Online Shape Learning using Binary Search Trees

Nikolaos Tsapanos, Anastasios Tefas and Ioannis Pitas

Dept. of Informatics, Aristotle University of Thessaloniki, Box 451
Thessaloniki,GR-54124

Abstract

In this paper we propose an online shape learning algorithm based on the self-balancing binary search tree data structure for the storage and retrieval of shape templates. This structure can also be used for classification purposes. We introduce a similarity measure with which we can make decisions on how to traverse the tree and even backtrack through the search path to find more possible matches. Then we describe every basic operation a binary search tree can perform adapted to such a tree of shapes. Note that as a property of binary search trees, all operations can be performed in $O(\log n)$ time and are very efficient. Finally, we present experimental data evaluating the performance of the proposed algorithm and demonstrating the suitability of this data structure for the purpose it was designed to serve.

Key words: Incremental Learning Techniques, Online Pattern Recognition, Binary Search Trees
PACS:

1 Introduction

One of the goals of pattern recognition is to indentify similarities between input data and training data. This task can be accomplished by a learner $l(\mathcal{P})$, where $\mathcal{P}$ is the set of the learner's parameters (neuron weights, covariance matrices etc). Given a training set $\mathcal{D} = \{d_1, \ldots, d_n\}$ and an initialization of the parameters $\mathcal{P}^{(0)}$, an offline training process can be used to find the parameters of the learner for which it "learns" $\mathcal{D}$. This process is typically the optimization of a function measuring the performance of the learner on the training data.

In the online case, it is possible that none of the elements of the training set $\mathcal{D}$ is known beforehand. An online training method should, therefore, be able to alter the parameters of an online learner $l(\mathcal{P}^{(i)})$ that "knows" i elements so that it "learns" a new element d_{i+1} of the expanding training set without "forgetting" the previous elements. The parameters of the resulting online learner after it "learns"

d_{i+1} should also be determined as a function of the learners current parameters and d_{i+1}, without directly involving all or most of the training data already available, otherwise it wouldn't be different from an offline learner. It is also possible that an online learner will be required to "forget" an element d_j it has already "learned", while retaining the ability to perform well on the rest of the training data. An online training method that alters the parameters of a learner in order to exlude the element d_j should also be able to do so as a function of the learners current parameters and d_j.

When designing an online learner two of the most important matters to address are speed and scalability. Speed refers to the time in which the learner classifies, learns and forgets data. The scalability of the learner refers to its ability to maintain an acceptable performance in both speed and classification as the size of the training dataset grows and shrinks over time after multiple insertions and deletions. Self-balancing binary search trees are well known data structures for quick search, insertion and deletion of data [8]. They are therefore a very nice base for an online learner.

In this paper, we propose a way to adapt this kind of data structure to a binary search tree for shapes, or a *shape tree* as it will be referred to from now on. For our purposes, we view a shape as a set of points with 2-dimensional integer coordinates. We do so because this is the simplest way to represent a shape, though there are several other, more complicated options [3]. By designing such a shape tree, we can insert and search for shapes in logarithmic worst case time. Furthermore, for each doubling of the data, a binary search tree needs only one more level to store the additional data. This data structure can then be applied to the task of object recognition.

Object recognition by shape matching traditionally involves comparing an input shape with various template shapes and reporting the template with the highest similarity (usually a Hausdorff based metric [5],[6]) to the input shape. The final decision depends on the maximum reported similarity. As the template database becomes larger, however, exhaustive search becomes impractical and the need of a better way to organize the database aiming to optimize the cost of the search operations arises. This can be accomplished by inserting the shape templates into a shape tree. Then we can use the proposed shape tree to quickly search for the template with the highest similarity to the input shape without having to go through the entire template dataset.

The application of binary search trees for shape storage and retrieval, however, is not straight forward. The main issue we have to address is that shape similarity has much less discriminatory power than number ordering (for example, shape dissimilarity is not transitive and the triangular inequality doesn't hold [4]). This means that in order for a node to make a decision on which child to direct a search that node must have a more complicated classifier and undergo training for that classi-

fier. This training process must also not involve the number of shapes already stored inside the tree.

Our work is most closely related to [1] and [2], where a tree similar to B-trees is proposed. This tree is constructed bottom-up, based on grouping similar shapes together in size-limited groups, then selecting a representative of the group and repeating, until the entire tree is formed. While searching, the traversal of more than one children of any node is permitted. However, this structure does not provide any theoretical worst case performance guarantee and no way to add further shapes without having to reconstruct the tree.

We propose the shape tree as a learner $l(\mathcal{P})$ where the parameters are the contents of the tree's nodes that will be described later on. The novelties of the proposed data structure consist of a novel variation of Hausdorff Distance (which is used as a shape similarity measure), a novel weak classifier that each tree node uses to make decisions and the organization of said weak classifiers into a binary search tree for shapes.

This paper is organized as follows: section 2 details the basics of the shape tree, section 3 describes all the online tree operations (search, insertion, deletion, rotations) to search and incrementally or decrementally manage the knowledge of a shape tree, section 4 presents experimental data and section 5 concludes the paper.

2 Shape Tree Basics

In this section we go over the basic design of the shape tree's nodes and the method through which decisions are made. We first introduce a variant of the Hausdorff distance that will serve as our similarity measure. We then detail the two types of nodes and their contents. We finally describe how we can use the nodes' contents and our similarity measure to perform the search operation in a shape tree.

2.1 Similarity Measure

The Hausdorff Distance, while originally introduced to measure the difference between point sets, has been widely used to measure shape similarity as well (since a shape can be considered as a set of points). It contains a sub measure called the Directed Hausdorff Distance. In order to measure the directed distance from a point set $\mathcal{X}$ to a point set $\mathcal{Y}$, the Directed Hausdorff Distance is defined as

$$D_{DHD}(\mathcal{X},\mathcal{Y}) = \max_{\mathbf{x}\in\mathcal{X}}(\min_{\mathbf{y}\in\mathcal{Y}}(d(\mathbf{x},\mathbf{y}))) \tag{1}$$

where $d(\mathbf{x},\mathbf{y})$) is the Euclidean distance between point $\mathbf{x}$ and point $\mathbf{y}$. As the Directed Hausdorff Distance is not reflective (typically $D_{DHD}(\mathcal{X},\mathcal{Y}) \neq D_{DHD}(\mathcal{Y},\mathcal{X})$), the overall Hausdorff Distance is defined as the maximum of the directed distances from each point set to the other

$$D_{HD} = \max(D_{DHD}(\mathcal{X},\mathcal{Y}), D_{DHD}(\mathcal{Y},\mathcal{X})) \tag{2}$$

In practice, however, the Hausdorff Distance proves to be too sensitive to outliers. This has lead to the proposal of several variants, among which is the Modified Hausdorff Distance [7]:

$$D_{MHD}(\mathcal{X},\mathcal{Y}) = \frac{1}{|\mathcal{X}|}\sum_{\mathbf{x}\in\mathcal{X}} d(\mathbf{x},\mathcal{Y}) \tag{3}$$

where $|\mathcal{X}|$ is the cardinality of the point set $\mathcal{X}$ and $d(\mathbf{x},\mathcal{Y}) = \min_{\mathbf{y}\in\mathcal{Y}} ||\mathbf{x}-\mathbf{y}||_2$. This variant of the directed Hausdorff Distance has been found to yield better results than other alternatives proposed [7]. Since pixel coordinates are integers, an efficient way to calculate this measure is using the distance transform matrix $\mathbf{Y}$ of the point set $\mathcal{Y}$ to quickly look $d(\mathbf{x},\mathcal{Y})$ up [9]. If $\mathbf{x} = [x_0, x_1]^T$ and $\mathbf{Y}$ is the distance transform matrix of $\mathcal{Y}$ whose elements are $y(i,j)$, we can rewrite (3) as

$$D_{MHD}(\mathcal{X},\mathcal{Y}) = \frac{1}{|\mathcal{X}|}\sum_{\mathbf{x}\in\mathcal{X}} y(x_0,x_1) \tag{4}$$

We will use this similarity measure as our base. In order to normalize the measure in $(0,1]$, we use an activation function on the elements $y(i,j)$ of the distance transform matrix $\mathbf{Y}$ before summing them. We call this measure the Activated Hausdorff Proximity:

$$P_{AHP}(\mathcal{X},\mathcal{Y}) = \frac{1}{|\mathcal{X}|}\sum_{\mathbf{x}\in\mathcal{X}} e^{-\alpha y(x_0,x_1)} \tag{5}$$

This particular activation functions offers several advantages. First of all, it makes the measure infinitely differentiable, which can be useful for future theoretical studies. It also rewards the points of $\mathcal{X}$ being on or very close to a point in $\mathcal{Y}$ by contributing a number close to the unit to the sum. Finally, when considering outliers, the effect that such points have on the measure is almost the same after a certain distance, thus providing robustness to the position of outliers.

Note that if we choose a distance transform function that outputs an integer matrix, we can also use a precomputed array containing the values of $e^{-\alpha k}$ for every integer k that we expect from the distance transform. This way, for every point $\mathbf{x}$ we can look $e^{-\alpha y(x_0,x_1)}$ up with only 3 memory references.

Fig. 1. A binary search tree. Internal nodes are circular while leaf nodes are square. Note that the indices represent node id and not key value.

Fig. 2. A sample leaf node. The actual template is on the left and its distance transform on the right.

2.2 Tree Nodes

There are two types of nodes in our binary search tree: leaf nodes and internal nodes. The actual templates are stored in the leaf nodes, while the internal nodes are used to traverse the tree. Each internal node can have up to two children. Its children can be either leaf nodes or other internal nodes. These two types of nodes are not interchangeable. Templates cannot be stored in a non leaf node and a search path cannot end in a node that does not contain a template. A sample binary search tree can be seen in Figure 1.

2.2.1 Leaf Nodes

A leaf node contains a single template T_i from the training set that has been inserted into the shape tree. The template is stored as a set of 2-dimensional points with integer coordinates. The distance transform of the template is stored here as well to speed up calculations. A leaf node also has a pointer to its parent. The contents of a leaf node are illustrated in Figure 2.

2.2.2 Internal Nodes

The internal nodes are the classifier nodes. They do not contain any real template information and they are used to determine the search path to the leaf nodes where

$\mathbf{S}_L$ $\mathbf{T}_L$ $\mathbf{T}_R$ $\mathbf{S}_R$

(a) The node's parameters.

(b) The templates in the node's left subtree.

(c) The templates in the node's right subtree.

Fig. 3. A sample internal node.

the actual data is stored. Each internal node contains the distance transform $\mathbf{T}_L$ of a "left" template T_L and the distance transform $\mathbf{T}_R$ of a "right" template T_R. These templates are used to determine whether a tree search will continue in the left subtree or in the right subtree.

An internal node also contains a matrix $\mathbf{S}_L$ that contains the sum of every template that is under its left subtree and another matrix $\mathbf{S}_R$ with the sum of every template under its right subtree. These matrices are used to retrain the node during insertions and deletions. Finally, an internal node has a pointer to its left child, a pointer to its right child and the necessary information regarding the balance of the tree (node balance for AVL trees, colour for red/black trees etc). In our experiments, we implemented an AVL tree [10], but any type of tree that achieves balance through tree rotations can also be used.

2.3 Classification

In order to search for a test set of points $\mathcal{X}$ in a shape tree, we must find a path of internal nodes from the root to the leaf node that corresponds to the matching template. Each internal node on that path must decide on which of its subtrees it must direct the search. This can be achieved by using the internal nodes' parameters $\mathbf{T}_L$ and $\mathbf{T}_R$. We measure the similarity of $\mathcal{X}$ with both $\mathbf{T}_L$ and $\mathbf{T}_R$ using (5). We direct the search to the root of the subtree that yielded the highest similarity. If $P_{AHP}(\mathcal{X}, \mathbf{T}_L) > P_{AHP}(\mathcal{X}, \mathbf{T}_R)$, then the search is directed to the left subtree, while it is otherwise directed to the right subtree.

Additionally, we can measure the certainty of a node that it directed the search

to the correct path by calculating the confidence c of this node's decision as $c = |P_{AHP}(\mathcal{X}, \mathbf{T}_L) - P_{AHP}(\mathcal{X}, \mathbf{T}_R)|$. We can later use this confidence to backtrack the search path in the tree.

3 Online Shape Tree Operations

In this section we describe all the online operations that can be performed on a shape tree. These operations are search, insertion, deletion, and balance. We assume that the training set $\mathcal{T} = \{T_1, \ldots, T_{|\mathcal{T}|}\}$ contains all the templates that will be inserted into (and later maybe deleted from) the shape tree. We use n to denote the number of templates that are already stored in the tree, T_i to denote that last template from the training set that was inserted into the tree and T_j to denote a template that has been inserted and has to be deleted from the tree at a given time.

3.1 Search

We will now describe how to find the closest match of a set of points $\mathcal{X}$ in a shape tree. Starting from the root of the tree we follow the path of nodes as dictated by comparing the similarities of $\mathcal{X}$ with each node's $\mathbf{T}_L$ and $\mathbf{T}_R$ until we reach a leaf. Then we report the template of that leaf as a possible result.

Since we cannot give any theoretical guarantees that the first search result is the best one, we need a way to find more possible results. In order to do this, we note the confidence c of each node in the path to the previous result and we backtrack through the path and reverse the decision of the node with the lowest confidence and proceed to search the subtree we skipped in the previous search. Once a node's decision has been reversed, we set its confidence to 1, so that it won't switch again until the search is over. This way, if we allow r tries (note that $r << n$), we come up with r template candidates. Again, we cannot provide theoretical guarantees that these are the r closest matches or that they include the absolute best match. We report the match with the highest similarity after exhaustively testing every one of these candidates.

Regarding the values of nodes' confidence along the path to a leaf, we expect that the confidence will be lower toward the end of the path (because the templates with a low least common ancestor will be similar) and toward the root of the path (because there will be a lot of templates to separate in each subtree). It would be a good idea to replace the confidence by a function of $|P_{AHP}(\mathcal{X}, \mathbf{T}_L) - P_{AHP}(\mathcal{X}, \mathbf{T}_R)|$ and the depth of a node, however we find that it is more practical to artificially restrict switching paths at the higher levels in the first few tries.

Analysis

We assume that the dimensions of $\mathbf{T}_L$ and $\mathbf{T}_R$ are constant. This also means that $\mathcal{X}$ is also bounded by a constant, since there cannot be any more points inside the input shape than the area of the shape window. We replace the constant time operations of a binary search tree (number comparison) with operations that also require constant time with respect to the number of nodes n (similarity calculation). The search operation can therefore be performed in $O(\log n)$ time as per the binary search tree bibliography [8]. If we allow for r tries, then the complexity is $O(r \log n)$ but, as already noted, $r << n$.

3.2 Insertion

Here we describe the process through which a shape tree "learns" a new template T_{i+1}. To insert a new template T_{i+1} into the tree we first create a leaf node p_{i+1} that contains it. Then we search for T_{i+1} in the current tree as previously described. If we come to an internal node with only 1 child during our search, we add the new node as its other child. If the search stops at a leaf node p_k, we replace it with an internal node q and add p_{i+1} and p_k as the new internal node's children. The template T_{i+1} is also added to the proper sum matrix ($\mathbf{S}_L$ or $\mathbf{S}_R$) of every internal node it traverses. This means that the new template will be inserted near similar templates and guarantees that if we search for the template again, the search will find it in the first try (provided no tree rotations have been performed since it's insertion).

After inserting a node, we then follow the reverse path to the root, balancing and retraining every internal node that was affected by the insertion. Balance and node training are described in Sections 3.4 and 3.5, respectively.

Analysis

Again, the changes we propose involve constant time operations (node training is bounded by the dimensions of the node's matrices which we consider constant and indepedent from the number of nodes n), so insertion takes $O(\log n)$ time as a property of binary search trees [8].

3.3 Deletion

Here we describe the process through which a shape tree "forgets" a template T_j. As implied by the definition of this task, only the deletion of leaf nodes is supported. Moreover, the node to be deleted must be known beforehand. Though nothing pre-

vents us from search for in input shape and then deleting the outcome of the search, doing this is inadvisable, as it can lead to problems. For example, trying to delete a template that hasn't been stored will result in the deletion of another template that is similar to the target.

In order to delete a template T_j that is stored in leaf node q_j, we start at that node and travel backwards to the root via the parent node pointers. We subtract the node's template T_j from the proper sum matrix ($\mathbf{S}_L$ or $\mathbf{S}_R$) of each node traversed and rebalance where necessary. Finally, we retrain every affected node. Balance and node training are described in Sections 3.4 and 3.5, respectively.

Deleting nodes and rebalancing can result in an internal node with no children. Since we do not allow that, we check whether an internal node is left childless along the path to the root. We mark that node for deletion and repeat the procedure again.

Analysis

Since the tree search operation and the the tree rebalancing procedures take $O(\log n)$ time, the deletion of a single leaf node also takes $O(\log n)$ time. However, the deletion of a leaf node can also cause a cascade of childless internal node deletions. We use amortized analysis [13] to show that the amortized cost of every deletion is $O(\log n)$.

We assume that there will be a total of n deletions. In order for that to be possible, there also have to be at least n insertions. We credit each inserted node with $3\log n$ units. Each inserted node must pay $\log n$ units for its insertion. It deposits $\log n$ units to itself to pay for its deletion. The node's insertion may have forced the insertion of an additional internal node. The inserted leaf node deposits the last $\log n$ units to the newly inserted internal node. The deletion of this internal node has been "paid for" by the insertion of the leaf node that required the internal node's insertion. Since $3n\log n$ units are enough to pay for n deletions, the amortized cost of the deletion operation is $(3n\log n)/n = O(\log n)$.

3.4 Balance

In this section we describe how the AVL tree rotations can be performed on a shape tree. Some rotations have been slightly modified to fit the shape tree's restrictions. Node balance in an AVL tree is measured as the numerical difference between the height of the node's right subtree and the height of it's left subtree. A node can be perfectly balanced, having an individual balance of 0, or slightly imbalanced with an individual balance of either -1 or 1. An AVL tree is considered to be balanced as long as every node is either perfectly balanced, or slightly imbalanced. If a node's

Fig. 4. The RR case. The rectangles at each side of a node represent it's sum matrices.

balance reaches -2 or 2, then that node needs to be rebalanced.

Rebalancing is accomplished through tree rotations. These rotations affect the current imbalanced node, one of it's children and one of it's grandchildren in the same side. We will outline the case when a node's balance reaches 2 here, as the other case when the balance is -2 is symmetrical. There are two subcases, the RR case and the RL case. Finally, there is an additional special case that can come up when deleting a node.

3.4.1 The RR case

The RR case happens when the current node's right child's balance is non-negative. In this case, the nodes need to be rotated to the left. At the same time, each affected node's sum matrices ($\mathbf{S}_L$ and $\mathbf{S}_R$) must be recalculated. Suppose that q_1 is the imbalanced node, q_2 is q_1's right child and q_3 is q_2's right child. Also suppose that the subtrees excluding q_1,q_2 and q_3 are (from left to right) s_1,s_2,s_3 and s_4. To perform the rotation, q_2 becomes the parent of both q_1 and q_3, while q_1 becomes q_2's left child and takes q_2's left subtree as it's right subtree. Figure 4 shows the nodes' configuration before the rotation, while Figure 6 shows the result of the rotation. The sum matrices are recalculated as shown in these two figures.

3.4.2 The RL case

The RL case happens when the current node's right child's balance is -1. Suppose that q_1 is the imbalanced node, q_2 is q_1's right child and q_3 is q_2's left child. Again suppose that the subtrees excluding q_1,q_2 and q_3 are (from left to right) s_1,s_2,s_3 and s_4. In a normal binary search tree, this would be addressed by making q_3 the parent of q_1 and q_3. However, q_3 can be a leaf node and we do not allow leaf nodes to become internal nodes. Like the previous case, we choose q_2 to be the parent of both q_1 and q_3. The subtrees do not change order and are attached to q_1 and q_3 as their children. Figure 5 shows the nodes' configuration before the rotation,

while Figure 6 shows the result of the rotation. The sum matrices are recalculated as shown in these two figures.

3.4.3 Special case

There is a special case in which node q_1's balance is 2 (the -2 case is symmetrical), but q_1 doesn't have a left child at all. Performing a left rotation would make q_1 a leaf node and this not allowed in a shape tree. To handle this case, we simply replace q_1 with q_1's right child.

3.5 Node Training

After insertion, deletion, or balance takes place, every affected node must be re-trained. Since the shape tree was designed to be able to perform these operations online (or even real time), the training process must be very fast. This prohibits

Fig. 5. The RL case. The rectangles at each side of a node represent it's sum matrices.

Fig. 6. The result of rebalancing rotations. The rectangles at each side of a node represent it's sum matrices. To see which matrices should be added together please refer to the appropriate Figures for each case.

taking into account every template that the node stores during training, as this takes linear time with respect to the number of such templates and the templates can be as many as the entire training set.

The fastest and simplest training method is to simply extract the template $\mathbf{T}_L$ from $\mathbf{S}_L$ and the template $\mathbf{T}_R$ from $\mathbf{S}_R$. $\mathbf{S}_L$ is the sum of every template in the left subtree, so we can scan the matrix to find all the non-zero entries and build the set of points for $\mathbf{T}_L$ from the matrix coordinates of those entries (likewise for $\mathbf{T}_R$).

Since we train a node by only considering it's own parameters, the training can be performed in constant time. Unfortunately, this also means that we cannot guarantee that the resulting node will be able to correctly classify every template under it. The search operation allows for more than one tries for this very reason.

4 Experimental Results

In this section we present the results of several experiments testing the various aspects of the proposed data structure's performance as a tool for classification. We begin by introducing the database that all tests were performed on and describing how the proposed data structure can be used to make a classifier that can learn the data. We then proceed with the testing. We first tested the shape tree's speed by measuring the time required to insert the training data into the tree and the time required to epmty the tree by deleting all its contents. We then tested the learning capabilities of the tree by using the tree to classify its own training set. We also tested the generalization capabilities of the tree by measuring how well it classifies samples that it has trained for but not encountered before. Finally, we tested the robustness of the shape tree by measuring its classification performance while constantly inserting and deleting templates. The tests run on an AMD Athlon X2 6400 processor (each processor clocked at 3.2GHz). Every averaged number is presented as *mean* (*standard deviation*).

4.1 The database

All the experiments were conducted using the MNIST handwritten digits database [11]. This database contains 70046 28×28 images of the numbers 0 through 9 as written by 250 writers. They are separated into a subset of 60027 training images and a subset of 10019 test images. For a more detailed breakdown of the images, please refer to table 1, where each row lists the number of image for each digit in both subsets.

The state of the art approach uses a linear filter for encoding an image into a code

Digit	Training	Test
0	5923	980
1	6745	1135
2	5958	1036
3	6131	1010
4	5845	982
5	5440	900
6	5918	958
7	6266	1031
8	5851	974
9	5950	1013

Table 1
The number of samples from the MNIST database by digit and subset.

and a sparsifying logistic function that the code goes through before being decoded by another linear filter [12]. The learning is achieved by training the encoder to produce codes whose sparse representation accurately reconstruct the original image after decoding. The error rate on the MNIST database reported is 0.39%.

4.2 Classification using shape trees

We begin by labelling every image with the digit it depicts. We then consider every image of the training set as a template T_i that consists of the point set $\mathcal{T}_i$. The templates are inserted into the tree in random order. When an image needs to be tested, we convert it to a point set $\mathcal{X}$ and input this set into the tree. The shape tree then returns a list of r possible candidates $\{T_1, \ldots, T_r\}$, where r is the number of tries we have set. At this point we have several options in selecting a final candidate. There are two figures we can use to form a final criterion, the directed proximity from the input to the template $P_{AHP}(\mathcal{X}, \mathcal{T}_i)$ and the directed proximity from the template to the input $P_{AHP}(\mathcal{T}_i, \mathcal{X})$.

One possibility is to use the first proximity, but this is problematic. For example, the points of the digits 3 and 5 overlap a lot with the points of the digit 8. This can lead to many 3s and 5s being incorrectly classified as 8s. The proximity from a 3 or a 5 to an 8, however is smaller than the proximity from an 8 to a 3 or a 5. Taking this and equation (2) into account, we measure the total proximity from the input $\mathcal{X}$ to the template $\mathcal{T}_i$ as $\min(P_{AHP}(\mathcal{X}, \mathcal{T}_i), P_{AHP}(\mathcal{T}_i, \mathcal{X}))$. We make our final decision for the best match

$$\mathcal{T}_{best} = \arg\max_{\mathcal{T}_i}(\min(P_{AHP}(\mathcal{X}, \mathcal{T}_i), P_{AHP}(\mathcal{T}_i, \mathcal{X})))$$

If the label of $\mathcal{T}_{best}$ matches the label of $\mathcal{X}$ then the classification is considered to be correct.

Fig. 7. The times (in ms) that a node insertion takes vs. number of nodes already in tree.

4.3 Insertion

In this test we started with an empty tree and proceeded to insert all the 60027 templates of the training set into it. We measured the time each insertion took. The graph of these values can be seen in Figure 7. By observing the graph, we can see that it reasonably follows a logarithmic curve, as theoretically expected. The various spikes are attributed to insertions that cause more rebalances than usual. Note that since insertion searches for the target template first, this graph includes searching times. Average insertion time was $0.0026(0.0006)$ ms, while total tree construction time was 154 seconds.

4.4 Deletion

Starting from the tree constructed in the previous test, we deleted every template that the tree contained and measure the time each deletion took. The graph of these values can be seen in Figure 7. It appears that deletion is faster than insertion. This is natural, considering that insertion needs to search for the target template first, while deletion reaches the root through parent pointers. Again, the spikes in the graph are attributed to some deletions causing a greater number of rebalances. Average deletion time was $0.00023(0.00029)$ ms, while the total time it took to empty the tree was 14 seconds.

Fig. 8. The times (in ms) that a node insertion takes vs. number of nodes already deleted from the tree.

4.5 Learning Capabilities

In order to test the shape tree's learning capabilities and determine how the number of tries affects its classification abilities, we constructed a tree by inserting all the 60027 templates of the training set into it. The tree was then tasked with classifying those same templates it was tasked with learning. We used different values for the amount of tries the tree is allowed to use when searching for each template. Search time was also measured.

The results of this test are presented in Table 2. Each row contains the test results for an increasing number of total tries. Classification rates were measured as the number of correct classifications divided by the total number of templates for each digit. Classification rates were rounded to the second decimal digit, unless rounding would result in a value of 1.00, in which case it was left at 0.99. Judging from this test's results, the shape tree is extremely capable of learning its training set as long as it is given enough tries to do so. It should also be noted that the increase in search time is not constantly analogous to the number of tries.

Tries	0	1	2	3	4	5	6	7	8	9	Time in ms
4	0.98	0.98	0.93	0.92	0.90	0.84	0.97	0.93	0.91	0.91	0.000076 (0.00038)
8	0.99	0.99	0.97	0.96	0.96	0.92	0.98	0.97	0.96	0.95	0.00012 (0.00011)
16	0.99	0.99	0.99	0.98	0.98	0.97	0.99	0.99	0.98	0.97	0.00024 (0.00014)
32	0.99	0.99	0.99	0.99	0.99	0.98	0.99	0.99	0.99	0.99	0.00047 (0.00024)
64	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.00089 (0.00024)
128	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.0017 (0.00039)
256	1.00	0.99	1.00	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.0027 (0.0011)

Table 2
Classification rates and average search time for different number of tries.

4.6 Generalization Capabilities

Generalization capabilities were tested by first inserting the training set into the tree then using that tree to classify the test set. The number of tries was set at 64, as this number seemed to yield improved results without slowing down the classification procedure considerably. We didn't use a validation set, because doing so would go against the spirit of online learning.

Firstly, we used the 60027 templates for training and tested the resulting trees in the 10019 digits that were designated as the test set. After that, we reversed the sets and used the 10019 templates to train the trees and the 60027 of the original training set to test this second batch of trees. The templates are inserted into the tree in random order. However, it is obvious that different insertion orders will produce different shape trees. Thus, in order to illustrate the effect of the insertion ordering has on the classification performance, we construct ten different trees that correspond to ten different random orderings of the inserted templates. We present results for each tree and average classification rates for the ten trees.

The classification rates and search times for the first batch of trees are reported in Table 3. Each row of the table lists the classification rates for each of the 10 trees constructed using the 60027 template training set. Total average classification rate was 0.9286 . Results also suggest that shape trees are consistent in their performance, as there are no wild variations in classification rates.

The classification rates and search times for the second batch of trees are reported in Table 4. In this case, each row of the table lists the classification rates for each of the 10 trees constructed using the 10019 templates of the original test set as training set and the original 60027 training samples were used for testing. Total average classification rate was 0.8970 . In comparison with the first batch of trees, the decrease in search times is minimal. Even though the previous training set is 6 times bigger, due to the logarithmic property of binary search trees the speed is not noticeably burdened. The classification rate of this second batch of trees is also 3% lower. This can be expected when considering that the training dataset is smaller.

	0	1	2	3	4	5	6	7	8	9	Time in ms
Tree 1	0.98	0.98	0.94	0.94	0.92	0.87	0.97	0.92	0.88	0.90	0.0015 (0.00037)
Tree 2	0.91	0.99	0.94	0.93	0.90	0.88	0.98	0.92	0.86	0.89	0.0017 (0.00078)
Tree 3	0.99	0.99	0.93	0.93	0.93	0.88	0.97	0.93	0.86	0.89	0.0015 (0.00037)
Tree 4	0.99	0.99	0.94	0.92	0.92	0.88	0.97	0.93	0.87	0.89	0.0015 (0.00035)
Tree 5	0.99	0.99	0.94	0.92	0.93	0.88	0.97	0.92	0.87	0.88	0.0015 (0.00034)
Tree 6	0.99	0.98	0.95	0.93	0.91	0.87	0.96	0.93	0.86	0.86	0.0015 (0.00031)
Tree 7	0.99	0.99	0.94	0.92	0.92	0.89	0.97	0.91	0.88	0.87	0.0014 (0.0003)
Tree 8	0.99	0.99	0.94	0.93	0.92	0.89	0.97	0.92	0.88	0.90	0.0015 (0.00032)
Tree 9	0.99	0.99	0.94	0.92	0.92	0.87	0.96	0.93	0.86	0.88	0.0015 (0.00032)
Tree 10	0.98	0.98	0.94	0.94	0.91	0.88	0.97	0.92	0.86	0.87	0.0016 (0.00048)
Average	0.99	0.99	0.94	0.93	0.92	0.89	0.97	0.92	0.87	0.88	0.0015 (0.0004)

Table 3
Classification rates and average search time for the trees trained with 60027 templates and tested on 10019 digits.

	0	1	2	3	4	5	6	7	8	9	Time in ms
Tree 1	0.98	0.98	0.91	0.89	0.85	0.82	0.94	0.90	0.81	0.87	0.0011 (0.0003)
Tree 2	0.98	0.98	0.91	0.89	0.87	0.83	0.96	0.92	0.82	0.86	0.0012 (0.00027)
Tree 3	0.98	0.98	0.92	0.89	0.84	0.81	0.96	0.90	0.82	0.86	0.0011 (0.00026)
Tree 4	0.98	0.98	0.91	0.87	0.84	0.83	0.94	0.91	0.82	0.86	0.0011 (0.00026)
Tree 5	0.97	0.98	0.91	0.89	0.84	0.82	0.96	0.91	0.81	0.86	0.0011 (0.00026)
Tree 6	0.98	0.98	0.91	0.89	0.84	0.80	0.95	0.92	0.81	0.86	0.0011 (0.00024)
Tree 7	0.97	0.98	0.92	0.86	0.87	0.84	0.95	0.90	0.83	0.86	0.0011 (0.0003)
Tree 8	0.97	0.97	0.92	0.88	0.84	0.83	0.96	0.90	0.81	0.85	0.0010 (0.00027)
Tree 9	0.98	0.98	0.90	0.87	0.85	0.83	0.96	0.90	0.82	0.86	0.0011 (0.00023)
Tree 10	0.97	0.98	0.91	0.87	0.86	0.85	0.94	0.91	0.83	0.84	0.0011 (0.00031)
Average	0.98	0.98	0.91	0.88	0.85	0.83	0.95	0.90	0.82	0.86	0.001 (0.00027)

Table 4
Classification rates and average search time for the trees that were reversely trained with 10019 templates and tested on 60027 digits.

4.7 Robustness

As the final experiment, we tested the proposed data structure's robustness to multiple insertions and deletions. We used the original training set to train the tree and the original test set to measure classification rates. Since the classes that are most likely to be confused with each other are those that correspond to the digits 3, 5 and 8, we begin by inserting these templates into the initial tree. We denote the fact that the tree currently contains these digits as $\{3,5,8\}$. Thus, we consider the worst case scenario where the most difficult classes are always present inside the shape tree. We denote the insertion of additional digits with the symbol $\oplus$ and the deletion of digits with the symbol $\ominus$.

After starting with the initial tree we proceed to insert and delete digits at consec-

Fig. 9. The classification rate graphs for the digits 3 (black), 5 (dark grey) and 8 (light grey).

Digits in tree	0	1	2	3	4	5	6	7	8	9
$\{3,5,8\}$	-	-	-	0.92	-	0.93	-	-	0.96	-
$\{3,5,8\} \oplus \{0,1\}$	0.99	0.99	-	0.91	-	0.92	-	-	0.94	-
$\{0,1,3,5,8\} \oplus \{7,9\}$	0.99	0.99	-	0.90	-	0.91	-	0.95	0.92	0.93
$\{0,1,3,5,7,8,9\} \ominus \{0,9\}$	-	0.99	-	0.91	-	0.93	-	0.98	0.95	-
$\{1,3,5,7,8\} \oplus \{4,6\}$	-	0.99	-	0.90	0.98	0.91	0.98	0.96	0.91	-
$\{1,3,4,5,6,7,8\} \ominus \{1,4,6,7\}$	-	-	-	0.92	-	0.92	-	-	0.96	-
$\{3,5,8\} \oplus \{0,1,2,4,6,7,9\}$	0.98	0.96	0.93	0.87	0.93	0.88	0.97	0.94	0.89	0.89

Table 5
Classification rates at the various stages of the experiment.

utive stages. At each new stage (after the insertions or deletions are finished), we measure the classification rates for all the digits that are into the tree at the current stage. We also monitor the classification rates for the digits 3, 5 and 8 the are present at every stage. The graph that contains the change of classification rates of the three most difficult classes over the stages of this experiment can be seen in Figure 9, while the classification rates of all the digits involved in all the stages are presented in table 5 with a '-' denoting that the tree isn't trained for that specific digit and that digit is therefore not tested.

By observing Figure 9 we can see that the classifier has an easier time classifying the three digits when there are fewer overall templates stored in the tree. The classification rate decreases when more templates are inserted and increases as templates are removed. Note that when the only templates inside the tree are those that correspond to the digits 3, 5 and 8 in stage 6 of the experiment their classification rate is the same as it was in the initial tree that also contained these three digits only. Furthermore, the final classification rates match the ones in Table 3, obtained from the generalization test. This observation highlights the online training capabilities of the proposed shape tree.

We should also note that we haven't used any preproccessing steps in order to improve the performance of the proposed approach, since our objective is to highlight the suitability of the proposed shape learning data structure for the online pattern recognition task.

5 Conclusion

In this paper, a novel online shape learning algorithm along with an appropriate data structure for shape storage has been proposed. It is a variation of the binary search tree data structure that uses weak classifiers at each node to determine search paths. As such, it is possible to insert, search for and delete shape templates in logarithmic time.

This structure was designed to be able to store templates inside it and return a list of candidate templates that bear the highest similarity to an input shape very efficiently. The application of such a data structure in the creation of a classifier was also presented.

Experiments performed on the MNIST handwritten digit database indicate that the proposed approach is very efficient in terms of speed, performance and scalability. The advantage of our approach over the previous ones is that our method can also work online. It can, at any point, add new templates of an existing class or even entirely new classes into its knowledge. It can also forget classes, while still retaining its classification abilities on the data that are still stored inside the shape tree.

Notice that at no point during the design of the proposed data structure did we take into account that it will be used to recognize handwritten numbers, or use any preprocessing to improve results. The shape tree is a general tool for handling shapes. It simply views shapes as sets of points and tries to make the best possible match with the data it has learned.

References

[1] D.M. Gavrila and V. Philomin, "Real-time object detection for "smart" vehicles," *IEEE International Conference on Computer Vision*, vol. 01, 1999.

[2] Dariu M. Gavrila, "A Bayesian, exemplar-based approach to hierarchical shape matching," in *IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 29, Issue 8, pp. 1408-1421*, Aug. 2007.

[3] Dengsheng Zhang and Guojun Lu, "Review of shape representation and description techniques," *Pattern Recognition*, vol. 37, no. 1, pp. 1–19, 2004.

[4] Remco C. Veltkamp and Michiel Hagedoorn, "Shape similarity measures, properties and constructions," in *VISUAL '00: Proceedings of the 4th International Conference on Advances in Visual Information Systems*, London, UK, 2000, pp. 467–476, Springer-Verlag.

[5] G. Klanderman D. Huttenlocher and W.J. Rucklidge, "Comparing images using the Hausdorff distance," in *IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 15, no. 9, pp. 850-863*, Sep. 1993.

[6] W.J. Rucklidge, "Locating objects using the Hausdorff distance," in *Proceedings of Fifth International Conference on Computer Vision, pp. 457-464*, June 1995.

[7] M.-P. Dubuisson and A.K. Jain, "A modified Hausdorff distance for object matching," in *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on*, 1994, pp. 1: 566–568.

[8] Donald E. Knuth, *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*, Addison-Wesley Professional, April 1998.

[9] Gunilla Borgefors, "Distance transformations in digital images," in *Computer Vision, Graphics, and Image Processing, Volume 34 , Issue 3, pp. 344-371*, June 1986.

[10] Georgii M. Adelson-Velskii and Evgenii M. Landis, "An algorithm for the organization of information," in *Proceedings of the USSR Academy of Sciences, Volume 146, pp. 263-266*, 1962.

[11] Lecun, Y. Bottou, L. Bengio, Y. Haffner, P. "Gradient-based learning applied to document recognition" in *Proceedings of the IEEE, Volume 86, pp. 2278-2324* , 1998.

[12] Marc'Aurelio Ranzato and Christopher S. Poultney and Sumit Chopra and Yann LeCun "Efficient Learning of Sparse Representations with an Energy-Based Model" in *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems* , 2006.

[13] Borodin, Allan and El-Yaniv, Ran "Online computation and competitive analysis"