Shape Retrieval using Concavity Trees

Ossama El Badawy and Mohamed Kamel
Pattern Analysis and Machine Intelligence Laboratory
Systems Design Engineering, University of Waterloo
200 University Ave. W., Waterloo, ON, N2L 3G1, Canada
{obadawy, mkamel}@pami.uwaterloo.ca

Abstract

Concavity trees are well-known abstract structures. This paper proposes a new shape-based image retrieval method based on concavity trees. The proposed method has two main components. The first is an efficient (in terms of space and time) contour-based concavity tree extraction algorithm. The second component is a recursive concavity-tree matching algorithm that returns a distance between two trees. We demonstrate that concavity trees are able to boost the retrieval performance of two feature sets by at least 15% when tested on a database of 625 silhouette images.

1. Introduction and Background

Shape is an important feature of a pattern. In the case of binary images, it can be argued that the most prominent feature of the patterns in the image is their shape.

A concavity tree is a data structure used for describing non-convex two dimensional shapes. It was first introduced by Sklansky [10] and has since been further researched by others [2–5, 11]. A concavity tree is a rooted tree in which the root represents the whole object whose shape is to be analysed/represented. The next level of the tree contains nodes that represent concavities along the boundary of that object. The following level contains nodes, each representing one of the concavities of its parent, i.e., its meta-concavities. If an object or a concavity is itself convex, then the node representing it does not have any children. Figure 1 shows an example of an object (a) and its corresponding concavity tree (e). The object has two concavities, as reflected in level one of the tree. The concavity on the left (b) is convex, so it’s corresponding node (no. 1) has no children. The concavity on the right (c) is concave, with one convex triangular meta-concavity (d). Therefore, its corresponding node (no. 2) has one child (node no. 3) corresponding to the triangular meta-concavity. Typically, each node in a concavity tree stores attributes, or features, describing the underlying object or concavity. Concavity trees are clearly a tool for structural pattern recognition in which a pattern (a 2-D shape in our case) is decomposed into its primitive components.

The decision of whether two objects have similar shapes or not is based on both the shape of the whole object as well as the shapes of the objects’ primitives. Moreover, shape can be described using high- and low-level features. This paper presents a new method for matching the shape of two objects based on matching their corresponding labelled concavity trees. To the best of our knowledge, this work is the first to report on the applicability of concavity trees to shape-based image database retrieval.

The main goal of the proposed method is to enhance image similarity retrieval by incorporating high-level structural shape information (trees) as well as low-level shape information (labels) of the whole object and its primitives into the matching process. The analysis and matching of single-object logo images is one viable application of the proposed method. We compare the retrieval performance of moment invariants [7] and of the SCX (Solidity, Eccentricity, and eXtent) feature set [6] when they are used by themselves in a traditional nearest neighbour image retrieval problem and when the two sets are used to label concavity trees. We use a set of 50 hand-sketched rotated and scaled query images to perform the tests and report an increase of 15% in the

![Figure 1. An object (a), its concavities (b,c), meta-concavities (d), and corresponding concavity tree (e).](image-url)
case of the moment invariants and 20% in the case of the SCX feature set.

2. The proposed Method

This section is divided into two parts: tree extraction and tree matching.

2.1 Concavity tree extraction

For tree extraction, we use a new contour-based algorithm with a worst case time complexity of $O(hn)$ where $n$ is the number of the object’s contour pixel and $h$ is the height of the resulting tree. Typically, $h$ is less than ten and $n$ in the hundreds. Given the general recursive nature of trees, our algorithm is best outlined using recursion. We start with some definitions.

2.1.1 Definitions

**Definition 1** A sequence $A = [a_k]_{k=1}^n = [a_1, a_2, \cdots, a_n]$ of length $n$ is an ordered collection of elements $a_k$’s. The elements of $A$ do not have to be distinct. A subsequence $S$ of a sequence $A$ is formed by discarding some of the elements of $A$ and keeping the other elements in their original order. For a sequence $A = [a_k]_{k=1}^n$, we define $\mathcal{A}$ to be the set of all the elements of $A$ (order and multiplicity are ignored, therefore, $|\mathcal{A}| \leq n$).

**Definition 2** Two pixels are said to be direct neighbours if they share a side, and indirect neighbours if they touch only at a corner. The name neighbour denotes either type.

**Definition 3** A path $A = [a_k]_{k=1}^n$ of length $n$ ($n > 1$) is a sequence of pixels $a_1, a_2, \cdots, a_n$ such that for $1 < k \leq n$, $a_{k-1}$ is a neighbour of $a_k$. A direct path refers to a similar sequence but where the pixels are required to be direct neighbours. A closed path is a path whose first and last pixels coincide. A simple path is one where all the pixels are distinct. A simple closed path is a closed path where all pixels are distinct except for the first and last ones. A subpath $S$ of a path $A$ is a subsequence of $A$ that is also a path.

**Definition 4** A set of pixels $S$ is 8-connected if for every pair of pixels $p$ and $q$ in $S$, there is a path whose first and last pixels are $p$ and $q$ respectively and all its other pixels belong to $S$. $S$ is said to be 4-connected if the path is a direct path.

**Definition 5** A contour of an 8-connected set of pixels $S$ is a closed path that traverses the subset of all pixels in $S$ which have at least one direct neighbour not in $S$. We note that $S$ has many possible contours depending on the direction as well as the starting point of the traversal.

**Definition 6** The (2-D) convex hull of a set of points $S$ in the 2-D space is the smallest convex polygon enclosing all the points in $S$. If this polygon has $n$ vertices $v_1, v_2, \cdots, v_n$, where $v_i$ and $v_{i+1}$ are adjacent, $1 \leq i < n$, then the convex hull can be represented by a sequence $H = [h_k]_{k=1}^{n+1}$ where $h_1 = v_1, h_2 = v_2, \cdots, h_n = v_n$, and $h_{n+1} = v_1$. □

The proposed algorithm assumes that the input is a bilevel digital image $I$, foreground pixels have a value of “1”, and background pixels have a value of “0”. Moreover, the set of foreground pixels $F$ is assumed to be 8-connected and the set of background pixels $B$ is 4-connected (therefore, no holes). Without loss of generality, we can assume that the contour of $I$ is a simple closed path.

Algorithm 1 outlines the steps involved in our concavity tree extraction. The contour $C$ of the object in the image is extracted using the contour tracing algorithm from [9]. We can guarantee that $c_1 = c_n$ is a vertex of the convex hull by making sure that $c_1 = c_n$ is an extreme point of $F$. Step 2 represents the core part (function $fConcavityTree()$) of the algorithm that recursively constructs the tree.

Each call to $fConcavityTree()$ returns the convex hull $T$ of its input $C$. When $fConcavityTree()$ is first called, it is passed the contour $C$ of $F$ (a simple closed path). Any later (recursive) calls are passed (simple) subpaths of the contour of $F$. When the first call to $fConcavityTree()$ returns, $T$ will contain the convex hull tree of $F$. In addition, all nodes in $T$ will be simple paths except for the root, which will be the contour of $F$ (a simple closed path). $fConcavityTree()$ starts by initializing $T$ and setting its root to input $C$. Then the convex hull $H$ of the set of pixels in $C$ is computed using the algorithm in [8]. At each call to $fConcavityTree()$, $c_1$ and $c_n$ are guaranteed to be two adjacent vertices in the convex hull of $C$. Therefore, $a_1$ and $a_p$ will be equal to $c_1$ and $c_n$, respectively. The outer loop in Algorithm 1 works on each section $S_i$ of $C$ bound by two consecutive points in $A$. The inner loop, on the other hand, works on each (if any) concave section $V_j$ of $S_i$. Each of the $V_j$’s is in turn passed to recursive calls of $fConcavityTree()$, which will return their corresponding concavity trees to be added as children to the concavity tree formed in the calling function.

The structure of a concavity tree is invariant to “rigid” changes in the scale, rotation, or position of the patterns. However, digital boundaries tend to be irregular because of digitization and noise. Such effects usually result in small and meaningless concavities scattered randomly throughout the boundary. These irregularities obviously need to be sorted out. Figure 2 shows an example of an object (a) and its corresponding concavity tree (b). The boundary of the original image in (a) is very ragged, which resulted in a concavity tree with 66 nodes as shown in (b). The extraction algorithm is able to extract only meaningful concavities (whose depth is greater than say 0.05 of the “width” of the shape in the direction of the concavity). This results in
the smaller and more expressive concavity tree and shape in parts (d) and (c), respectively, where there are 4 concavities, each having 2 meta-concavities.

It is also worth mentioning that concavity trees can (inherently) be sensitive to small variations in the shape of the object. This anomaly can however be dealt with by a modification to the algorithm that will only extract the core part of a concavity.

Algorithm 1 Concavity tree extraction.

Notation:
- \( I \) is the input image,
- \( F \) is the set of foreground (“1”) pixels,
- \( B \) is the set of background (“0”) pixels,
- \( T \) is a rooted tree (the concavity tree), and
- \( C = \{ c_k \} \), \( 1 \leq k \leq n \) is a simple (possibly closed) path,
- \( H = \{ h_k \} \), \( 1 \leq k \leq m \) is the convex hull of \( C \),
- \( A = \{ a_k \} \), \( 1 \leq k \leq p \) is a subsequence of \( C \),
- \( S_i = \{ s_k \} \), \( 1 \leq k \leq l_i \) is a subpath of \( C \),
- \( V_{ij} = \{ v_k \} \), \( 1 \leq k \leq q_j \) is a subpath of \( S_i \),
- \( \overline{ab} \) is the line segment joining pixels \( a \) and \( b \), and
- \( d(\overline{ab}, y) \) is the shortest vertical distance between \( \overline{ab} \) and pixel \( y \).

Require: \( I \) is bilevel, \( F \) is 8-connected, and \( B \) is 4-connected.

1: \( C \leftarrow \) Extract contour of \( F \).
2: \( T \leftarrow fConcavityTree( C ) \).

Function \( T = fConcavityTree( C ) \)

3: \( T = nil \)
4: AddChild( \( T \), \( C \) ).
5: \( H \leftarrow \) the convex hull of \( C \).
6: \( A = \{ a_k \} \), \( 1 \leq k \leq p \) \leftarrow \{ c_k | c_k \in H \} \) \{All elements of \( C \) that are also in \( H \)\}
7: for \( i = 1 \) to \( p - 1 \) do
8: \( S_i = \{ s_k \} \), \( 1 \leq k \leq l_i \) \leftarrow \{ c_k, c_k, c_{k+1}, \ldots, c_{k+l_i} \} \) \{All elements of \( C \) starting at \( a_i \) and ending at \( a_{i+1} \)\}
9: Find all \( j \) subpaths \( V_{ij} = \{ v_k \} \), \( 1 \leq k \leq q_j \) of \( S_i \) such that:
10: \( q_j > 2 \), \( d(\overline{s_1 s_i}, v_1) = 0 \), \( d(\overline{s_1 s_i}, v_{q_j}) = 0 \), and
11: \( d(\overline{s_1 s_i}, \) all other \( v_k) > 0 \).
12: for all \( V_{ij} \) do
13: \( AddChild( T, fConcavityTree( V_{ij} ) ) \).
14: end for
15: end for

2.2 Concavity tree matching

We use a new tree matching algorithm suited to concavity trees. The algorithm (detailed in [11]) takes two trees and returns a distance (as a measure of dissimilarity) between them. The returned distance \( d(t, s) \) between two trees \( t \) and \( s \) whose roots respectively have \( n \) and \( m \) sub-

![Figure 2. Concavity tree example and how to remove boundary noise.](image)

trees \((m \geq n, m \geq 0, n \geq 0)\) is recursively defined as

\[
d(t, s) = ar + b \left[ \frac{1}{n} \sum_{i=1}^{n} d_i \right] + c \sum_{j=1}^{m-n} \left( \frac{1}{2} \right)^j d_j \tag{1}
\]

where \( r \) is the Euclidean distance between the feature vectors stored in the roots of \( t \) and \( s \). Constants \( a, b, \) and \( c \) are three positive real numbers such that \( a + b + c = 1 \). \( d_i \) is the tree distance between pair \( i \) of a mapping between the subtrees of \( s \) and \( t \). Finally, \( d_j \) is the tree distance between subtree \( n + j \) of \( s \) and subtree \( n + j \) of \( t \), which is a dummy tree (with no subtrees) inserted in \( t \).

The mapping between the subtrees of \( s \) and \( t \) is done based on both the features stored in the nodes in addition to other attributes, namely, the relative size of the concavity with respect to its parent and the number of nodes in the subtrees. The Euclidean distance between the root of a dummy tree and the root of real subtree is defaulted to the max possible distance between two features (1 in case of SCX and approx 25 in case of moment invariants).

The constants \( a, b, \) and \( c \) determine the weight given in the matching process to the overall shape \((a)\), the shapes of the underlying primitives (concavities) \((b)\), and the cost of inserting new nodes in the smaller tree \((c)\). Typically, the weights are equally chosen, however, one can experiment with different weights or combine them into an overall distance. For example, if \( a = 1 \), the matching with and without the tree is the same (no weights are given to subtrees).

If the maximum distance between the feature vectors stored in the nodes is bounded, one can easily make the
maximum distance between the two trees bounded as well (between 0 and 1 for example). The distance would get closer and closer to 1 when the two trees are more and more different. The limit of the distance would then be 1. What makes this characteristic of the algorithm suitable for shape matching using concavity trees is the following. If \( t_1, t_2, \) and \( t_3 \) are three trees such that \( t_1 \) has say 10 nodes, \( t_2 \) has 100 nodes, and \( t_3 \) has 110 nodes, \( d(t_1, t_2) \) is approximately the same as \( d(t_1, t_3) \) (slightly smaller). Which is desirable, since both \( t_2 \) and \( t_3 \) are much different from \( t_1 \). Another important characteristic of the algorithm is that as we move down the tree, each level gets less and less weight in the overall distance. So all the level is treated in a similar way. This is different from other tree matching algorithm where the weight of each node decreases monotonically both horizontally and vertically.

3. Experimental Results

We use a database of 625 logo images to test the performance of the system. In order to test the retrieval scheme, we used 50 hand-sketched query images. In Figure 3, parts (a) and (b), the vertical axis is the percentage of the query images that were in the first \( k \) (horizontal axis) retrievals. In the plot for example, about 30% of the 50 query images returned the correct database image as the first hit (using SCX labelled trees). From the plots, it is clear that the performance of the SCX feature was improved by 20% when used in conjunction with concavity trees. The performance of moment invariants was similarly boosted by 15%. Figure 3c shows a sample query that resulted in the correct result to move from the 15th position (using just SCX) to first position when using both trees and SCX.

4. Conclusion

This paper proposed a new method for shape-based image retrieval using a structural approach. Labelled concavity trees are used as shape descriptors and a tree matching algorithm measures similarity between shapes. We show that using the trees in the matching process boosts the retrieval performance of two feature sets by at least 15%.

References


Figure 3. Performance curves and query example.