Piecewise linear digital curve representation and compression using graph theory and a line segment alphabet

András Hajdu*, and Ioannis Pitas, *Senior Member, IEEE*

Department of Informatics

Aristotle University of Thessaloniki

Box 451, 54124 Thessaloniki, Greece

Tel: +30-231-099-6361

Fax: +30-231-099-8453

e-mail: {*hajdua, pitas@aiia.csd.auth.gr*}

Abstract

The use of an alphabet of line segments to compose a curve is a possible approach for curve data compression. Many approaches are developed with the drawback that they can process simple curves only. Curves having more sophisticated topology with self-intersections can be handled by methods considering recursive decomposition of the canvas containing the curve. In this paper, we propose a graph theory based algorithm for tracing the curve directly to eliminate the decomposition needs. This approach obviously improves the compression performance, as longer line segments can be used. We tune our method further by selecting optimal turns at junctions during tracing the curve. We assign a polygon approximation to the curve which consists of letters coming from an alphabet of line segments. We also discuss how other application fields can take advantage of the provided curve description scheme.

Index Terms

curve compression, Euler graph, Chinese Postman problem, optimal curve tracing, curve partitioning.

EDICS Category: COD-LSYI, MOD-MRPH

Piecewise linear digital curve representation and compression using graph theory and a line segment alphabet

I. Introduction

Digital planar curves are used in several fields of computer graphics, discrete geometry and digital image analysis. Many results have been produced regarding their geometric behavior since [1]. A special topic is digital curve compression. Besides simple techniques like chain coding, a usual way is to partition the curve into straight line segments [2] for compression. These methods usually focus on simple curves with no self-intersections, and assume the preliminary knowledge on the sequential order of the curve points. The state-of-the-art approach JBEAM [3] considers an alphabet of short line segments (called beamlets) to compose the curve. This method divides the binary image containing the curve using quadtree decomposition till having a *single* linear curve segment in every quadtree cell that can be substituted by a beamlet. The advantage of this approach is that any curve can be handled by sufficiently fine quadtree decomposition. However, a drawback is the obligation of decomposing subsequently, when a cell contains such segments that already could be coded separately.

In this paper, we propose a graph theoretical approach to trace curves having arbitrary topology to obtain better compression performance, when splitting the curve into straight line segments. Because of the tracing step, the proposed method has better compression performance than JBEAM [3]. The main improvement lies in the fact that we perform a complete tracing of the curve instead of decomposing its storing canvas recursively, while only line segments remain in the quadtree cells.

The structure of this paper is as follows. In section II we recall the graph theoretical background that serves as a basis for our approach in tracing curves. We also explain how the suitable graph representation of the digital curve is obtained. Section III describes how the tracing is optimized regarding coding the curve with straight line segments. The method selected for compression is presented in section IV. Section V contains our comparative analyses with other state-of-the-art

approaches. We explain some variants of the basic approach in section VI with highlighting their advantages and drawbacks. Finally, some open issues and other possible applications are discussed in section VII.

II. TRACING CURVES USING GRAPH THEORY

In this section we recall some notions and results of graph and curve theory that we apply to trace a curve and also some techniques that were considered to obtain the corresponding graph representation of the curve.

A. Graph theoretical background

A *graph* G is defined as a pair (V, E), where V is a set of *vertices*, and $E \subseteq V \times V = \{\{u, v\} \mid u, v \in V\}$ is a set of *edges* between the vertices. As we use graph representations of curves, we focus on *undirected* graphs, so $\forall u, v \in V : \{u, v\} = \{v, u\}$ holds. To cover a wide class of curves, we allow *loops* (edges of type $\{u, u\}$) and *multiple* edges (more edges between two vertices). The *degree* of a vertex is the number of edges containing the vertex. A *path* is a list of vertices $\{u_1, u_2, \ldots, u_n\}$ having edges between any two consecutive vertices: $\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{n-1}, u_n\}$, with $u_1 = u_n$ in the case of a *route* (closed path). G is *connected*, if any two of its vertices have a path connecting them. A path through G which includes every edge exactly once is called an *Euler path* (or an *Euler route* if the start and end vertices coincide) [4], [5]. Note that any Euler route is also an Euler path. G is an *Euler graph*, if it contains an Euler path through all of its edges. An *Euler decomposition* of G has the form $G = \bigcup_{i=1}^{n} G_i$ such that all the G_i's are disjoint Euler graphs (in the sense that they cannot contain the same edge). We recall some well-known facts on Euler graphs and their decomposition (see e.g. [6], [7]):

i) Every Euler graph is connected.

ii) A connected graph contains an Euler route iff all of its vertices have even degree. The route can start from any vertex.

iii) A connected graph contains an Euler path iff at most two of its vertices have odd degree. If there are two vertices with odd degree, the path starts from either of them and ends in the other.

iv) Every connected graph has an Euler decomposition into disjoint Euler graphs.

B. Assignment of a graph to a digital curve

The definition of simple curves in the Euclidean space was given by P. Urysohn in 1923 and K. Menger in 1932 independently (see [8] for a review). The curves were classified based on the branching indeces of the curve points, where a branching index of a curve point is equal to the number of curve segments meeting at the given point. The adequate mathematical formulation for the Euclidean space can be found in [8], [9]. For the discrete domain $\mathbb{Z}^2$, this definition can also be adapted using the well-known 8-neighboring relation. More precisely (see [8], [9]):

- a digital curve C has branching index $B(p) > 0$ at its point $p \in C$ iff exactly $B(p)$ 8-neighbors of p belong to C,
- $p \in C$ is a regular point iff $B(p) = 2$,
- $p \in C$ is a branch point iff $B(p) \geq 3$,
- $p \in C$ is an end point iff $B(p) = 1$,
- the 2D digital curve C is simple iff all of its points are regular.

To make graph theoretical algorithms be applicable to digital curves, we also need a precise concept for a junction (see [8], [9]):

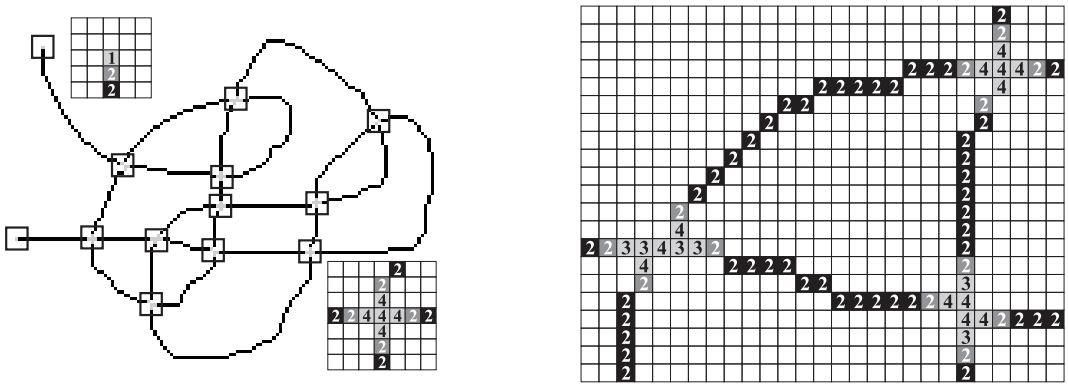
- an 8-connected region of branch points is called a junction. The branching index of a junction $J \subseteq C$ is the number of regular or end points of C being 8-neighbors of any of the branch points of J.

We mention here that this classic approach for defining the junctions of digital curves might be restrictive in some applications, which issue will be discussed in section VII. Using all the above definitions, we are ready to assign an abstract curve graph to a curve (see [9]):

- the abstract curve graph $G_C = (V_C, E_C)$ of the curve C is an undirected graph, where the vertices in V_C are either junctions or end points of C. Two vertices are connected by an edge iff the corresponding junctions or end points are 8-connected. Thus, the degree of a vertex is just the branching index of the corresponding junction.

As for the technical details of the extraction of an abstract curve graph, we consider 1 pixel wide curves. If the input curve is not 1 pixel wide, we can apply a preliminary thinning step on it. To determine the edge set E_C, we locate the end points of the edges as regular points being 8-neighbors to junctions (if both of their 8-neighbors are branch points, the edge is degenerated having length 1). Then, the edge end points are organized into pairs (edges) based on the condition

that an 8-connected path can be found between them whose elements are regular points. Figure 1a depicts the result of locating end points and junctions (shown framed, in light gray), while 1b take a closer look for the selection of edge end points (dark gray), and for the edges defined by them. These figures also indicate the branching indices of the curve points.



(a) (b)

Fig. 1. Locating vertices and extracting the edges for the abstract curve graph $G_C = (V_C, E_C)$. (a) Input test planar curve C with its end points and junctions to compose V_C are framed. (b) Extracting edges for E_C via locating edge end points (dark grey) and connecting them with 8-paths.

Note that loops and multiple edges are also handled by this approach without any difficulties. To find the 8-paths between edge end points we can use the recursive Floodfill8 algorithm [10] starting from the edge end points. The simplified abstract curve graph representation of the curve in Figure 1a is shown in Figure 2. The vertex indices are assigned in the order of the vertex scanning procedure in the figure.

Now we are ready to summarize our main approach in the following curve tracing (CT) algorithm:

CT algorithm

---

1) Extract the abstract curve graph $G_C = (V_C, E_C)$ of the curve C.
2) Create an Euler decomposition $\bigcup_{i=1}^{n} C_i$ of C based on G_C.
3) Trace all the C_i's separately through their Euler paths.

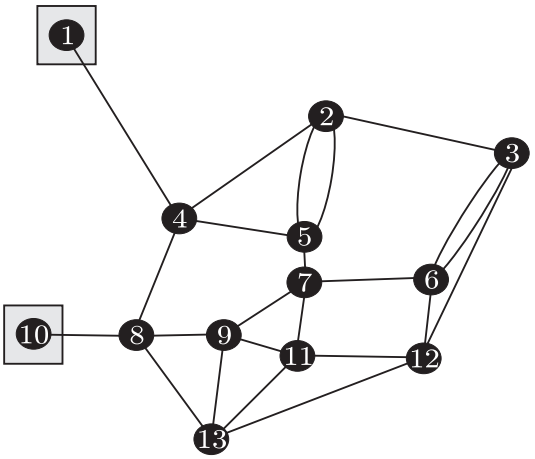


Fig. 2. The simplified abstract curve graph of the curve shown in Figure 1a with vertices of odd degree framed.

III. Optimized tracing for compression

The first step to trace an Euler curve is to locate a starting vertex according to statement *iii)* in section II-A. We check the vertices and select one having odd degree. If all the degrees are even, we can choose an arbitrary vertex to start from. Then we take an edge from the starting vertex to initialize the tracer. For example, in the graph shown in Figure 2 two vertices (1, and 10) have odd degrees. Thus, the Euler path should start from vertex 1 to finish at vertex 10, or vice versa.

As more Euler paths may exist, we have to decide which edge to take next, when reaching junctions. As our intention is to substitute the curve part with straight line segments for curve compression, the natural decision is to go on straight ahead at junctions. Thus, let us assume that we arrive at an edge end point E_0 at a junction that has edge end points $E_0, E_1, \ldots, E_k$ not visited yet. We calculate the centroid of the junction by:

$$\overline{E} = \frac{1}{k+1} \sum_{j=0}^{k} E_j. \tag{1}$$

$\overline{E}$ can be rounded to have integer coordinates, or considered as a real valued vector, as well. Let α_i denote the angle $\angle E_0 \overline{E} E_i$ for $i = 1, \ldots, k$. The curve traversal direction is chosen to correspond to the edge end point E_l for which:

$$|180^\circ - \alpha_l| = \min_{j=1}^{k} \{|180^\circ - \alpha_j|\}. \tag{2}$$

To extract a path between E_0 and E_l we can use the Floodfill8 algorithm again. Now, we have to start Floodfill8 from E_0 to flood the vertex points of the junction with selecting the path with minimal length. See Figure 3 for an example on how the decision is made to trace through a junction based on the above discussion. The selected path through the junction contains E_0, $\overline{E}$, and E_2 and indicated by $\times$ marks in the figure.

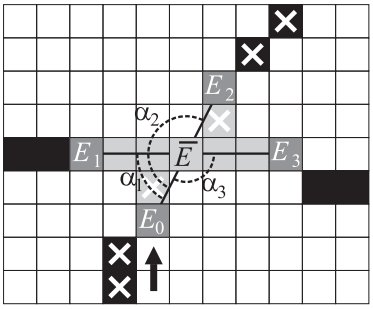


Fig. 3. Optimal curve tracing through a junction with finding the most straight direction and connecting corresponding edge end points.

Using this junction traversal decision, we are able to trace the whole curve along one of its Euler paths. The traced curve is composed by concatenating the edges with the short segment going through the vertices. For the complete tracing see Figure 4, where beside the start and end point of the Euler path, arrow heads show the optimal directions at the curve junctions. Borrowing the vertex numbering from Figure 2, the Euler path is:

$$\{1, 4, 5, 2, 4, 8, 13, 11, 7, 5, 2, 3, 6, 7, 9, 13, 12, 6, 3, 12, 11, 9, 8, 10\}.$$

During the extraction of an Euler path, we have to note that we are not always free in choosing the most straight direction at junctions. Fleury's algorithm [12] guarantees to find an Euler path, and we have to combine our junction traversal method with the classic graph theoretical recommendations (see e.g. [12], [13]), which are briefly:

• Always leave one edge available to get back to the starting vertex or to the other odd vertex.

• Do not use an edge to go to a vertex unless there is another edge available to leave it.

Note that the graph theoretical recommendations have higher priority than optimal traversal selection to guarantee the proper Euler path.

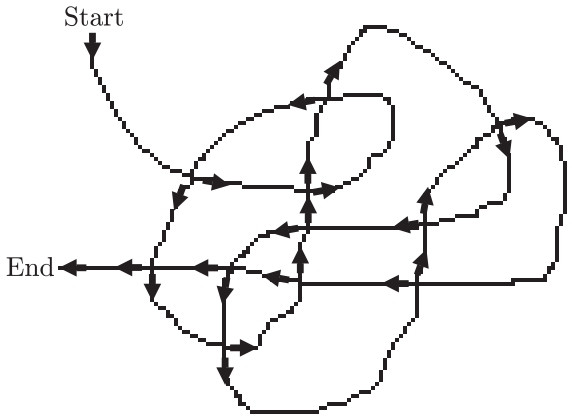


Fig. 4. Tracing the whole curve by choosing optimal directions at junctions.

IV. Curve compression by an alphabet of line segments

We can choose from a vast number of techniques to partition a curve into digital straight line segments [2]. These techniques can be classified as *offline* (the curve is examined globally to find an optimal partitioning), or *online* (the curve is decomposed into line segments during its traversal). Though our proposed approach is suitable for both tasks, we discuss an online coding possibility here. To partition the curve into digital straight segments we use a linear online method presented in [14].

To obtain a coding scheme from the straight line decomposition, we replace all the produced linear curve segments by elements of an alphabet of line segments. We create a finite alphabet Λ whose letters are digital line segments of all possible orientations having length at most T pixels. As an obvious consequence, we have to stop processing the curve when the maximal segment length T is reached and we have to look for the next segment, even if the coded one would continue straight. Moreover, to keep the cardinality of Λ small, we consider unique straight line segments to connect two points. For this purpose, we consider the Bresenham line drawing algorithm [15] to create the letters of Λ. Note that this way we allow some information loss, since the Bresenham segments may slightly differ from the ones extracted during the online curve segmentation process. On the other hand, these differences are really minor perceptually, since digital straightness is our essential requirement. It is easy to prove that the cardinality of

Λ, and the number of bits needed for coding a letter can be calculated as:

$$|\Lambda| = 4T(T-1), \text{ and } \log_2 |\Lambda| \leq 2(\log_2 T + 1), \tag{3}$$

respectively. As an example for such an alphabet see Figure 5 for $T = 6$ with the letters shown only for the domain $0 \leq y \leq x$. Thus, the alphabet also contains the letters obtained from the shown set by rotations of $0°$, $\pm 90°$, $180°$, and by mirrorings to the lines $x = 0$, $y = 0$, $y = x$, and $y = -x$, respectively.

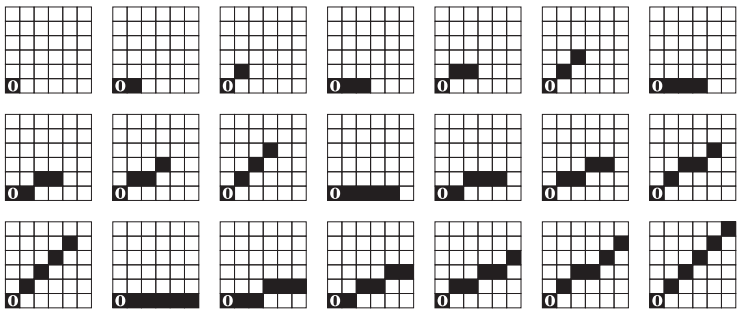


Fig. 5. An example alphabet Λ for $T = 6$ (line segments of length at most 6). Only the letters belonging to the domain $0 \leq y \leq x$ are depicted.

To check the compression efficiency of our method in comparison with other state-of-the-art methods, we considered a dataset of typical test curves shown in Figure 6a-d. To demonstrate the extendability of our method to such curves that cannot be traversed with a simple Euler path, we consider another example shown in Figure 6e.

Note that the graph representation of Figure 6e contains four vertices having odd parities. Consequently, it should be decomposed at least into two Euler paths. In our example, starting from any of the odd vertices and judging by the linearity criteria at the junctions, we decompose the original curve into the two segments shown in Figure 6f and Figure 6g, respectively. To compress the original curve, we compress these Euler paths separately.

V. Comparative analyses

A. Comparing with JBEAM

To test and compare the compression efficiency of our method, we fixed the following setup. We used $T = 32$ as a threshold for the maximum line segment length for all the test curves, and

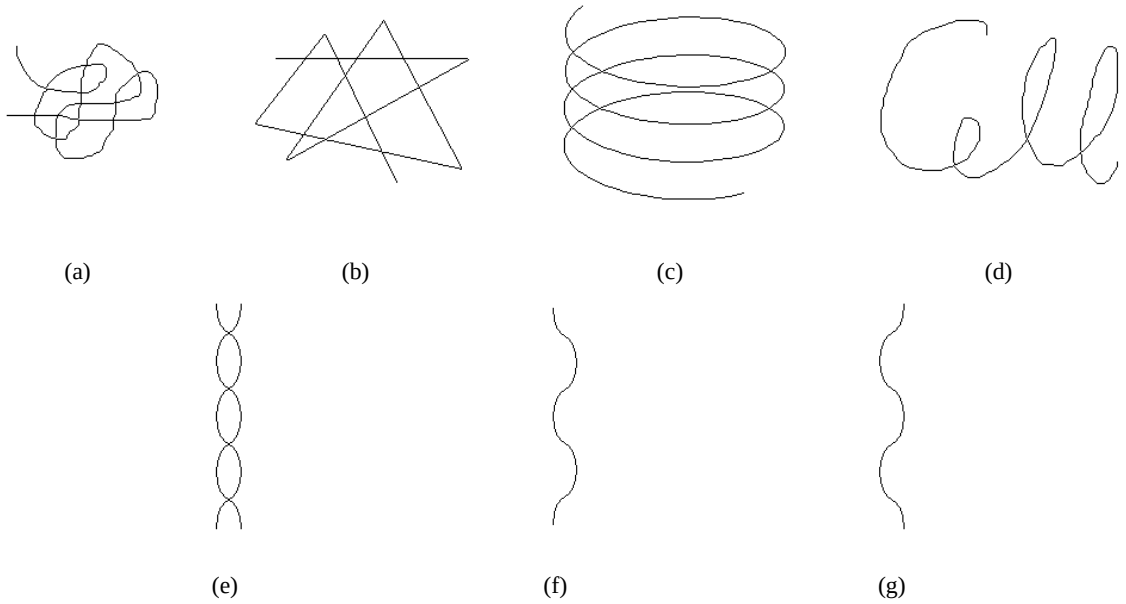


Fig. 6. Test curves of different types. (a) General. (b) Lines. (c) Spring. (d) Script. (e) Non-Euler. (f)-(g) Euler paths to compose the Non-Euler curve.

considered the default lossy JBEAM parametrization [3]. Our experimental results are shown in Table I.

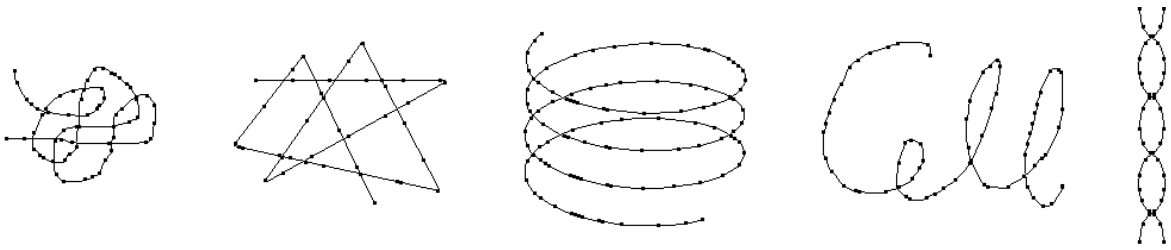
Test curve	# of pixels	JBEAM (# of bits)	Proposed method (CT)	
			# of bits	# of segments
General	2127	1586	744	62
Lines	2745	1398	468	39
Spring	4113	2308	1224	102
Script	2511	1419	828	69
Non-Euler	1242	834	$2 \times 240 = 480$	$2 \times 20 = 40$

TABLE I

COMPARATIVE QUANTITATIVE RESULTS AGAINST JBEAM.

We can conclude that the proposed method has a 50% improvement on average in compression against JBEAM. Figure 7 depicts the coding results for our sample curves. We marked the end points of the line segments found by our coding method. For the sake of completeness, we mention that the coordinates of the starting point of every Euler path should be stored, as well.

However, we ignored this issue in our calculations, since it produces only insignificant increase in the number of bits of the compressed curve.



(a) (b) (c) (d) (e)

Fig. 7. The partitioning of the test curves into line segments. (a) General. (b) Lines. (c) Spring. (d) Script. (e) Non-Euler.

B. MPEG-4 contour-based shape coding

Within MPEG-4, a vertex-based shape approximation was developed to code the outline of shapes [16]. For image/video transmission, the usual task is to transmit region-like shapes, thus, their boundaries can be represented by simple closed curves. In MPEG-4, the boundary of the shape is approximated by a polygon for lossy shape coding. For lossless shape coding, the polygon approximation degenerates to chain coding [17]. The polygon is found through a recursive splitting process that starts with the longest axis (diameter) of the shape as an initial polygon $\overline{V_0V_1}$. A polygon segment $\overline{V_kV_{k+1}}$ is associated with the curve part C_k composed by the points $c_{k,i}$ with $0 \leq i \leq I$. The approximation error at C_k is defined as:

$$d_{\max}(k) = \max_{0 \leq i \leq I} d(\overline{V_kV_{k+1}}, c_{k,i}) \tag{4}$$

using the Euclidean distance d. Now, if $d_{\max}(k) > d^*_{\max}$ for a fixed threshold $d^*_{\max}$, then $c_{k,j}$ is selected as a new polygon vertex, where $d(\overline{V_kV_{k+1}}, c_{k,j}) = \max_{0 \leq i \leq I} d(\overline{V_kV_{k+1}}, c_{k,i})$. In other words, we recursively split those polygon segments which are not sufficiently close to the curve. The boundary point having largest distance from the polygon segment is selected as a new vertex. For better coding performance, we shift the polygon vertices to re-index them so that the first and last vertices have largest difference between their horizontal or vertical coordinates. After storing the position of V_0, each remaining vertex position is encoded by a difference vector from

Test curve	# of pixels	# of segments		# of bits		# of bits with Huffmann coding	
		MPEG-4	Proposed	MPEG-4	Proposed	MPEG-4	Proposed
Running	238	41	36	492	408	239	207
Hungary	593	88	86	1056	1032	439	435
Walking	753	81	72	972	864	410	378

TABLE II

COMPARATIVE RESULTS WITH MPEG-4 CODING.

its predecessor in the form $V_d = V_k - V_{k+1}$. Finally, the components of the difference vectors are coded further by variable-length (e.g. Huffmann [11]) coding tables.

Though our main intention in the paper is to code non-simple curves, our scheme can be naturally applied to simple closed contours, as well. Technically, the obvious way is to break the connection between any two points of the curve, which leads to an abstract curve graph representation having a single edge. Since straightness is the main condition for the proposed method, we have to find a corresponding MPEG-4 threshold d^*_{max}. From [2] we know that a finite arc is a digital straight segment, if and only if, its points are between or on a pair of parallel lines having a main diagonal distance of at most $\sqrt{2}$. Thus, $d^*_{\text{max}} \leq \frac{\sqrt{2}}{2}$ should hold to make the approximated curve partitions be straight line segments, as well. However, in our experiments we found that the above MPEG-4 approach leads to worse performance than the proposed technique with respect to the number of polygon segments with $d^*_{\text{max}} = \frac{\sqrt{2}}{2}$. The main reason is that the selection of new vertices is ambiguous, since more curve points can reside at the largest distance from the approximating polygon. We found that approximately the same performance can be achieved by having $d^*_{\text{max}} = 1$. Though with $d^*_{\text{max}} = 1$ we slightly hurt the criterion of straightness, the subjective perceptual performance is reported to be acceptable for a human observer up to $d^*_{\text{max}} = 1.4$ [16]. For a comparative analysis, we consider some simple closed curves (shape boundaries) shown in Figure 8. The polygon vertices found by the MPEG-4 method with $d^*_{\text{max}} = 1$ are also marked with large dots in the figure.

Table II contains the corresponding quantitative results. We also present data to see the additional coding improvement we can gain by applying a consequent Huffmann coding.

There are some more recommendations to achieve minor improvements using the MPEG-4

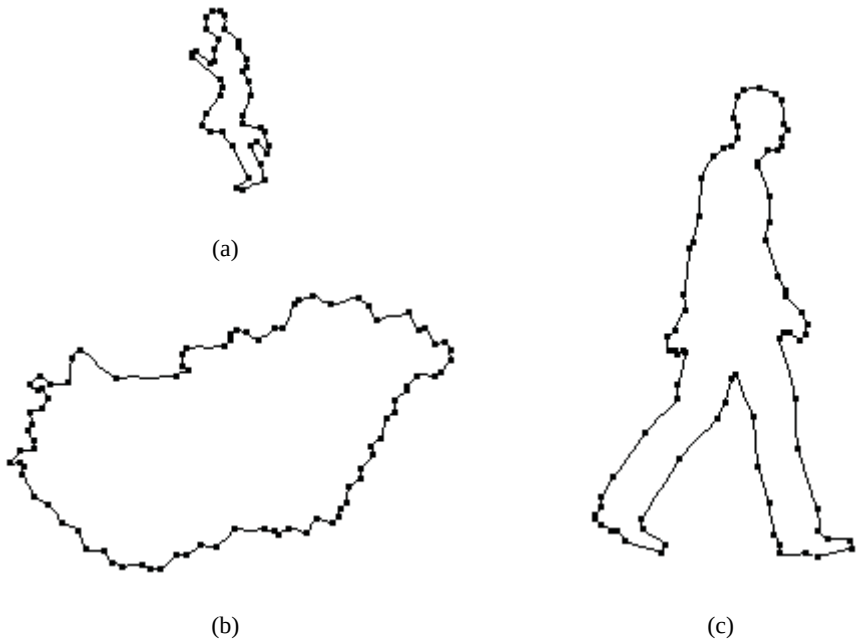


Fig. 8. Closed curves for a comparative study with MPEG-4 technique. The vertices of the approximating MPEG-4 polygons are also indicated. (a) Running. (b) Hungary. (c) Walking.

coder which would be adaptable to the proposed scheme, as well. On one hand, for lossy shape coding, the selection of the vertices on the object boundary might not be optimal. Therefore, the vertices can be shifted by 1 pixel within a neighborhood of size 3×3. On the other hand, the maximal length of the alphabet elements can be fixed dynamically as the length of the longest polygon segment found. Note that, in this case, this length information should be transmitted, as well.

VI. Alternative compression approaches

We can apply slightly different approaches for the compression of the curves regarding the ones presented in the previous section. In all the cases we consider the same alphabet Λ of line segments introduced previously, and the same online method to partition the curves into line segments.

A. Compressing edges separately

The process presented in Section II-B is suitable to extract the edges of an abstract curve graph, where the edges are actually curve segments. We might as well execute our compression approach at edge level without the intention to perform any tracing of the curve. Though this simple approach avoids the oversegmentation of the quadtree cells considered by JBEAM [3], it has several drawbacks. In this case, curve segments (edges) are compressed separately, so we have to store the coordinates of the start pixel of each curve segment for appropriate geometric positioning. Moreover, since no junction point information is stored, we have to connect the consecutive curve segments during the decoding process (e.g. using the Bresenham algorithm [15]), which may lead to curve distortion at junctions.

The compression rate can be calculated easily, as now we have to summarize simply the number of line segments that are needed to compress the separate curve segments. Also for further use, we introduce the corresponding weight function $w : E_C \to \mathbb{N}$ for the edges. Let $u, v \in G_C$. Then the weight (cost) of the edge $\{u, v\}$ is defined as:

$$w(\{u, v\}) = \# \text{ of line segments needed to compress } \{u, v\}. \tag{5}$$

Thus, the curve can be stored using:

$$\sum_{\{u,v\}\in E} w(\{u, v\}) \tag{6}$$

letters from Λ. For an example, see Figure 9, where the w weights are shown for the General test curve.

Moreover, check Table III to have a comparison for all our test curves. From the table we can see that besides covering junctions, the CT method presented in Section IV has better compression performance.

B. Curve compression without partitioning

As we discussed in section IV, our basic approach considers an Euler decomposition of the curve to avoid redundancy in coding edges. However, we can skip this step by finding a path that may contain some edges more then once. This sort of graph theoretical problem is known as the Chinese Postman Problem (CPP for short) [18], [19].

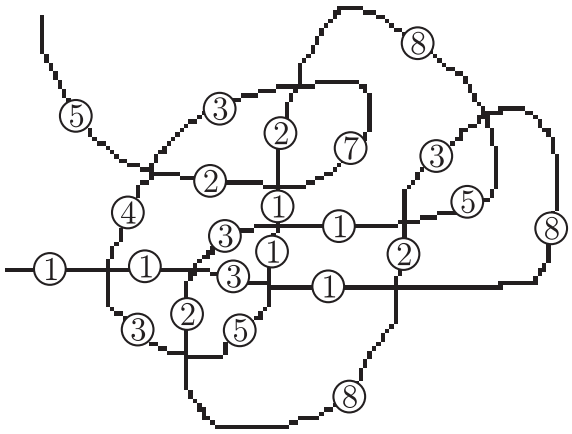


Fig. 9. Weighting the abstract curve graph G_C. The weights of the edges (curve segments) correspond to the number of line segments needed to code the edges, respectively.

Test curve	Compressing edges separately		
	# of edges	# of bits	# of segments
General	23	948	79
Lines	21	600	50
Spring	15	1236	103
Script	8	864	72
Non-Euler	10	672	56

TABLE III

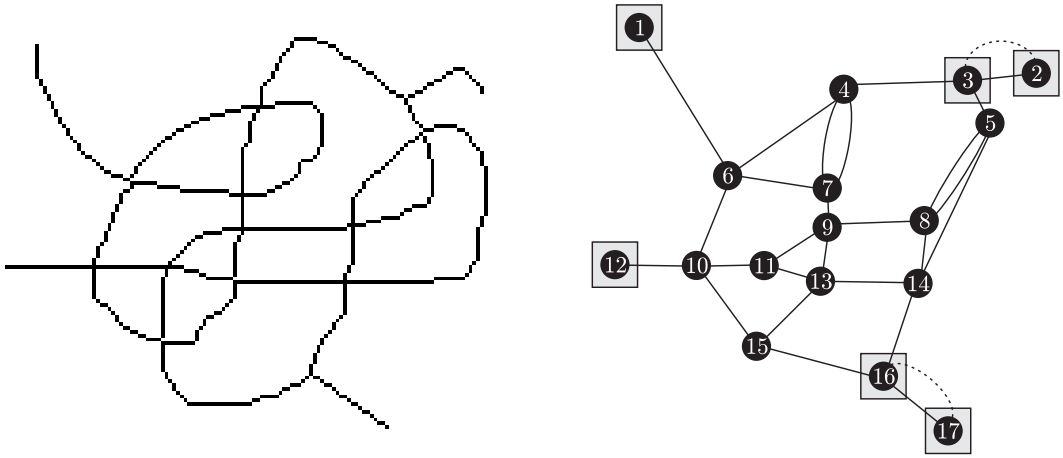
EXPERIMENTAL RESULTS OF SEPARATE (CURVE PART) EDGE COMPRESSION.

In this case, we consider the abstract curve graph G_C as a weighted one, where as a natural choice we can use the weight function defined in (5). In general, we can solve a CPP problem through the following steps:

1) Calculate the shortest length of a path between every two vertices with odd parities using the Floyd algorithm [20].
2) Perform a perfect maximum matching [11] for the set of vertices with odd parities. That is, organize these vertices into pairs in such a way that the sum of the corresponding path lengths is minimal.

3) Find the shortest paths using Dijkstra's algorithm [11], [21] between the vertex pairs found in step 2), and add these paths as artificial edges to the graph.
4) Now the graph becomes an Euler one, so an Euler path could be found, as discussed in section III.

Let us consider the example in Figure 10a which actually depicts a slightly different variant of the General test curve.



(a) (b)

Fig. 10. An example for the CPP algorithm. (a) Non-Euler variant of the General curve. (b) Adding new edges (dashed) to have an Euler abstract curve graph.

Now, as can be seen from its simplified graph representation in Figure 10b, this curve is not an Euler one, as it has six odd degree vertices (1,2,3,12,16,17). After calculating edge weights according to (5), we can organize these vertices into the optimal pairs (1,12), (2,3), (16,17). The respective distance of these pairs are 10, 3, 6 (with summing the edge weights in the shortest path between them). Thus step 3) of the CPP algorithm will provide the (redundant) artificial edges $\{2, 3\}$ and $\{16, 17\}$ to be added to the graph. These edges are shown with dashed lines in Figure 10b. Note that, this way, the graph becomes an Euler one, having vertices 1 and 12 of odd degree. Finally, using the approach discussed in section III, the following Euler path is found in our example:

$$\{1, 6, 7, 4, 6, 10, 15, 13, 9, 7, 4, 3, 2, 3, 5, 8, 9, 11, 15, 16, 17, 16, 14, 8, 5, 14, 13, 11, 10, 12\}.$$

This path is also shown in Figure 11 using bidirectional arrows, where the path traverses edges $\{2, 3\}$ and $\{16, 17\}$ twice.

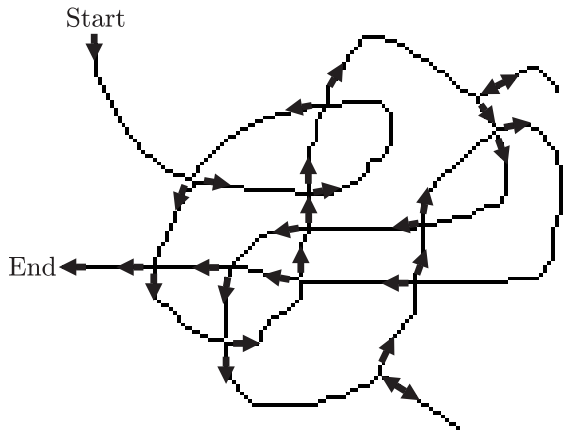


Fig. 11. Tracing a non-Euler curve using the CPP algorithm (bidirectional arrows correspond edges to be taken twice).

The compression performance obviously drops using the CPP approach in comparison with the CT one, as some curves segments are stored more than once. For example, while the CT algorithm needs $62+6+3 = 71$ line segments to code the curve in Figure 10a, the CPP approach needs $62+2\times 6+2\times 3 = 80$ ones. Note that, according to Table I, the CT algorithm needed 62 line segments without edges $\{2, 3\}$ and $\{16, 17\}$ for the compression. Besides its simple idea, the advantage of the CPP method lies in the fact that the curve should not be broken into Euler parts, and thus only the coordinates of the start pixel should be stored.

VII. Conclusion and discussion

In this paper, we propose curve compression techniques based on graph theory and the use of a line segment alphabet. We use the abstract curve graph representation of the curve to be compressed by locating junctions and considering curve parts as graph edges. The abstract curve graph can be decomposed into Euler subgraphs, which can be traced separately. To replace the curve parts with linear segments, we fix an alphabet of bounded line segments. We also discuss some variants of the proposed approach. Experimental results are presented in comparison with state-of-the-art curve compression approaches, as well.

We must mention that JBEAM [3] also realizes a progressive approach in curve compression. That is, when only a part of bits has been received, the digital curve can be recovered approximately. Our method does not have a direct support for progressive encoding, however, some kind of progressivity could be reached by re-ordering the transmitted Bresenham segments. For example, to have an impression about the shape of the curve, we can send the segments having indices equidistantly sampled along the graph route. Another approach may relate to the case, when junctions are more important. Then we can transmit those segments first which contain or touch branch points. In all these cases, additionally data are needed to be transmitted, since the segments are no longer consequent ones. The necessary extra amount of such data is an open issue.

In our approach, it is a key consideration to keep linearity when replacing curve segments with Bresenham ones. Minor deficiencies may occur from the fact that we use different methodology for judging on linearity and replacing with fixed linear line segments. In other words, if we fix two endpoints, more curve segments between them can be judged as linear but replaced with the same Bresenham one. This approach naturally leads to some kind of distortion, however the linear behavior is preserved perceptually. For a given Bresenham segment, the expected rate of the distortion is related to the possible number of segments judged as linear between its endpoints. Consequently, longer segments could cause larger distortion on average, however, we need smaller number of longer segments to code the curve, since the number of the curve points is fixed. Accordingly, though it might be a future topic to clarify the precise relation between the maximal length of the Bresenham alphabet and the distortion, we do not expect remarkable differences.

In several applications, we can encounter with such intersections where the local tangents of the intersecting curve segments almost coincide. For example, we can think of an intersection of nearly horizontal lines. In the Euclidean case, we would not have any difficulties, but in the digital domain the intersection consists of more than one pixel. One possibility to overcome this difficulty is to revise the definition of a junction, and to apply a distance threshold $\mathcal{T}$. Then we can collect branch points closer than $\mathcal{T}$, and merge into a junction the regular points between them. In other words, we decide upon the allowed closeness of the tangents. The threshold value $\mathcal{T}$ can be adjusted according to the expected behavior of the curves in the application.

The steps we considered in our approach can have individual importance in other application

fields, as well. Our intention to “untie” curves can have impact in curve watermarking [22], [23], where the capability to provide the input data in terms of few large blocks is highly welcome. The ability of tracing complex curves looks feasible in reconstructing hand-written text or figures, similarly e.g. to [24]. The compression method can also be used to efficiently store the boundaries of shapes, e.g. the templates of a human silhouette database.

ACKNOWLEDGEMENT

This work was partially supported by the project SHARE: Mobile Support for Rescue Forces, Integrating Multiple Modes of Interaction, EU FP6 Information Society Technologies, Contract Number FP6-004218. The authors are also grateful to the reviewers for their thorough work to improve the content of the paper.

REFERENCES

[1] A. Rosenfeld, “Arcs and Curves in Digital Pictures.” *Journal of ACM*, vol. 20, no. 1, pp. 81–87, 1973.

[2] R. Klette, and A. Rosenfeld, “Digital straightness - a review.” *Disc. Appl. Math.*, vol. 139, no. 1-3, pp. 197–230, 2004.

[3] X. Huo, and J. Chen, “JBEAM: multiscale curve coding via beamlets.” *IEEE Transactions on Image Processing*, vol. 14, no. 11, pp. 1665–1677, 2005.

[4] N.L. Biggs, E.K. Lloyd, and R.J. Wilson, *Graph Theory.* Calendon Press, Oxford, 1998.

[5] L. Euler, “Solutio problematis ad geometrian situs pertinentis.” *Commentarii academiae scientarum Petropolitanae*, vol. 8, pp. 128–140, 1736.

[6] H. Fleischner, “Eulerian Graphs and Related Topics. Part 1. Vol. 1.” *volume 45 of Annals of Discrete Mathematics*, North-Holland Publishing Co., Amsterdam, 1990.

[7] H. Fleischner, “Eulerian Graphs and Related Topics. Part 1. Vol. 2.” *volume 50 of Annals of Discrete Mathematics*, North-Holland Publishing Co., Amsterdam, 1991.

[8] R. Klette, and A. Rosenfeld, *Digital Geometry - Geometric Methods for Picture Analysis.* Morgan Kaufmann, San Francisco, 2004.

[9] G. Klette, “Branch Voxels and Junctions in 3D skeletons.” in *Proc. Int. Conf. 10th IWCIA 2006*, Berlin, Germany, *Lecture Notes in Computer Science*, vol. 4040, Springer, Berlin, pp. 34–44, 2006.

[10] T. Pavlidis, *Algorithms for Graphics and Image Processing.* Computer Science Press, Rockville, MD, 1982.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms.* Second Edition. MIT Press and McGraw-Hill, 2001.

[12] Fleury, “Deux problemes de geometrie de situation.” *Journal de mathematiques elementaires*, pp. 257–261, 1883.

[13] S. Skiena, *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Reading, MA: Addison-Wesley, 1990.

[14] I. Debled-Rennesson, and J. Reveillés, “A linear algorithm for segmentation of digital curves.” *Int. Journ. Patt. Recogn. and Artif. Intell.*, vol. 9, pp. 635–662, 1995.

[15] J.E. Bresenham, “Algorithm for computer control of a digital plotter.” *IBM Systems Journal*, pp. 25–30, 1965.

[16] A.K. Katsaggelos, L.P. Kondi, F.W. Meier, J. Ostermann, and G.M. Schuster, "MPEG-4 and rate-distortion-based shape-coding techniques." *Proceedings of the IEEE*, vol. 86, no. 6, pp. 1126–1154, 1998.

[17] J. Ostermann, "Methodologies used for evaluation of video tools and algorithms in MPEG-4." *Signal Process.*, vol. 9, pp. 343–365, 1997.

[18] Mei-Ko Kwan, "Graphic programming using odd or even points." *Chinese Math.*, vol. 1, pp. 273–277, 1962.

[19] M.G. Guan, "A survey on the Chinese postman problem." *J. Math. Res. Exposition*, vol. 4, no. 1, pp. 113–119, 1984.

[20] R.W. Floyd, "Algorithm 97: Shortest Path." *Communications of the ACM*, vol. 5, no. 6, pp. 345, 1962.

[21] E.W. Dijkstra, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[22] H. Gou, and M. Wu, "Fingerprinting curves." *Lecture Notes on Computer Science*, vol. 3304, pp. 13–28, 2004.

[23] V. Solachidis, and I.Pitas, "Watermarking polygonal lines using Fourier descriptors." *IEEE Computer Graphics and Applications*, vol. 24, no. 3, pp. 44–51, 2004.

[24] E. Saund, "Finding perceptually closed paths in sketches and drawings." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 4, pp. 475–491, 2003.

András Hajdu received his MSc degree in Mathematics from the Lajos Kossuth University, Hungary, in 1996. He obtained his PhD degree in Mathematics and Computer Science from the University of Debrecen, Hungary, in 2003. He worked as a Post Doc researcher for the Artificial Intelligence Information Analysis Laboratory, Dept. of Informatics, Aristotle University of Thessaloniki in 2005-2006. From 2001 he served as Assistant Lecturer and since 2003 he has been an Assistant Professor at the University of Debrecen. He is a member of the Janos Bolyai Mathematical Society, John von Neumann Computer Society (Hungary), Public Body of the Hungarian Academy of Sciences, and the Hungarian Association for Image Analysis and Pattern Recognition. He has authored or co-authored 17 journal papers and 43 conference papers. His main interest lies in discrete mathematics with applications in digital image processing.

Ioannis Pitas received the Diploma of Electrical Engineering in 1980 and the PhD degree in Electrical Engineering in 1985 both from the Aristotle University of Thessaloniki, Greece. Since 1994, he has been a Professor at the Department of Informatics, Aristotle University of Thessaloniki. From 1980 to 1993 he served as Scientific Assistant, Lecturer, Assistant Professor, and Associate Professor in the Department of Electrical and Computer Engineering at the same University. He served as a Visiting Research Associate or Visiting Assistant Professor at several Universities. He has published 145 journal papers, 360 conference papers and contributed in 18 books in his areas of interest and edited or co-authored another 5. He has also been an invited speaker and/or member of the program committee of several scientific conferences and workshops. In the past he served as Associate Editor or co-Editor of four international journals and General or Technical Chair of three international conferences. His current interests are in the areas of digital image and video processing and analysis, multidimensional signal processing, watermarking and computer vision.