Multiple Polyline to Polygon Matching

Mirela Tănase[1], Remco C. Veltkamp[1], and Herman Haverkort[2]

[1] Department of Computing Science, Utrecht University, The Netherlands
[2] Department of Computing Science, TU Eindhoven, The Netherlands

Abstract. We introduce a measure for computing the similarity between multiple polylines and a polygon, that can be computed in $O(km^2n^2)$ time with a straightforward dynamic programming algorithm. We then present a novel fast algorithm that runs in time $O(kmn \log mn)$. Here, m denotes the number of vertices in the polygon, and n is the total number of vertices in the k polylines that are matched against the polygon. The effectiveness of the similarity measure has been demonstrated in a part-based retrieval application with known ground-truth.

1 Introduction

The motivation for multiple polyline to polygon matching is twofold. Firstly, the matching of shapes has been done mostly by comparing them as a whole [2,8,10]. This fails when a significant part of one shape is occluded, or distorted by noise. In this paper, we address the partial shape matching problem, matching portions of two given shapes. Secondly, partial matching helps identifying similarities even when a significant portion of one shape boundary is occluded, or seriously distorted. It could also help in identifying similarities between contours of a non-rigid object in different configurations of its moving parts, like the contours of a sitting and a walking cat. Finally, partial matching helps alleviating the problem of unreliable object segmentation from images, over or undersegmentation, giving only partially correct contours.

Contribution. Firstly, we introduce a measure for computing the similarity between multiple polylines and a polygon. This similarity measure is a turning angle function-based similarity, minimized over all possible shiftings of the endpoints of the parts over the shape, and also over all independent rotations of the parts. Since we allow the parts to rotate independently, this measure could capture the similarity between contours of non-rigid objects, with parts in different relative positions. We then derive a number of non-trivial properties of the similarity measure.

Secondly, based on these properties we characterize the optimal solution that leads to a straightforward $O(km^2n^2)$-time and space dynamic programming algorithm. We then present a novel $O(kmn \log mn)$ time and space algorithm. Here, m denotes the number of vertices in the polygon, and n is the total number of vertices in the k polylines that are matched against the polygon.

Thirdly, we have experimented with a part-based retrieval application. Given a large collection of shapes and a query consisting of a set of polylines, we want to

X. Deng and D. Du (Eds.): ISAAC 2005, LNCS 3827, pp. 60–70, 2005.
© Springer-Verlag Berlin Heidelberg 2005

retrieve those shapes in the collection that best match our query. The evaluation using a known ground-truth indicates that a part-based approach improves the global matching performance for difficult categories of shapes.

2 Related Work

Arkin et al. [2] describe a metric for comparing two whole polygons that is invariant under translation, rotation and scaling. It is based on the L_2-distance between the turning functions of the two polygons, and can be computed in $O(mn \log mn)$ time, where m is the number of vertices in one polygon and n is the number of vertices in the other.

Most partial shape matching methods are based on computing local features of the contour, and then looking for correspondences between the features of the two shapes, for example points of high curvature [1,7]. Such local features-based solutions work well when the matched subparts are almost equivalent up to a transformation such as translation, rotation or scaling, because for such subparts the sequences of local features are very similar. However, parts that we perceive as similar, may have quite different local features (different number of curvature extrema for example).

Geometric hashing [12] is a method that determines if there is a transformed subset of the query point set that matches a subset of a target point set, by building a hash table in transformation space. Also the Hausdorff distance [5] allows partial matching. It is defined for arbitrary non-empty bounded and closed sets A and B as the infimum of the distance of the points in A to B and the points in B to A. Both methods are designed for partial matching, but do not easily transform to our case of matching multiple polylines to a polygon.

Partially matching the turning angle function of two polylines under scaling, translation and rotation, can be done in time $O(m^2n^2)$ [4]. Given two matches with the same squared error, the match involving the longer part of the polylines has a lower dissimilarity. The dissimilarity measure is a function of the scale, rotation, and the shift of one polyline along the other. However, this works for only two single polylines.

Latecki et al [6], establish the best correspondence of parts in a decomposition of the matched shapes. The best correspondence between the maximal convex arcs of two simplified versions of the original shapes gives the partial similarity measure between the shapes. One drawback of this approach is that the matching is done between parts of simplified shapes at "the appropriate evolution stage". How these evolution stages are identified is not indicated in their papers, though it certainly has an effect on the quality of the matching process.

3 Polylines-to-Polygon Matching

We concentrate on the problem of matching an ordered set $\{P_1, P_2, \ldots, P_k\}$ of k polylines against a polygon P. We want to compute how close an ordered set of polylines $\{P_1, P_2, \ldots, P_k\}$ is to being part of the boundary of P in the given

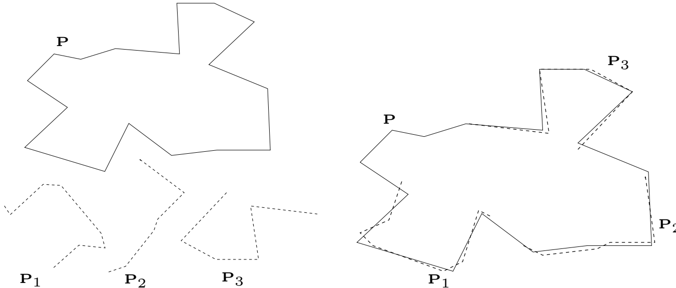

Fig. 1. Matching an ordered set $\{P_1, P_2, P_3\}$ of polylines against a polygon P

order in counter-clockwise direction around P (see figure 1). For this purpose, the polylines are rotated and shifted along the polygon P, in such a way that the pieces of the boundary of P "covered" by the k polylines are mutually disjoint except possibly at their endpoints. Note that P is a polygon and not an open polyline, only because of the intended application of part based retrieval.

3.1 Similarity Between Multiple Polylines and a Polygon

The *turning function* Θ_A of a polygon A measures the angle of the counter-clockwise tangent with respect to a reference orientation as a function of the arc-length s, measured from some reference point on the boundary of A. It is a piecewise constant function, with jumps corresponding to the vertices of A. A rotation of A by an angle θ corresponds to a shifting of Θ_A over a distance θ in the vertical direction. Moving the location of the reference point $A(0)$ over a distance $t \in [0, l_A)$ along the boundary of A corresponds to shifting Θ_A horizontally over a distance t.

Let $\Theta : [0, l] \rightarrow \mathbb{R}$ be the turning function of a polygon P of m vertices, and of perimeter length l. Since P is a closed polyline, the domain of Θ can be easily extended to the entire real line, by $\Theta(s + l) = \Theta(s) + 2\pi$. Let $\{P_1, P_2, \ldots P_k\}$ be a set of polylines, and let $\Theta_j : [0, l_j] \rightarrow \mathbb{R}$ denote the turning function of the polyline P_j of length l_j. If P_j is made of n_j segments, Θ_j is piecewise-constant with $n_j - 1$ jumps.

For simplicity of exposition, $\mathsf{f}_j(t, \theta)$ denotes the quadratic similarity between the polyline P_j and the polygon P, for a given placement (t, θ) of P_j over P: $\mathsf{f}_j(t, \theta) = \int_0^{l_j} (\Theta(s + t) - \Theta_j(s) + \theta)^2 ds$.

We assume the polylines $\{P_1, P_2, \ldots, P_k\}$ satisfy the condition $\sum_{j=1}^{k} l_j \leq l$. The similarity measure, which we denote by $\mathsf{d}(P_1, \ldots, P_k; P)$, is the square root of the sum of quadratic similarities f_j, minimized over all valid placements of $P_1, \ldots, P_k$ over P:

$$\mathsf{d}(P_1, \ldots, P_k; P) = \min_{\substack{\text{valid placements} \\ (t_1, \theta_1) \ldots (t_k, \theta_k)}} \left(\sum_{j=1}^{k} \mathsf{f}_j(t_j, \theta_j) \right)^{1/2} .$$

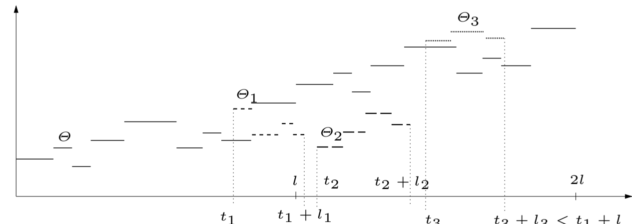

Fig. 2. To compute $\mathsf{d}(P_1, \ldots, P_k; P)$ between the polylines $P_1, \ldots, P_3$ and the polygon P, we shift the turning functions Θ_1, Θ_2, and Θ_3 horizontally and vertically over Θ

It remains to define what the valid placements are. The horizontal shifts $t_1, \ldots, t_k$ correspond to shiftings of the starting points of the polylines $P_1, \ldots, P_k$ along P. We require that the starting points of $P_1, \ldots, P_k$ are matched with points on the boundary of P in counterclockwise order around P, that is: $t_{j-1} \leq t_j$ for all $1 < j \leq k$, and $t_k \leq t_1 + l$. Furthermore, we require that the matched parts are disjoint (except possibly at their endpoints), sharpening the constraints to $t_{j-1} + l_{j-1} \leq t_j$ for all $1 < j \leq k$, and $t_k + l_k \leq t_1 + l$ (see figure 2).

The vertical shifts $\theta_1, \ldots, \theta_k$ correspond to rotations of the polylines P_1, $\ldots, P_k$ with respect to the reference orientation, and are independent of each other. Therefore, in an optimal placement the quadratic similarity between a particular polyline P_j and P depends only on the horizontal shift t_j, while the vertical shift must be optimal for the given horizontal shift. We can thus express the similarity between P_j and P for a given positioning t_j of the starting point of P_j over P as: $\mathsf{f}_j^*(t_j) = \min_{\theta \in \mathbb{R}} \mathsf{f}_j(t_j, \theta)$.

The similarity between the polylines $P_1, \ldots, P_k$ and the polygon P is thus:

$$\mathsf{d}(P_1, \ldots, P_k; P) = \min_{\substack{t_1 \in [0,l),\, t_2, \ldots, t_k \in [0,2l); \\ \forall j \in \{2,\ldots,k\}:\ t_{j-1}+l_{j-1} \leq t_j; \quad t_k + l_k \leq t_1 + l}} \left(\sum_{j=1}^{k} \mathsf{f}_j^*(t_j) \right)^{1/2} . \quad (1)$$

3.2 Properties of the Similarity Function

In this section we give a few properties of $\mathsf{f}_j^*(t)$, as functions of t, that constitute the basis of the algorithms for computing $\mathsf{d}(P_1, \ldots, P_k; P)$ in sections 3.3 and 3.4. We also give a simpler formulation of the optimization problem in the definition of $\mathsf{d}(P_1, \ldots, P_k; P)$. Arkin et al. [2] have shown that for any fixed t, the function $\mathsf{f}_j(t, \theta)$ is a quadratic convex function of θ. This implies that for a given t, the optimization problem $\min_{\theta \in \mathbb{R}} \mathsf{f}_j(t, \theta)$ has a unique solution, given by the root $\theta_j^*(t)$ of the equation $\partial \mathsf{f}_j(t, \theta)/\partial \theta = 0$. As a result:

Lemma 1. *For a given positioning t of the starting point of P_j over P, the rotation that minimizes the quadratic similarity between P_j and P is given by $\theta_j^*(t) = -\int_0^{l_j} (\Theta(s+t) - \Theta_j(s)) ds / l_j$.*

We now consider the properties of $\mathsf{f}_j^*(t) = \mathsf{f}_j(t, \theta_j^*(t))$, as a function of t.

Lemma 2. *The quadratic similarity $\mathsf{f}_j^*(t)$ has the following properties:*
i) it is periodic, with period l;
ii) it is piecewise quadratic, with mn_j breakpoints within any interval of length l; moreover, the parabolic pieces are concave.

For a proof of this and the following lemmas, see [11].

The following corollary indicates that to compute the minimum of the function f_j^*, we need to look only a discrete set of at most mn_j points.

Corollary 1. *The local minima of the function f_j^* are among the breakpoints between its parabolic pieces.*

We now give a simpler formulation of the optimization problem in the definition of $\mathsf{d}(P_1, \ldots, P_k; P)$. In order to simplify the restrictions on t_j in equation (1), we define: $\overline{\mathsf{f}}_j(t) := \mathsf{f}_j^* \left(t + \sum_{i=1}^{j-1} l_i \right)$. In other words, the function $\overline{\mathsf{f}}_j$ is a copy of f_j^*, but shifted to the left with $\sum_{i=1}^{j-1} l_i$. Obviously, $\overline{\mathsf{f}}_j$ has the same properties as f_j^*, that is: it is a piecewise quadratic function of t that has its local minima in at most mn_j breakpoints in any interval of length l. With this simple transformation of the set of functions f_j^*, the optimization problem defining $\mathsf{d}(P_1, \ldots, P_k; P)$ becomes:

$$\mathsf{d}(P_1, \ldots, P_k; P) = \min_{\substack{t_1 \in [0,l),\, t_2, \ldots, t_k \in [0,2l); \\ \forall j \in \{2, \ldots, k\}:\ t_{j-1} \leq t_j; \quad t_k \leq t_1 + l_0}} \left(\sum_{j=1}^{k} \overline{\mathsf{f}}_j(t_j) \right)^{1/2}, \qquad (2)$$

where $l_0 := l - \sum_{i=1}^{k} l_i$. Notice that if $(\overline{t}_1^*, \ldots, \overline{t}_k^*)$ is a solution to the optimization problem in equation (2), then $(t_1^*, \ldots, t_k^*)$, with $t_j^* := \overline{t}_j^* + \sum_{i=1}^{j-1} l_i$, is a solution to the optimization problem in equation (1).

3.3 Characterization of an Optimal Solution

In this section we characterize the structure of an optimal solution to the optimization problem in equation (2), and give a recursive definition of this solution. This definition forms the basis of a straightforward dynamic programming solution to the problem.

Let $(\overline{t}_1^*, \ldots, \overline{t}_k^*)$ be a solution to the optimization problem in equation (2).

Lemma 3. *The values of an optimal solution $(\overline{t}_1^*, \ldots, \overline{t}_k^*)$ are found in a discrete set of points $X \subset [0, 2l)$ of the breakpoints of the functions $\overline{\mathsf{f}}_1, ..., \overline{\mathsf{f}}_k$, plus two copies of each breakpoint: one shifted left by l_0 and one shifted right by l_0.*

We call a point in $[0, 2l)$, which is either a breakpoint of $\overline{\mathsf{f}}_1, ..., \overline{\mathsf{f}}_k$, or such a breakpoint shifted left or right by l_0, a *critical point*. Since function $\overline{\mathsf{f}}_j$ has $2mn_j$ breakpoints, the total number of critical points in $[0, 2l)$ is at most $6m \sum_{i=1}^{k} n_i = 6mn$. Let $X = \{x_0, \ldots, x_{N-1}\}$ be the set of critical points in $[0, 2l)$.

With the observations above, the optimization problem we have to solve is:

$$\mathsf{d}(P_1,\ldots,P_k;P) = \min_{\substack{t_1,\ldots,t_k \in X \\ \forall j>1:\ t_{j-1}\le t_j\,;\, t_k - t_1 \le l_0}} \left(\sum_{j=1}^{k} \overline{\mathsf{f}}_j(t_j)\right)^{1/2} . \quad (3)$$

We denote: $$D[j,a,b] = \min_{\substack{t_1,\ldots,t_j \in X \\ x_a \le t_1 \le \ldots \le t_j \le x_b}} \sum_{i=1}^{j} \overline{\mathsf{f}}_i(t_i) , \quad (4)$$

where $j \in \{1,\ldots,k\}$, $a,b \in \{0,\ldots,N-1\}$, and $a \le b$. Equation (4) describes the subproblem of matching the set $\{P_1,\ldots,P_j\}$ of j polylines to a subchain of P, starting at $P(x_a)$ and ending at $P(x_b + \sum_{i=1}^{j} l_i)$. We now show that $D[j,a,b]$ can be computed recursively. Let $(t_1^{\circledast},\ldots t_j^{\circledast})$ be an optimal solution for $D[j,a,b]$. Regarding the value of $t_j^{\circledast}$ we distinguish two cases:

- $t_j^{\circledast} = x_b$, in which case $(t_1^{\circledast},\ldots t_{j-1}^{\circledast})$ must be an optimal solution for $D[j-1,a,b]$, otherwise $(t_1^{\circledast},\ldots t_j^{\circledast})$ would not give a minimum for $D[j,a,b]$; thus in this case, $D[j,a,b] = D[j-1,a,b] + \overline{\mathsf{f}}_j(x_b)$;
- $t_j^{\circledast} \neq x_b$, in which case $(t_1^{\circledast},\ldots t_j^{\circledast})$ must be an optimal solution for $D[j,a,b-1]$; otherwise $(t_1^{\circledast},\ldots t_j^{\circledast})$ would not give a minimum for $D[j,a,b]$; thus in this case $D[j,a,b] = D[j,a,b-1]$.

We can now conclude that :

$$D[j,a,b] = \min\left(D[j-1,a,b] + \overline{\mathsf{f}}_j(x_b),\ D[j,a,b-1]\right), \text{for } j \ge 1 \wedge a \le b, \quad (5)$$

where the boundary cases are $D[0,a,b] = 0$ and $D[j,a,a-1]$ has no solution.

A solution of the optimization problem (3) is then given by

$$\mathsf{d}(P_1,\ldots,P_k;P) = \min_{x_a,x_b\in X,\ x_b-x_a\le l_0} \sqrt{D[k,a,b]} . \quad (6)$$

Equations (5) and (6) lead to a straightforward dynamic programming algorithm for computing the similarity measure $\mathsf{d}(P_1,\ldots,P_k;P)$ in $O(km^2n^2)$ time.

3.4 A Fast Algorithm

The above time bound to computate of the similarity measure $\mathsf{d}(P_1,\ldots,P_k;P)$ can be improved to $O(kmn \log mn)$. The refinement of the dynamic programming algorithm is based on the following property of equation (5):

Lemma 4. *For any polyline P_j, $j \in \{1,\ldots,k\}$, and any critical point x_b, $b \in \{0,\ldots,N-1\}$, there is a critical point x_z, $0 \le z \le b$, such that:*

i) $D[j,a,b] = D[j,a,b-1]$*, for all* $a \in \{0,...,z-1\}$*, and*
ii) $D[j,a,b] = D[j-1,a,b] + \overline{\mathsf{f}}_j(x_b)$*, for all* $a \in \{z,...,b\}$*.*

For given j and b, we consider the function $\mathcal{D}[j,b] : \{0, N-1\} \rightarrow \mathbb{R}$, with $\mathcal{D}[j,b](a) = D[j,a,b]$. Lemma 4 expresses the fact that the values of function $\mathcal{D}[j,b]$ can be obtained from $\mathcal{D}[j,b-1]$ up to some value z, and from $\mathcal{D}[j-1,b]$ (while adding $\overline{\mathsf{f}}_j(x_b)$) from this value onwards. This property allows us to improve the time bound of the dynamic programming algorithm. Instead of computing arrays of scalars $D[j,a,b]$, we will compute arrays of functions $\mathcal{D}[j,b]$. The key to success will be to represent these functions in such a way that they can be evaluated fast and $\mathcal{D}[j,b]$ can be constructed from $\mathcal{D}[j-1,b]$ and $\mathcal{D}[j,b-1]$ fast.

Algorithm FastCompute $\mathbf{d}(P_1,\ldots,P_k;P)$

1. Compute the set of critical points $X = \{x_0,\ldots,x_{N-1}\}$, and sort them
2. For all $j \in \{1,...,k\}$ and all $b \in \{0,...,N-1\}$, evaluate $\overline{\mathsf{f}}_j(x_b)$
3. $ZERO \leftarrow$ a function that always evaluates to zero (see Lemma 5)
4. $INFINITY \leftarrow$ a function that always evaluates to ∞ (see Lemma 5)
5. $MIN \leftarrow \infty$
6. $a \leftarrow 0$
7. **for** $j \leftarrow 1$ **to** k **do**
8. $\quad\mathcal{D}[j,-1] \leftarrow INFINITY$
9. **for** $b \leftarrow 0$ **to** $N-1$ **do**
10. $\quad\mathcal{D}[0,b] \leftarrow ZERO$
11. $\quad$**for** $j \leftarrow 1$ **to** k **do**
12. $\quad\quad$Construct $\mathcal{D}[j,b]$ from $\mathcal{D}[j-1,b]$ and $\mathcal{D}[j,b-1]$
13. $\quad$**while** $x_a < x_b - l_0$ **do**
14. $\quad\quad a \leftarrow a+1$
15. $\quad$val $\leftarrow$ evaluation of $\mathcal{D}[k,b](a)$
16. $\quad MIN \leftarrow \min(\text{val}, MIN)$
17. **return** $\sqrt{MIN}$

The running time of this algorithm depends on how the functions $\mathcal{D}[j,b]$ are represented. In order to make especially steps 12 and 15 of the above algorithm efficient, we represent the functions $\mathcal{D}[j,b]$ by means of balanced binary trees. Asano et al. [3] used an idea similar in spirit.

An efficient representation for function $\mathcal{D}[j,b]$. We now describe the tree $\mathcal{T}_{j,b}$ used for storing the function $\mathcal{D}[j,b]$. Each node ν of $\mathcal{T}_{j,b}$ is associated with an interval $[a^-_\nu, a^+_\nu]$, with $0 \leq a^-_\nu \leq a^+_\nu \leq N-1$. The root ρ is associated with the full domain, that is: $a^-_\rho = 0$ and $a^+_\rho = N-1$. Each node ν with $a^-_\nu < a^+_\nu$ is an internal node that has a split value $a_\nu = \lfloor (a^-_\nu + a^+_\nu)/2 \rfloor$ associated with it. Its left and right children are associated with $[a^-_\nu, a_\nu]$ and $[a_\nu + 1, a^+_\nu]$, respectively. Each node ν with $a^-_\nu = a^+_\nu$ is a leaf of the tree, with $a_\nu = a^-_\nu = a^+_\nu$. For any index a of a critical point x_a, we will denote the leaf ν that has $a_\nu = a$ by λ_a. Note that so far, the tree looks exactly the same for each function $\mathcal{D}[j,b]$: they are balanced binary trees with N leaves, and $\log N$ height. Moreover, all trees have the same associated intervals, and split values in their corresponding nodes. With each node ν we also store a weight w_ν, such that $\mathcal{T}_{j,b}$ has the following property: $\mathcal{D}[j,b](a)$ is the sum of the weights on the path from the tree root to the leaf λ_a. Such a representation of a function $\mathcal{D}[j,b]$ is not unique. Furthermore, we store with each node ν a value m_ν which is the sum of the weights on the

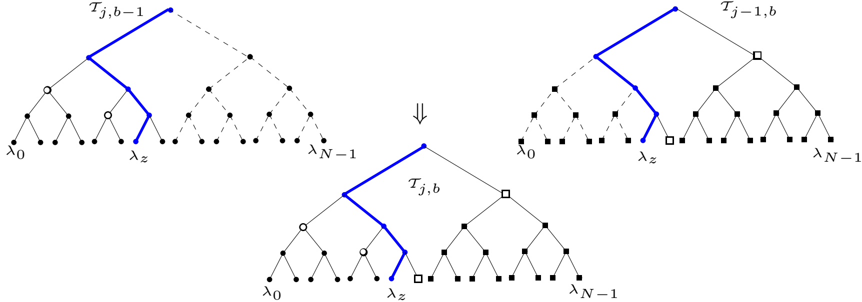

Fig. 3. The tree $\mathcal{T}_{j,b}$ is contructed from $\mathcal{T}_{j,b-1}$ and $\mathcal{T}_{j-1,b}$ by creating new nodes along the path from the root to the leaf λ_z, and adopting the subtrees to the left of the path from $\mathcal{T}_{j,b-1}$, and the subtrees to the right of the path from $\mathcal{T}_{j-1,b}$

path from the left child of ν to the leaf λ_{a_ν}, that is: the rightmost descendant of the left child of ν.

Lemma 5. *The data structure $\mathcal{T}_{j,b}$ for the representation of function $\mathcal{D}[j,b]$ can be operated on such that:*

(i) The representation of a zero-function (i.e. a function that always evaluates to zero) can be constructed in $O(N)$ time. Also the representation of a function that always evaluates to ∞ can be constructed in $O(N)$ time.
(ii) Given $\mathcal{T}_{j,b}$ of $\mathcal{D}[j,b]$, evaluating function $\mathcal{D}[j,b](a)$ takes $O(\log N)$ time.
(iii) Given $\mathcal{T}_{j-1,b}$ and $\mathcal{T}_{j,b-1}$ of the functions $\mathcal{D}[j-1,b]$ and $\mathcal{D}[j,b-1]$, respectively, a representation $\mathcal{T}_{j,b}$ of $\mathcal{D}[j,b]$ can be computed in $O(\log N)$ time.

Proof. We restrict ourselves to item (iii), the main element of the solution.

To construct $\mathcal{T}_{j,b}$ from $\mathcal{T}_{j,b-1}$ and $\mathcal{T}_{j-1,b}$ efficiently, we take the following approach. We find the sequences of left and right turns that lead from the root of the trees down to the leaf λ_z, where z is defined as in lemma 4. Note that the sequences of left and right turns are the same in the trees $\mathcal{T}_{j,b}$, $\mathcal{T}_{j,b-1}$, and $\mathcal{T}_{j-1,b}$, only the weights on the path differ. Though we do not compute z explicitly, we will show below that we are able to construct the path from the root of the tree to the leaf λ_z corresponding to z, by identifying, based on the stored weights, at each node along this path whether the path continues left or right.

Lemma 4 tells us that for each leaf left of λ_z, the total weight on the path to the root in $\mathcal{T}_{j,b}$ must be the same as the total weight on the corresponding path in $\mathcal{T}_{j,b-1}$. At λ_z itself and right of λ_z, the total weights to the root in $\mathcal{T}_{j,b}$ must equal those in $\mathcal{T}_{j-1,b}$, plus $\overline{\mathsf{f}}_j(x_z)$. We construct the tree $\mathcal{T}_{j,b}$ with these properties as follows. We start building $\mathcal{T}_{j,b}$ by constructing a root ρ. If the path to λ_z goes into the right subtree, we adopt as left child of ρ the corresponding left child ν of the root from $\mathcal{T}_{j,b-1}$. There is no need to copy ν: we just add a pointer to it. Furthermore, we set the weight of ρ equal to the weight of the root of $\mathcal{T}_{j,b-1}$. If the path to λ_z goes into the left subtree, we adopt the right child from $\mathcal{T}_{j-1,b}$ and take the weight of ρ from there, now adding $\overline{\mathsf{f}}_j(x_z)$.

Then we make a new root for the other subtree of the root ρ, i.e. the one that contains λ_z, and continue the construction process in that subtree. Every time we go into the left branch, we adopt the right child from $\mathcal{T}_{j-1,b}$, and every time we go into the right branch, we adopt the left child from $\mathcal{T}_{j,b-1}$ (see figure 3). For every constructed node ν, we set its weight w_ν so that the total weight of ν and its ancestors equals the total weight of the corresponding nodes in the tree from which we adopt ν's child — if the subtree adopted comes from $\mathcal{T}_{j-1,b}$, we increase w_ν by $\overline{\mathsf{f}}_j(x_z)$.

By keeping track of the accumulated weights on the path down from the root in all the trees, we can set the weight of each newly constructed node ν correctly in constant time per node. The accumulated weights together with the stored weights for the paths down to left childrens' rightmost descendants, also allow us to decide in constant time which is better: $\mathcal{D}[j, b-1](a_\nu)$ or $\mathcal{D}[j-1, b](a_\nu)+\overline{\mathsf{f}}_j(x_z)$. This will tell us if λ_z is to be found in the left or in the right subtree of ν.

The complete construction process only takes $O(1)$ time for each node on the path from ρ to λ_z. Since the trees are perfectly balanced, this path has only $O(\log N)$ nodes, so that $\mathcal{T}_{j,b}$ is constructed in time $O(\log N)$. □

Theorem 1. *The similarity* $\mathsf{d}(P_1, \ldots, P_k; P)$ *between* k *polylines* $\{P_1, \ldots, P_j\}$ *with* n *vertices in total, and a polygon* P *with* m *vertices, can be computed in* $O(kmn \log(mn))$ *time using* $O(kmn \log(mn))$ *storage.*

Proof. We use algorithm Fastcompute $\mathsf{d}(P_1, \ldots, P_k; P)$ with the data structure described above. Step 1 and 2 of the algorithm can be executed in $O(kmn + mn \log n)$ time. From lemma 5, we have that the zero-function $ZERO$ can be constructed in $O(N)$ time (line 3). Similarly, the infinity-function $INFINITY$ can be constructed in $O(N)$ time (line 4). Lemma 5 also insures that constructing $\mathcal{D}[j, b]$ from $\mathcal{D}[j-1, b]$ and $\mathcal{D}[j, b-1]$ (line 12) takes $O(\log N)$ time, and that the evaluation of $\mathcal{D}[k, b](a)$ (line 15) takes $O(\log N)$ time. Notice that no node is ever edited after it has been constructed. Thus, the total running time of the above algorithm will be dominated by $O(kN)$ executions of line 12, taking in total $O(kN \log N) = O(kmn \log(mn))$ time.

Apart from the function values of $\overline{\mathsf{f}}_j$ computed in step 2, we have to store the $ZERO$ and the $INFINITY$ function. All these require $O(kN) = O(kmn)$ storage. Notice that any of the functions constructed in step 12 requires only storing $O(\log(N)) = O(\log(mn))$ new nodes and pointers to nodes in previously computed trees, and thus we need $O(kmn \log(mn))$ for all the trees computed in step 12. So the total storage required by the algorithm is $O(kmn \log(mn))$. □

We note that the problem resembles a general edit distance type approximate string matching [9]. Global string matching under a general edit distance error model can be done by dynamic programming in $O(kN)$ time, where k and N represent the lengths of the two strings. The same time complexity can be achieved for partial string matching through a standard "assign first line to zero" trick [9]. This however does not apply here due to the condition $x_b - x_a \leq l_0$.

4 Experimental Results

Our algorithm has been implemented in C++ and is evaluated in a part-based shape retrieval application (see `http://give-lab.cs.uu.nl/Matching/Mtam/`) with the Core Experiment "CE-Shape-1" part B test set devised by the MPEG-7 group to measure the performance of similarity-based retrieval for shape descriptors. This test set consists of 1400 images: 70 shape classes, with 20 images per class. The shape descriptor selected by MPEG-7 to represent a closed contour of a 2D object or region in an image is based on the Curvature Scale Space (CSS) representation [8]. We compared our matching to the CSS method, as well as to matching the global contours with turning angle functions (GTA).

The performance of each shape descriptor was measured using the "bull's-eye" percentage: the percentage of retrieved images belonging to the same class among the top 40 matches (twice the class size). These experimental results indicate that for those classes with a low performance of the CSS matching, our approach consistently performs better. See figure 4 for two examples. The emphasis of this paper lies on the algorithmic aspects, but for a rigorous experimental evaluation, see [11]. The running time for a single query on the MPEG-7 test set of 1400 images is typically about one second on a 2 GHz PC.

CSS/GTA Query Image	Part-based Query Parts	Bull's Eye Score		
		CSS	GTA	MPP
beetle-20		10	30	65
ray-3		15	15	70

Fig. 4. A comparison of the Curvature Scale Space (CSS), Global Turning Angle function (GTA), and our Multiple Polyline to Polygon (MPP) matching (in %)

Acknowledgements. We thank Veli Mäkinen for the partial string matching reference, and Geert-Jan Giezeman for programming support. This research was supported by the FP6 IST projects 511572-2 PROFI and 506766 AIM@SHAPE, and the Dutch Science Foundation (NWO) project 612.061.006 MINDSHADE.

References

1. N. Ansari and E. J. Delp. Partial shape recognition: A landmark-based approach. *PAMI*, 12:470–483, 1990.
2. E. Arkin, L. Chew, D. Huttenlocher, K. Kedem, and J. Mitchell. An efficiently computable metric for comparing polygonal shapes. *PAMI*, 13:209–215, 1991.
3. T. Asano, M. de Berg, O. Cheong, H. Everett, H.J. Haverkort, N. Katoh, and A. Wolff. Optimal spanners for axis-aligned rectangles. *CGTA*, 30(1):59–77, 2005.
4. Scott D. Cohen and Leonidas J. Guibas. Partial matching of planar polylines under similarity transformations. In *Proc. SODA*, pages 777–786, 1997.

5. D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the hausdorff distance. *PAMI*, 15:850–863, 1993.
6. L. J. Latecki, R. Lakämper, and D. Wolter. Shape similarity and visual parts. In *Proc. Int. Conf. Discrete Geometry for Computer Imagery*, pages 34–51, 2003.
7. H.-C. Liu and M. D. Srinath. Partial shape classification using contour matching in distance transformation. *PAMI*, 12(11):1072–1079, 1990.
8. F. Mokhtarian, S. Abbasi, and J. Kittler. Efficient and robust retrieval by shape content through curvature scale space. In *Workshop on Image DataBases and MultiMedia Search*, pages 35–42, 1996.
9. Gonzala Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
10. K. Siddiqi, A. Shokoufandeh, S.J. Dickinson, and S.W. Zucker. Shock graphs and shape matching. *IJCV*, 55(1):13–32, 1999.
11. Mirela Tanase. *Shape Deomposition and Retrieval*. PhD thesis, Utrecht University, Department of Computer Science, February 2005.
12. Haim Wolfson and Isidore Rigoutsos. Geometric hashing: an overview. *IEEE Computational Science & Engineering*, pages 10–21, October-December 1997.