constant value, say 1, without altering at all the retrieval process, it turns out that M_{opt} is a function from $\Re^{31}$ to $\Re^{31+31}$. □

Figure 4 shows the basic architecture of a generic interactive retrieval system enriched with FeedbackBypass, with the flow of interactions being summarized in Figure 5 using a C++ like notation. Upon receiving the initial user query $\mathbf{q}$, the system forwards $\mathbf{q}$ to FeedbackBypass by invoking its `Mopt` method, which returns the predicted OQPs $(\mathbf{\Delta_{opt}}, \mathbf{W_{opt}})$ for $\mathbf{q}$. Then, the usual query processing-user evaluation-feedback computation loop can take place. When the loop ends, the new OQPs are passed to FeedbackBypass by invoking its `Insert` method, to be stored as new optimal parameters for $\mathbf{q}$. Clearly, this insertion step can be skipped at all if no feedback information has been provided by the user.

```
 //data structure for optimal query parameters (OQPs)
class Oqp {
 Vector Delta(D);
 Vector W(P);
}
 // get the user query
Vector &q = getUserQuery();
 // obtain OQPs from FeedbackBypass
Oqp &v = FeedbackBypass::Mopt(q);
Oqp &vPred = v.copy();
 // main feedback loop
while(feedbackLoop) {
  // compute results for q using OQPs
 Vector results[] = queryEvaluate(q, v);
  // get relevance scores for results
 Score scores[] = getUserFeedback(results)
  // compute new OQPs given the scores
 newValues(q, v, scores);
}
 // in case feedback information has been provided
if(vPred != v)
  // insert new OQPs for query q
 FeedbackBypass::Insert(q, v);
```

Figure 5: Basic interactions between an interactive retrieval system and FeedbackBypass

3.1 Requirements

The method we seek for learning M_{opt} from sample queries has to satisfy a set of somewhat contrasting requirements, which are summarized as follows:

Limited Storage Overhead. Since the number of possible queries to be posed to the system is huge and will grow over time, it is not conceivable to just do some "query book-keeping", i.e. storing the values of M_{opt} for all already-seen queries. The method we seek should have a complexity *independent* of the number of queries and only a low (e.g. linear) complexity in the dimensionalities of the feature and the output spaces.

Prediction. The method should also be able to provide reasonable "guesses" for new queries. It is also requested that the quality of this approximation has to increase over time, as more and more queries and user-feedback information are processed.

Dynamicity. Since we consider an interactive retrieval scenario, it is absolutely necessary that the method is able to efficiently handle updates, i.e. incorporate additional data without rebuilding the approximation of M_{opt} from scratch.

We have been able to achieve a satisfactory trade-off, thus meeting all above requirements, by implementing FeedbackBypass using a wavelet-based data structure, which we call the *Simplex Tree*.

4 The Simplex Tree

The Simplex Tree forms the core of our approach. It organizes the query domain $\mathcal{Q}$ as a set of non-overlapping multi-dimensional intervals on which the approximation for M_{opt} can be defined.

Recall that we want to approximate the optimal query mapping $M_{opt} : \mathcal{Q} \to \Re^D \times \mathcal{W}$, where $\mathcal{Q} \subseteq \Re^D$ and $\Re^D \times \mathcal{W} \subseteq \Re^N$, with $N = D + P$ (see Problem 1), given a small but evolving sample of data points, namely queries for which feedback data is available.

Of the various techniques that mathematical approximation theory provides, we have chosen wavelets to approximate the query mapping. Unlike other transforms, such as the Fourier transform, wavelets model a target function as composition of functions with a limited support. Therefore, modifying the wavelet at a later point in time entails only local recomputations but no re-organization of the representation as a whole.[1] In the following, we make use of this locality and develop the approximation of the optimal query mapping step by step by local recomputation around newly added feedback points. We will use the well-known Haar-Wavelet in the following.

[1]For a comprehensive overview of wavelets and multi-resolution analysis in particular, see e.g. [Kai94, Swe96]

In order to define wavelets in $\mathcal{Q} \subseteq \Re^D$ we first need to organize this high-dimensional vector space as a collection of intervals $\mathcal{S} = \{S_h\}$ such that their union covers the whole query domain, that is, $\mathcal{Q} \subseteq \bigcup_h S_h$. The delimiters of the intervals managed by the Simplex Tree are taken from the sets of points for which user feedback has been provided. Let us denote with $\mathbf{s}$ one of such delimiters, i.e. a query point *stored* in the Simplex Tree. For each $\mathbf{s}$ we maintain in the Simplex Tree also its N-dimensional vector of OQPs, $M_{opt}(\mathbf{s})$. Given $\mathcal{S}$ and a new query point $\mathbf{q}$, the wavelet-based prediction of the OQPs for $\mathbf{q}$ is then obtained, as explained in more detail below, from the OQPs of the stored points that delimit the (unique) interval that contains $\mathbf{q}$.

4.1 Multi-dimensional Triangulation

Given an initial set of query points for which feedback data is available, we define suitable intervals on which we can base our wavelet by *triangulating* the set. In general, a triangulation is a decomposition into simplices, i.e. intervals spanned by $D + 1$ points—that is, triangles in $\Re^2$, tetrahedrons in $\Re^3$, and so forth. Triangulations are one of the fundamental problems in computational geometry and very efficient techniques to find "good" triangulations are known for low dimensional spaces [Meh84, PS85]. Computing triangulations like the Delaunay triangulation, which minimizes the lengths of edges of the simplices, is computational expensive and too time consuming for dimensions higher than 10.

Instead, to keep the computational effort low, we use an *incremental* triangulation technique as we go forward and *split* for every new point its enclosing simplex. More formally, let $S = \{\mathbf{s}_1, \ldots, \mathbf{s}_{D+1}\}$ be the set of points spanning the simplex that encloses the new to-be-stored query point $\mathbf{q}$. Then,

$$S_h = \{\mathbf{s}_j | j \neq h\} \cup \{\mathbf{q}\}, \qquad 1 \leq h \leq D + 1$$

is a decomposition of S into $D + 1$ simplices.[2] Figure 6 shows examples for splits in two and three dimensions, respectively.

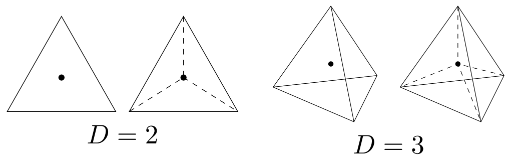


Figure 6: Splitting of 2- and 3-dimensional simplices

Note that splitting a simplex can be done in $O(1)$ time for a fixed dimension, and that the the number of simplices scales linearly with the number of stored query points. Obviously, we can only split a simplex if the new point is inside the simplex itself. To this end we need to define an

[2]For simplicity, we use the same notation to denote both a simplex, i.e. an interval of $\Re^D$, and the set of its $D + 1$ vertices.

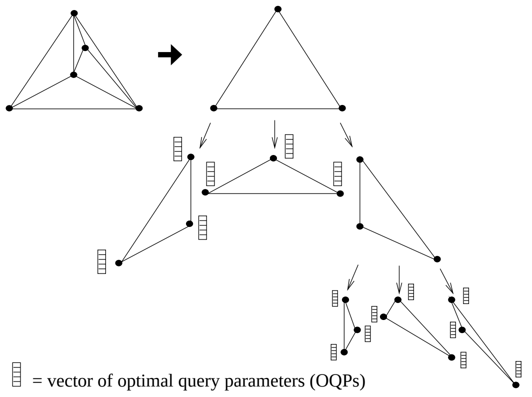


Figure 7: The structure of the Simplex Tree ($D = 2$)

initial simplex, denoted S_0, such that $\mathcal{Q} \subseteq S_0$, i.e. S_0 covers the entire query domain.

The specific details on how S_0 can be defined depend on the data set at hand. For instance, if $\mathcal{Q} = [0,1]^D$, setting $S_0 = \{(0,0,\ldots,0),(D,0,\ldots,0),\ldots,(0,0,\ldots,D)\}$ guarantees that $\mathcal{Q} \subseteq S_0$, as it can be easily verified. On the other hand, when the data set consist of normalized histograms (i.e. the sum over the bins equals 1), by dropping the value of one bin (e.g. the last one) leads to a query domain $\mathcal{Q}$ which already is a simplex, namely $S_0 = \{(0,0,\ldots,0),(1,0,\ldots,0),\ldots,(0,0,\ldots,1)\}$.

4.2 The Data Structure

The Simplex Tree is an index structure that can be characterized as follows:

- each node is a simplex S defined by $D+1$ points;
- every inner node S has pointers to $D+1$ children S_h, which partition S and are pairwise disjoint, i.e. $S = \bigcup_h S_h$ and $S_{h_1} \cap S_{h_2} = \emptyset \ \ \forall h_1, h_2$;
- every leaf node stores for each of its $D+1$ points $\mathbf{s}_j$ the corresponding OQPs, $M_{opt}(\mathbf{s}_j)$;
- S_0, the root, covers the entire query domain $\mathcal{Q}$.

Figure 7 shows the Simplex Tree corresponding to a 2-dimensional triangulation.

The operations necessary to maintain the index are sketched in Figure 8. Below, the individual parts are discussed in more detail.

Lookups. Given a new query point, we need to determine in which simplex the new query point is contained. Starting with the root node we traverse the tree descending at each inner node into the child node which contains the new point.[3]

We do not re-organize the tree in case it gets unbalanced due to the distribution of the data. Hence, the depth of the tree is $O(n)$ in the worst case, n being the number of stored query points, and $O(\log_{D+1} n)$ in the best case. We will assess the average behaviour experimentally in Section 5.

[3]Due to lack of space, we omit the discussion of special cases where the query point is not properly contained in one of the child simplices but it is an element of the delimiting hyperplanes of several simplices.

```
 // initially called with the root simplex
Simplex &SimplexTree::Lookup(Simplex &S, Vector &q) {
   // when in leaf node, we know we found it
  if (S.IsLeaf()) return S;
   // otherwise check each child
  for (int h = 0; h < D + 1; h++)
    if (S.child[h]->Contains(q))
       // descend into h-th child
      return Lookup(S.Child[h],q);
}
Oqp &SimplexTree::Predict(Point &q) {
   // get the enclosing simplex
  Simplex &S = Lookup(q);
   // interpolate in point q using the points of S
  return Wavelet::Interpolate(S,q);
}
void SimplexTree::Insert(Point &q, Oqp &v) {
   // get enclosing simplex
  Simplex &S = Lookup(q);
   // get predicted values in this point
  Oqp &vPredict = Predict(q);
   // if predicted and actual OQPs differ
   // by more than 'epsilon' insert the point
  if (v.Difference(vPredict) > epsilon)
    for (int h = 0; h < D + 1; h++){
       // get the h-th corner of the simplex
      Vector &pCorner = GetCorner(h);
       // create a simplex using the points of this
       // simplex but exclude pCorner and add q instead
      Simplex &SSon = S.CreateSimplex(pCorner, q);
       // add the new simplex as child
      S.AddChild(SSon);}
}
```

Figure 8: Basic functionality of the Simplex Tree

Interpolation. To interpolate off the Simplex Tree, i.e. define a wavelet representation of the mapping, first observe that for each point $\mathbf{s}$ in the Simplex Tree the value of $M_{opt}(\mathbf{s}) = (m_1(\mathbf{s}), m_2(\mathbf{s}), \ldots, m_N(\mathbf{s}))$ is stored. Thus, given a query point $\mathbf{q}$ for which an approximation of $M_{opt}(\mathbf{q})$ is sought, we can solve the problems of approximating each of the N $m_i(\mathbf{q})$ values independently of each other.

Using an unbalanced Haar-Wavelet for approximating $v_i = m_i(\mathbf{q})$ means to perform a linear interpolation in $\mathbf{q}$ given the values $v_i{}^{\mathbf{s}_j} = m_i(\mathbf{s}_j)$ of the $D+1$ points defining the simplex $S = \{\mathbf{s}_1, \ldots, \mathbf{s}_{D+1}\}$ enclosing $\mathbf{q}$. Since S is a D-dimensional linear subspace, solving

$$\begin{vmatrix} q_1 - s_{1,1} & \cdots & q_D - s_{1,D} & \widehat{v_i} - v_i{}^{\mathbf{s}_1} \\ s_{2,1} - s_{1,1} & \cdots & s_{2,D} - s_{1,D} & v_i{}^{\mathbf{s}_2} - v_i{}^{\mathbf{s}_1} \\ \cdots & & \cdots & \cdots \\ s_{D+1,1} - s_{1,1} & \cdots & s_{D+1,D} - s_{1,D} & v_i{}^{\mathbf{s}_{D+1}} - v_i{}^{\mathbf{s}_1} \end{vmatrix} = 0$$

for $\widehat{v_i}$ yields the desired approximation of $v_i = m_i(\mathbf{q})$. Note that, for a given data set, the complexity of interpolation is $O(1)$, since neither D nor P change.

Inserts. As opposed to typical spatial index structures the Simplex Tree is not an index whose aim is to store points to be searched. Instead, it stores points to organize the feature space into simplices. As a consequence, not every point needs to be inserted, since it is sufficient to insert only those points that can improve the quality of the approximation in a *significant* way. These are the points for which

$$\max_i |m_i(\mathbf{q}) - \widehat{v}_i| > \epsilon$$

holds, for a given threshold ϵ. In other words, if all the predictions $\widehat{v}_i$'s are already almost equal to the corresponding $m_i(\mathbf{q})$'s there is no need to store $\mathbf{q}$ in the Simplex Tree. The particular choice of the threshold ϵ determines the quality of the approximation: for low thresholds the approximation is more accurate whereas high thresholds cause more slack. More important, however, is the character of the optimal query mapping. If M_{opt} is composed of low frequencies, only very few query points are stored, whereas for a query mapping composed of high frequencies, more query points are needed to reach approximations of suitable quality. As a limit case, when the OQPs always coincide with the default ones, no point at all is inserted in the Simplex Tree. Consequently, the resource requirements of the Simplex Tree *do not* depend on the number of queries for which feedback is provided but on the intrinsic complexity of the optimal query mapping and on the insert threshold.

5 Experimental Evaluation

We have implemented FeedbackBypass in C++ under Linux, and tested its performance in order to answer the following basic questions:

- Which are the actual prediction capabilities of FeedbackBypass? How much feedback information does FeedbackBypass need to perform reasonably well? How long does it take to learn the optimal query mapping?
- How much do the predictions of FeedbackBypass depend on the specific data set? Alternatively, is FeedbackBypass robust to changes in the type of queries to be learned?
- How much do we gain, in terms of efficiency, by "skipping", or shortening, the feedback loop?

For evaluation purposes we used the IMSI data set consisting of about 10,000 color images.[4] Each image is already annotated with a *category* (such as "birds", "monuments", etc.). From each image, represented in the HSV color space, we extracted a 32-bins color histogram, by dividing the hue channel H into 8 ranges and the saturation channel S into 4 ranges.[5] To compare histograms we use the class of weighted Euclidean distances, with the (unweighted) Euclidean distance being the default function. We implemented both query point movement and re-weighting feedback strategies, as described in Section 2, which means that M_{opt} is a function from $\Re^{31}$ to $\Re^{62}$ (see also Example 1).

The setup for the experiments was as follows. From the whole data set we selected 2,491 images belonging to 7 categories: Bird (318 images), Fish (129), Mammal (834), Blossom (189), TreeLeaf (575), Bridge (148), and Monument (298). This subset of images was then used to randomly sample queries, whereas images in other classes were just used to add further noise to the retrieval process. For each query image, any image in the same category was considered a "good" match whereas all other images were considered "bad" matches, *regardless of their color similarity*. This leads to hard conceptual queries, which however well represent what users might want to ask to an image retrieval system. Since within each category images largely differ as to color content, any query based on a color distance cannot be expected to find more than a fraction of relevant images to be close in color space. For instance, all the 4 images shown in Figure 9 belong to the "Fish" category: only the 2nd image ("shark") has a dominant blue color, whereas others have strong components of yellow, gray, and orange, respectively. A similar evaluation procedure was also adopted in [RH00].

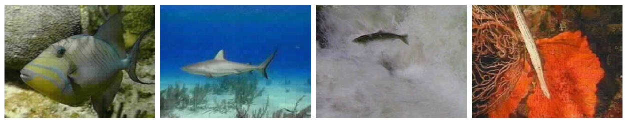


Figure 9: Sample images from the "Fish" category

To measure the effectiveness of FeedbackBypass we consider classical *precision* and *recall* metrics [Sal88], averaged over the set of processed queries. For a given number k of retrieved objects, precision (Pr) is the number of retrieved relevant objects over k, and recall (Re) is the number of retrieved relevant objects over the total number of relevant objects (in our case, the number of images in the category of the query).

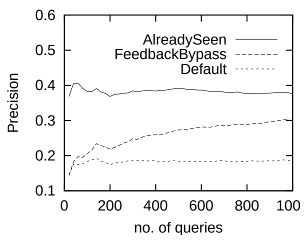
In our experiments we used a typical value of $k = 50$, and in any case k never exceeded 80. This is because we consider that a real user will hardly provide feedback information for larger result sets. As a consequence, since the number of retrieved good matches is limited above by k (and in practice stays well below the k limit), the use of distance functions more complex than weighted Euclidean, such as Mahalanobis, was not considered. Indeed, as observed in [RH00], improvement due to feedback information is possible only when the number of good matches is not much less than the number of parameters of the distance function to be learned, which is 31 in our case but would be $31 \times 32/2 = 496$ for the Mahalanobis distance.

The results we show refer to three different scenarios:

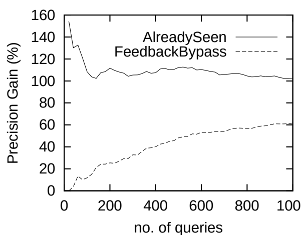
- Default: this is the strategy currently used by *all* interactive retrieval systems, which starts the search by using the user query point and the default distance function (i.e. the Euclidean one in our case);
- FeedbackBypass, for which precision and recall *always* refer to "new" (i.e. never seen before) queries for which the optimal query point and the optimal distance function, as predicted by the FeedbackBypass module, are used in place of the user query and the

[4]IMSI MasterPhotos 50,000: `http://www.imsisoft.com`.

[5]See also `http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html`.



(a)



(b)

Figure 10: Precision results: (a) absolute values; (b) gains with respect to the DEFAULT strategy

default Euclidean distance;

- AlreadySeen: this is mainly used for reference purpose, and corresponds to the case where the FeedbackBypass module delivers predictions for already seen queries, for which the predicted parameters indeed coincide with the optimal ones. It can be argued that the more the results from FeedbackBypass and AlreadySeen are similar, the more FeedbackBypass is approaching the intrinsic limit established by the use of a given class of distance functions and of specific relevance feedback strategies.

For each query, after measuring precision and recall for the first round of k results, we automatically run (using the category information associated to each image) the feedback loop until it converges to a stable situation, i.e. when no changes are observed anymore in the result list. The corresponding query parameters are then sent to FeedbackBypass for insertion.

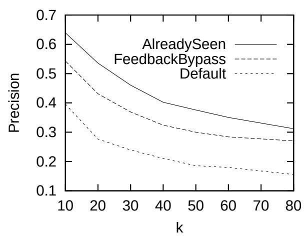
5.1 Speed of Learning

Figures 10 (a) shows average precision as a function of the number of processed queries. For this figure the number of retrieved objects was set to $k = 50$. It is evident that the performance of FeedbackBypass monotonically increases with the number of queries, and that the difference between FeedbackBypass and the Default strategy is already significant after the first few hundred queries. This is also emphasized in Figure 10 (b), where we show values of the *precision gain*, PrGain, defined as:

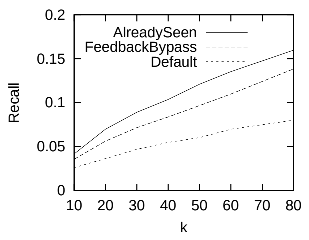
$$\text{PrGain}(\text{FeedbackBypass}) = \left(\frac{\Pr(\text{FeedbackBypass})}{\Pr(\text{Default})} - 1 \right) \times 100$$

and similarly for the AlreadySeen case. The number of good matches doubles for already seen queries, and increase by 60% for queries never seen before.

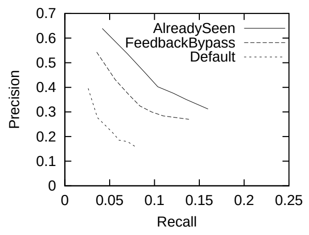
Figures 11 (a), (b), and (c) show, respectively, the values of average precision, recall, and precision vs. recall after 1000 queries, when k varies between 10 and 80. The graphs confirm that our method is able to provide accurate predictions even when the number of retrieved objects per query, k, is low. This can also be appreciated in Figures 12 (a) and (b), where precision and recall curves for $k = 20, 50$, and 80 are plotted versus the number of queries.



(a)

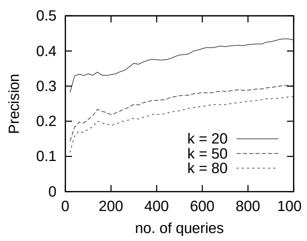


(b)

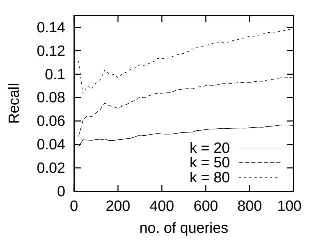


(c)

Figure 11: Precision (a), recall (b), and precision vs. recall curves (c) after 1000 queries



(a)



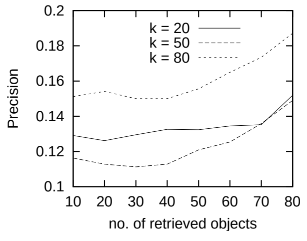
(b)

Figure 12: Precision (a) and recall (b) of FeedbackBypass for different values of k

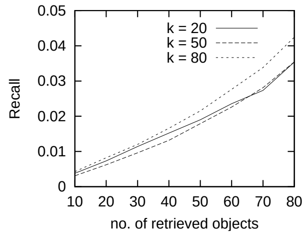
In the previous experiments we have considered a same value of k both to train the system and to evaluate it. However, it is also important to understand if training FeedbackBypass with larger values of k can be better than training FeedbackBypass with less information. Clearly, precision results shown in Figure 12 (a) are of little use to this purpose, since they are obtained with a different number of retrieved objects for each curve. Thus, we have compared several versions of FeedbackBypass each trained with a specific k value, when they are used to answer queries requesting the same number of objects from each version. The basic conclusion that can be drawn from the results shown in Figure 13 is that using larger k values is worthwhile, even if less objects are retrieved. This is particularly evident for the $k = 80$ curve, while less for the case $k = 50$.

5.2 Robustness

We now turn to consider how much the performance of FeedbackBypass depends on the specific queries for which predictions are required. For this experiment we



(a)



(b)

Figure 13: Precision (a) and recall (b) of FeedbackBypass for several values of k as a function of the number of retrieved objects

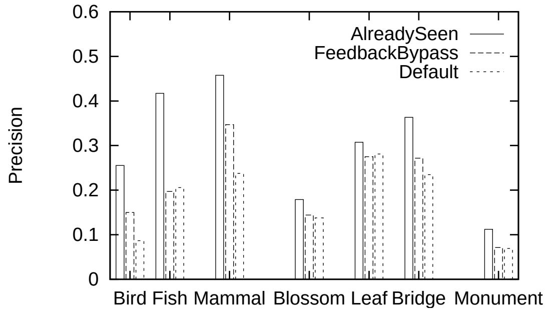
separately measured precision for the 7 query categories. Looking at precision results (see Figure 14 (a)) it can be observed that FeedbackBypass is able to provide useful predictions in all cases where there is a significant difference between the Default and the AlreadySeen cases. Indeed, such a difference is a clear indication that feedback information actually leads to improve the results. This is particularly evident for the largest query category (“Mammal”). On the other hand, when feedback only slightly improves the quality of the results (see the “TreeLeaf” category, denoted simply as “Leaf” in the figure), predictions for new queries do not provide benefits, as it could have been expected. This general behavior is only violated for the “Fish” category, where it seems that no improvement can be obtained from FeedbackBypass on new queries, even if performance of AlreadySeen is particularly good. However, since “Fish” is the smallest category (129 images), it can be argued that the number of sampled queries is still not enough to well approximate the optimal query mapping for that category. Similar results are observed in Figure 14 (b) for the recall metric.

5.3 Efficiency

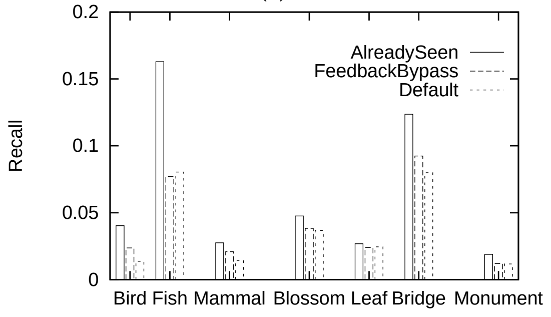
An important aspect that we analyze here is how much we can gain by using FeedbackBypass in terms of *efficiency*. Clearly, the overall performance of an interactive retrieval system will also depend on the specific access methods that are used to retrieve the stored objects, as well as by the indexed features. In order to provide unbiased results, we consider the following performance metrics:

- The average number of feedback iterations that FeedbackBypass saves with respect to the Default strategy, in order to obtain the same level of precision. Thus, for each query we start the feedback loop either from default or from predicted query parameters, and measure how many iterations are needed before no further improvements are possible. This “Saved-Cycles” measure tells us how many query requests to the underlying system we save, on the average, for each user query.

- The average number of objects that we *do not* have to retrieve for achieving the same level of precision than Default. Note that this “Saved-Objects” metric is simply computed as: Saved-Objects $=$ Saved-Cycles $\times\, k$



(a)

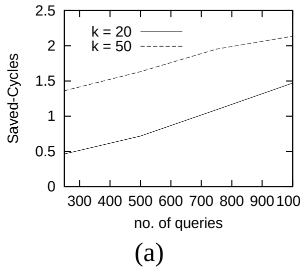


(b)

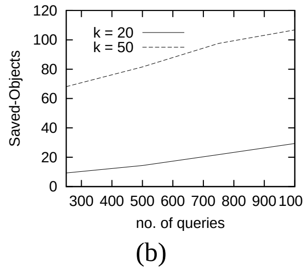
Figure 14: Precision (a) and recall (b) for the 7 query categories

Figure 15 presents results for $k = 20$ and $k = 50$. In both cases it can be seen that the savings improve over time, and that after 1000 queries they amount to about 2 cycles for $k = 50$, which translates in a net reduction of 100 objects retrieved from the underlying system.

Finally, in the last experiment we assess the Simplex Tree as such. Figure 16 shows the average number of simplices traversed to reach a leaf node, together with the depth of the tree, i.e. the maximum number of simplices that could be traversed. Both are logarithmically increasing, however, the average number of traversed simplices is significantly lower than the depth of the Simplex Tree, which leads to fast predictions of the optimal query parameters and underlines the efficiency of FeedbackBypass.



(a)



(b)

Figure 15: Average number of feedback cycles (a) and retrieved objects (b) saved by FeedbackBypass

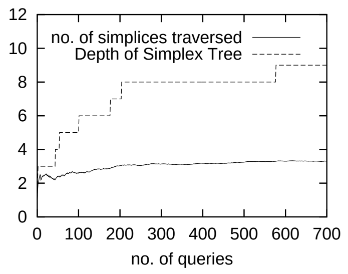


Figure 16: Average number of simplices traversed per query and depth of Simplex Tree

6 Conclusions

In this paper we have presented FeedbackBypass, a new method to speed-up the process of interactively searching for relevant information in multimedia databases. The key idea of FeedbackBypass is to organize the information gathered from user interaction as a multi-dimensional wavelet stored into the so-called Simplex Tree. Approximations obtained from this wavelet can be used to either "bypass" the feedback loop completely for already-seen queries, or to "predict" near-optimal parameters for new queries. We detailed the operations on the Simplex Tree, including inserts, lookups, and interpolation.

Our experiments show that FeedbackBypass works well on real high-dimensional data, and that its predictions consistently outperform basic retrieval strategies which start with default query parameters. We have also quantified the savings FeedbackBypass provides in terms of number of queries and of retrieved objects.

A key feature of FeedbackBypass is its orthogonality to existing feedback models, i.e. FeedbackBypass can be easily incorporated into current retrieval systems regardless of the particular mathematical model underlying the feedback loop. FeedbackBypass is distinguished by its low resource requirements which grow polynomially with the dimensionality of the data set, thus making it applicable to high-dimensional feature spaces.

Our future research is geared toward the application of FeedbackBypass to other types of multimedia data and a thorough investigation of the relationship between the resource requirements and the accuracy of the delivered predictions.

References

[BKK96] S. Berchtold, D.A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 28–39, Mumbai (Bombay), India, September 1996.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 426–435, Athens, Greece, August 1997.

[Fal96] C. Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[ISF98] Y. Ishikawa, R. Subramanya, and C. Faloutsos. MindReader: Querying Databases Through Multiple Examples. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 218–227, New York, NY, USA, August 1998.

[Kai94] G. Kaiser. *A Friendly Guide to Wavelets*. Birkhäuser, Boston, Basel, Berlin, 1994.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms Vol. 3: Muti-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, New York, etc., 1984.

[ORC$^+$97] M. Ortega, Y. Rui, K. Chakrabarti, S. Mehrotra, and T. Huang. Supporting Similarity Queries in MARS. In *Proc. of the Int.'l Conference on Multimedia*, pages 403–413, Seattle, WA, USA, November 1997.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, Berlin, New York, etc., 1985.

[RH00] Y. Rui and T. S. Huang. Optimizing Learning in Image Retrieval. In *Proc. of IEEE Int'l. Conf. on Computer Vision and Pattern Recognition*, Hilton Head, SC, USA, June 2000.

[RHOM98] Y. Rui, T. S. Huang, M. Ortega, and S. Mehrotra. Relevance Feedback: A Power Tool for Interactive Content-Based Image Retrieval. *IEEE Trans. on Circuits and Systems for Video Technology*, 8(5):644–655, September 1998.

[Sal88] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, 1988.

[SK97] T. Seidl and H.-P. Kriegel. Efficient User-Adaptable Similarity Search in Large Multimedia Databases. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 506–515, Athens, Greece, August 1997.

[Swe96] W. Sweldens. The Lifting Scheme: A Custom-design Construction of Biorthogonal wavelets. *Appl. Comput. Harmon. Anal.*, 3(2):186–200, 1996.