Image Anal Stereol 2019;38:227-235
Original Research Paper

doi:10.5566/ias.2095

EXTENDED ALGORITHM TO CONSTRUCT A QUADTREE FROM FREEMAN CHAIN CODE IN FOUR DIRECTIONS

ANDREJ NERAT✉, DAMJAN STRNAD, EVA ZUPANČIČ AND BORUT ŽALIK

University of Maribor, Faculty of Electrical Engineering and Computer Science, Koroška cesta 46, SI-2000 Maribor, Slovenia
e-mail: andrej.nerat@um.si, damjan.strnad@um.si, eva.zupancic@um.si, borut.zalik@um.si

(Received January 4, 2019; revised July 12, 2019; accepted July 18, 2019)

ABSTRACT

This paper introduces improvements to the algorithm that was proposed in 2001 by Chen and Chen. The algorithm constructs a quadtree directly from Freeman chain code in four directions. We have improved the algorithm in two ways: Firstly, a time efficient solution using the space filling Z-order curve is proposed for a self-intersection case that was not considered by Chen and Chen. Secondly, the algorithm is expanded to handle geometric objects containing holes. The computational efficiency of the extended algorithm was confirmed by the experiments.

Keywords: chain code, quadtree, chain code to quadtree conversion, space filling curve, Z-order curve.

INTRODUCTION

An efficient and unambiguous representation of geometric objects has attracted a lot of interest in the past. A spatial enumeration, a constructive solid geometry, and a boundary representation were identified as the most important representation methods (Mortensen, 1985; Mäntylä, 1987; Hoffmann, 1989). Due to their different properties, conversion algorithms between them were identified as extremely important (Anand and Knott, 1991; Krishnan *et al.*, 1996). The problem in 2D is considerably less demanding, especially if already rasterized geometric objects are considered. In this case, a geometric object is embedded in a raster space and represented either with a quadtree or with a chain code. The quadtree, introduced by Finkel and Bentley (1974), supports the spatial enumeration representation by dividing the 2D rasterized space recursively into four equally sized regions/quadrants (see Fig. 1). The quadrants that are not fully occupied by the object, are subdivided further.

The chain code, on the other hand, represents the rasterized geometric object by a set of instructions, which describe the walk-about along the rasterized objects's boundary. The chain code was introduced by Freeman (1961). He employed 4- or 8-connectivity to generate the so-called Freeman chain code in four (F4) or eight (F8) directions. Later, the vertex chain code (Bribiesca, 1999), three-orthogonal chain code (Sánchez-Cruz and Rodríguez-Dagnino, 2005), Unsigned Manhattan chain code (Žalik *et al.*, 2016b), slope chain code (Bribiesca and Bribiesca-Contreras, 2014) and extended slope chain code (Bribiesca *et al.*, 2019) were presented, too. Chain codes were also developed for 3D (Sánchez-Cruz *et al.*, 2014; Lemus *et al.*, 2014). However, Freeman's chain code remains the most frequently used, as it is the most intuitive. The F4 encoding scheme, for example, consists of 4 symbols, *i.e.*, F4= $\{i \mid i = 0,1,2,3\}$, where $i \times 90^o$ denotes the angle of the next movement in the counter-clockwise direction with respect to the positive

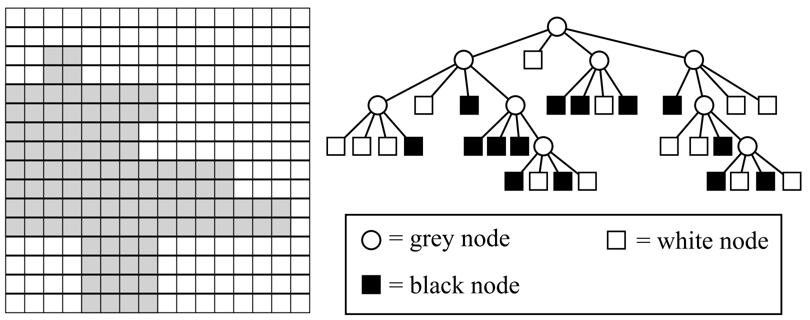

Figure 1. *Raster image and its quadtree representation.*

x-axis. The object from the Fig. 1, where its boundary is represented with F4 chain code, is shown in Fig. 2. The obtained chain code is 00330000332330000033000332222222333322221111 2222111111110011, where the starting position is determined with the blue marker. Although chain codes already represent concise description of the geometric objects, they can be compressed further (Žalik *et al.*, 2016a; 2018).

Since the properties of both representations differ, conversion methods from the chain codes to the quadtrees have been proposed (the conversion from the quadtree to the chain code is also known (Shih and Wong, 2001)). Most approaches work in two phases (Samet, 1980; Webber, 1984; Mark and Abel, 1985; Lattanzi and Shaffer, 1991), although, Chen and Chen (2001) proposed a more efficient one-phase method, which constructs a quadtree directly from F4 chain code by inserting only valid nodes into the quadtree. The method works only for the objects without holes. Unfortunately, the transformation used to convert the original chain code into its coarser (*i.e.*, lower resolution) forms may produce self-intersecting chain codes, which are not handled properly by the original method. The self-intersection, as considered by Chen and Chen, is not the self-intersection as understood in general, but rather the self-touching (see Fig. 3). It may appear during the chain code transformation from the finer to the coarser resolution levels. Chen and Chen considered only the case shown in Fig. 3a (let us name it Type_A), but the possibility from Fig. 3b (Type_B) remains unresolved. The aim of this work is to upgrade the Chen and Chen method for handling both self-intersection cases and to extend it to cope also with geometric objects containing holes.

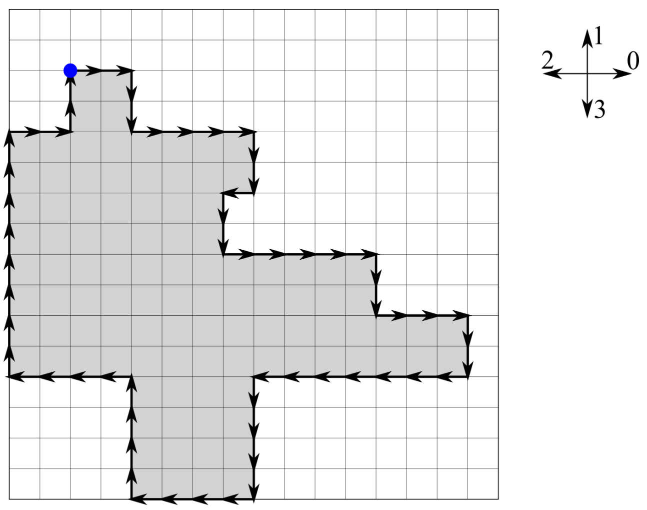

Figure 2. *Object represented with the F4 chain code.*

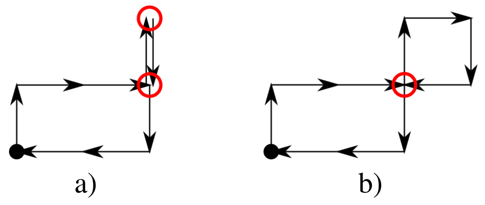

Figure 3. *The self-intersecting chain code of Type_A (a) and Type_B (b) with marked points of contacts.*

The paper is organised as follows. Section "Materials and methods" first provides a comprehensive overview of previous methods with the emphasis on the method proposed by Chen and Chen (2001). The scenarios, in which self-intersection chain codes may appear, are then addressed, and an adequate solution is proposed. Finally, the improved method is extended to the objects containing holes. Section "Results" provides experimental results, while Section "Discussion" concludes the paper.

MATERIALS AND METHODS

BACKGROUND

Two-phase approaches (Samet, 1980; Webber, 1984; Mark and Abel, 1985; Lattanzi and Shaffer, 1991) build firstly an initial quadtree as follows: The chain code is used to determine the black nodes in the quadtree leaves. The inner grey quadtreee nodes are then generated from the leaves towards the root. In the second phase, the missing descendants of the grey nodes are added, and coloured in regard to the neighbouring nodes. Finally, the quadtree is pruned by eliminating the descendants of the same colour.

The single phase method, introduced by Chen and Chen (2001), constructs the quadtree directly. The method is introduced briefly in the continuation. Let Q denote the quadtree and QN_i a set of quadtree nodes, constructed upon the chain code CC_i at the i-th resolution level, $i = 0, 1, 2, \ldots, M$. CC_M is the input Freeman chain code in four directions (F4). Q contains the root, the leaf nodes (*i.e.*, the black nodes), and the inner nodes (*i.e.*, the grey nodes). A set of inner nodes QIN_n determines the path from the root to the black node $n \in QN_i$. The method constructs the quadtree by adding sets of black and grey nodes starting at level M towards the root. A set of rules collected in a lookup table is used to transform the current chain code CC_i into its coarser representation CC_{i-1}. The terms current grid, coarser grid, and coarser grid points are used during this transformation (see Fig. 4). For building the quadtree, Chen and Chen (2001) defined the following functions:

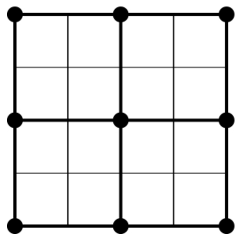

Figure 4. *Current grid, coarser grid (in thick lines) and coarser grid points (black circles).*

- SELECTSTARTINGPOINT; the selection of a starting point in the current chain code should assure that the starting point lies on the coarser grid point. Unfortunately, this cannot be achieved in all cases by traversing the input chain code and moving the starting point accordingly. Because of this, three ways of adjusting the starting point were suggested:
  - The simplest situation arises when the first chain code symbol from CC_i moves to the grid point of the coarser representation (Fig. 5a). In this case the starting point is moved to the coarser grid point, and the first chain code symbol is moved to the end of the chain code CC_i;
  - If the first chain code symbol moves to the centre of the coarser grid cell (Fig. 5b), the starting point becomes the coarser grid point at the right side of the chain code CC_i in regards to the chain code orientation. The current chain code CC_i is modified as follows: the chain code symbol moving from the coarser start point to the previous starting point is prepended, while the opposite chain code symbol is appended to the chain code (see arrows in red in Fig. 5b).
  - The most demanding situation occurs when the first chain code symbol starts in the centre of the coarser grid (Fig. 5c). In this case the starting point is moved to the edge and the first chain code symbol is moved to the end of the chain code. After that the next symbol in the chain code satisfies the first or the second case.

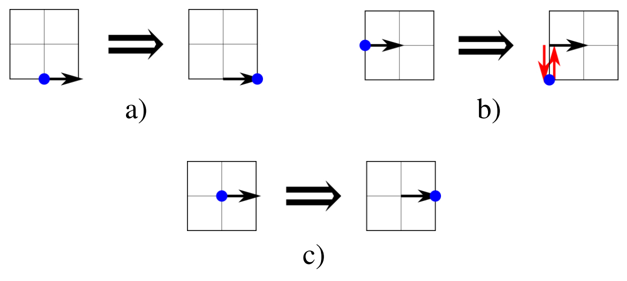

Figure 5. *Determining the starting point of the coarser chain code.*

- FETCHFOURINPUTPATTERN and ADJUSTINPUTPATTERN; There are two ways of transforming four chain code symbols into their coarser representation (see Fig. 6). When the sequence terminates in a coarser grid point (Fig. 6a), the algorithm proceeds with the next four CC_i symbols. However, if the sequence of the four chain code symbols finishes in the next cell of the coarser grid, the remaining part of the previous operation should be adjusted (the red part of sequence in Fig. 6b).

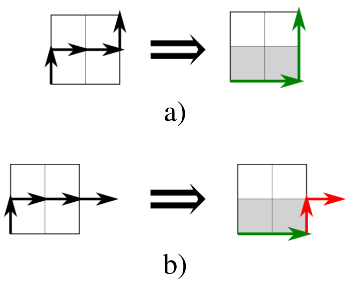

Figure 6. *The sequence of four chain code symbols may stop in the grid point of the coarser representation (a) or in the neighbouring cell of the coarser grid (b).*

- TABLELOOKUP; the next four chain code symbols are used as a key into a lookup table. The coarser representation of the considered chain code sequence (Fig. 6 shown in green) and the possible adjustment code for the unused part of the sequence (Fig. 6b in red) are obtained. Finally, the extracted black nodes (marked in grey in Fig. 6) are obtained using their Morton order and they are added to QN_i.
- GENERATEQTNODE; after the black leaf nodes are added to QN_i, the paths of grey nodes from the root to these nodes are formed. The paths obtained in the previous steps are skipped.
- CANCELLATION; the algorithm requires non self-intersecting chain code to work correctly. Because the reduction of chain code CC_i into its coarser representation CC_{i-1} can yield the so-called self-intersecting parts, the cleanup of the obtained chain code has to be performed at each iteration. The method, described by Chen and Chen (2001) considers only Type_A of F4 chain code self-intersection, *i.e.*, the sequences 02, 20, 13, and 31 (see Fig. 3), which can be solved easily by removing such pairs. However, as demonstrated in the next section, the self-intersections can also take another form, which cannot be detected by a simple comparison of the neighbouring chain code symbols.

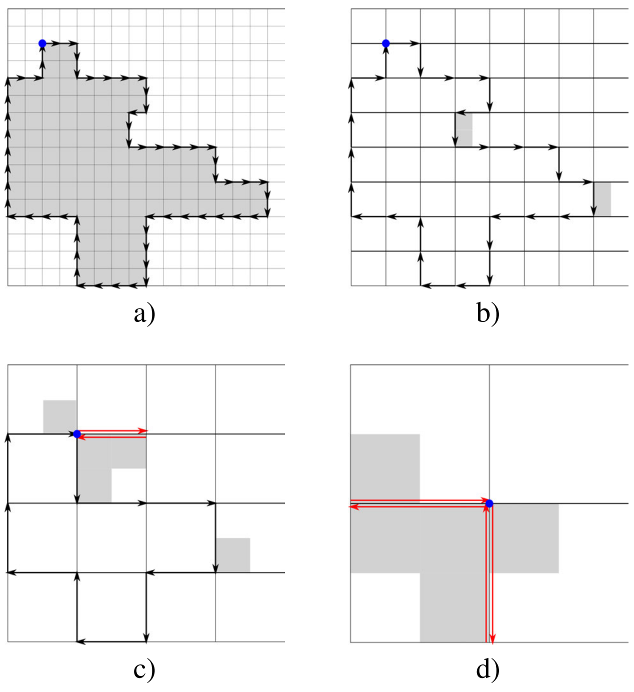

Figure 7. *Building a quad tree from chain code.*

In Fig. 7 the quadtree building process from the chain code is shown. The initial object is shown in Fig. 7a, where the starting position is identified with the blue marker. After the first iteration, the coarser chain code is formed, as shown in Fig. 7b. In Fig. 7c it can be seen that the starting point had to be adjusted as it was not laying on the grid point of the coarser chain code. The self-intersecting parts appeared, too (the red arrows in Fig. 7c). When the last black nodes are generated (Fig. 7d), the chain code contains only Type_A self-intersections that are removed and, after this, the algorithm finishes.

SOLUTION FOR SELF-INTERSECTION OF TYPE_B

Function CANCELLATION removes pairs of the chain code symbols that negate themselves and, therefore, do not generate any quadtree nodes. This is definitely a step in the right direction, but the process of the resolution reduction may also produce another type of chain code self-intersection. Let us consider the geometric object shown in Fig. 8. The initial chain code is 22221121121121121121100033033033033001101101101100033233233233233233, where the starting position is denoted with the blue marker (see Fig. 8a). The algorithm produces the coarser chain code 2121121121030330301011010323323323, where the Chen and Chen approach does not detect any self-intersection. However, as seen in Fig. 8b four self-intersections actually exist, which were introduced in Section "Introduction" as Type_B. If the coarser chain code is processed further, the algorithm generates an incorrect quadtree. Unfortunately, the self-intersections of Type_B cannot be detected just by observing the chain code symbols. Indeed, the geometric approach should be applied, which is time consuming if applied in a naive way (according to Žalik *et al.* (2017) it works in $O(n^2)$ time, where n is the number of chain code symbols). Even worse, in this case it has to be done at all resolution levels.

In the continuation, a solution is proposed for handling self-intersections of Type_B. The approach is based on the space-filling curves (Sagan, 2012; Bader, 2012) similar to those proposed in Žalik *et al.* (2017). However, instead of the Hilbert curve used in Žalik *et al.* (2017), the Z-order curve (Sagan, 1994) is employed, as it is faster to compute. If two chain code symbols have the same index on the Z-order curve, a self-intersection appears (in Fig. 8c these self-intersections are plotted in violet). After that, the chain code is split into a sequence of non-self-intersecting chain codes, as shown in Fig. 8d. As illustrated, five independent objects are obtained which share four common points. The algorithm for determination of the self-intersection points (either of Type_A or Type_B) works in three steps:

1. The position z_i on the Z-order curve is determined for each chain code symbol. These positions are stored into an auxiliary array Z.
2. Array Z is sorted in an increasing order. If neighbouring elements with the same value (*i.e.*, $Z[i] = Z[i+1]$) exist in Z, the self-intersection is found and the chain code is split into two chain codes $\mathrm{CC}_{i,1}$ and $\mathrm{CC}_{i,2}$.
3. Both chain codes $\mathrm{CC}_{i,1}$ and $\mathrm{CC}_{i,2}$ are checked recursively for self-intersection.

THE ALGORITHM EXTENSION FOR HANDLING THE OBJECTS WITH HOLES

The Chen and Chen's algorithm was not designed to handle objects with holes. In the continuation, we present our extension of their algorithm to cope with the holes efficiently. The algorithm has two requirements: Firstly, the holes should be oriented differently (counterclockwise in our case) than the outer loop, and secondly, the holes and the outer loop should not intersect. The violation of these requirements should be resolved before applying this algorithm.

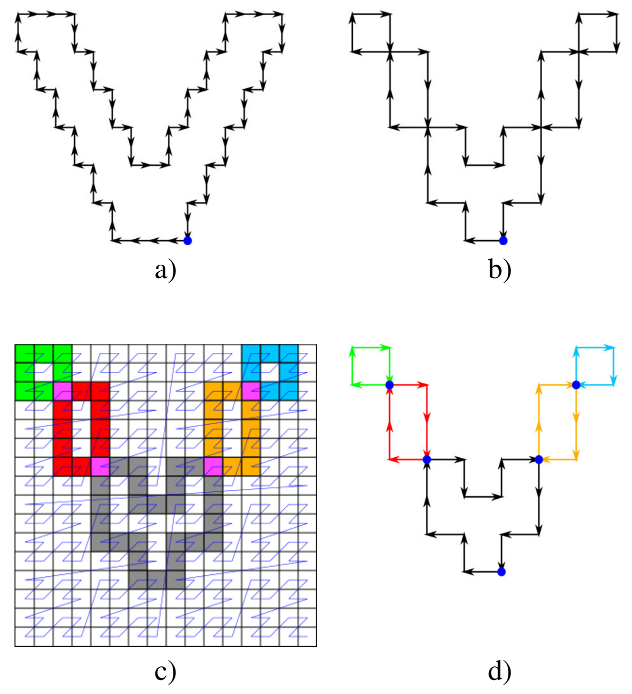

Figure 8. *The chain code (a) results in several Type_B self-intersections in its coarser representation (b); the Z-order curve is used to find self-intersection points (d); the self-intersected parts are split into five chain codes (d).*

The chain codes representing holes are processed firstly. In this way, the quadtree's black nodes are added at the end without the need to merge or delete already inserted nodes. For this, function GENERATEQTNODE is upgraded to determine which nodes should be inserted and which should be omitted as follows:

1. The white nodes being part of the hole, are inserted in the usual way.
2. The following cases are possible when inserting black node $n \in \mathrm{QN}_i$:
   (a) If node $m \in \mathrm{QIN}_n$ already exists and m is white, then n is not inserted;
   (b) If a white node m exists at the position where n is to be inserted, the insertion of m is omitted.
   (c) If there is a grey node m at the position where n should be inserted, all non-existing children of m are generated and set as black nodes.

An example is shown in Fig. 9. Firstly, the region representing the hole is inserted into the quadtree as a set of the white nodes (in Figs. 9c and 9b they are plotted in green). After that the chain code symbols representing the outer border are processed. Several black quadtree nodes are inserted directly (Fig. 9d). However, in the last step, the black node representing the left lower quadrant (framed with the red colour in Fig. 9e) should be inserted. Unfortunately, at this position, a grey node already exists (marked with the red colour in Fig. 9b). Such situation is considered as a conflict. Instead of inserting one black node representing the whole quadrant, only the missing nodes in the subtrees are inserted and marked as the black nodes. These nodes are plotted in violet in Figs. 9b and 9e.

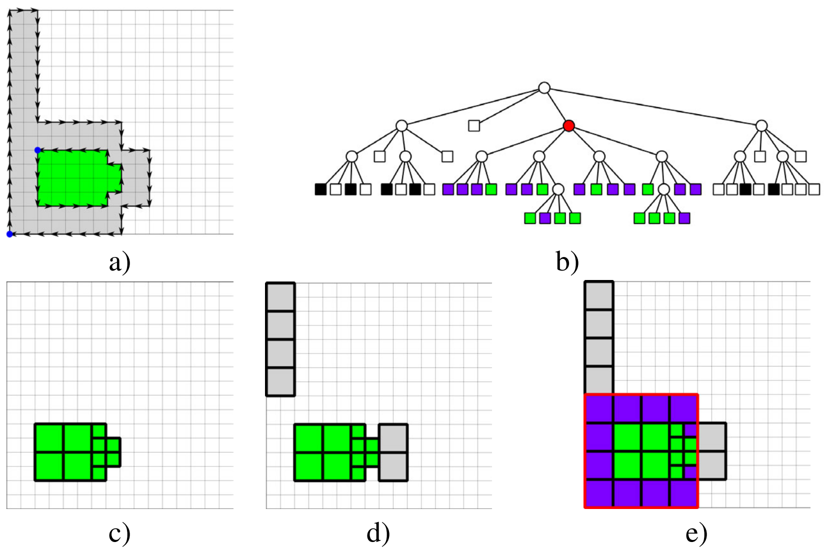

Figure 9. *Building quad tree (b) from chain codes describing letter 'b' (a). The quadrants describing the hole are inserted first (c), after that the chain code symbols describing the outside border are processed (d). The area where conflict appears is framed in red (e).*

RESULTS

The proposed improvements were tested extensively on a large number of rasterized objects represented by F4 chain codes. All tests were performed on a personal computer with an Intel i7-4790 3.36GHz processor and 8GB RAM. A representative set of used geometric objects is shown in Fig. 10, while the key information about them is reported in Table 1. All objects except Spider contain holes. The complexity of the shapes in these objects varies from very basic, like Smiley, to more complicated, like Dragon or Fiddlers. Several of the shapes have long sloped sections, like the legs in Spider, tongue in Dragon, or antennae in Butterfly. Other shapes have sharp turns like Skunk or Mockingbird.

An example of how the algorithm generates coarser F4 chain codes is displayed in Fig. 11. If no self-intersections occur when the resolution is reduced, the number of chain code symbols is approximately halved. In other cases, the number of chain code symbols on the coarser level of hierarchy is smaller (for example, from the case (e) to the case (f) and from the case (g) to the case (h) in Fig. 11).

Table 1. *Basic information about the chain codes in the experiment.*

Object	No. of F4 symbols	No. of holes	Width	Height
Ballet	16712	1	1647	2170
Buddha	11366	1	1997	2401
Butterfly	64448	126	2401	1651
Camel	20156	9	2401	1645
Cats	22238	5	2265	2401
Cupid	25160	4	2298	2287
Detective	14128	2	1581	2401
Dragon	26334	2	2319	2400
Fiddlers	27050	25	2401	1351
Frog	20646	4	2401	1934
Girl on bike	50752	52	2129	2289
Mockingbird	17800	4	2401	1966
Reindeer	19864	4	1902	1903
Rooster	32894	8	2083	2401
Skunk	23960	3	2419	2401
Smiley	16210	3	2401	2401
Spider	23900	0	2401	2358
Wolf	22058	1	2401	2401

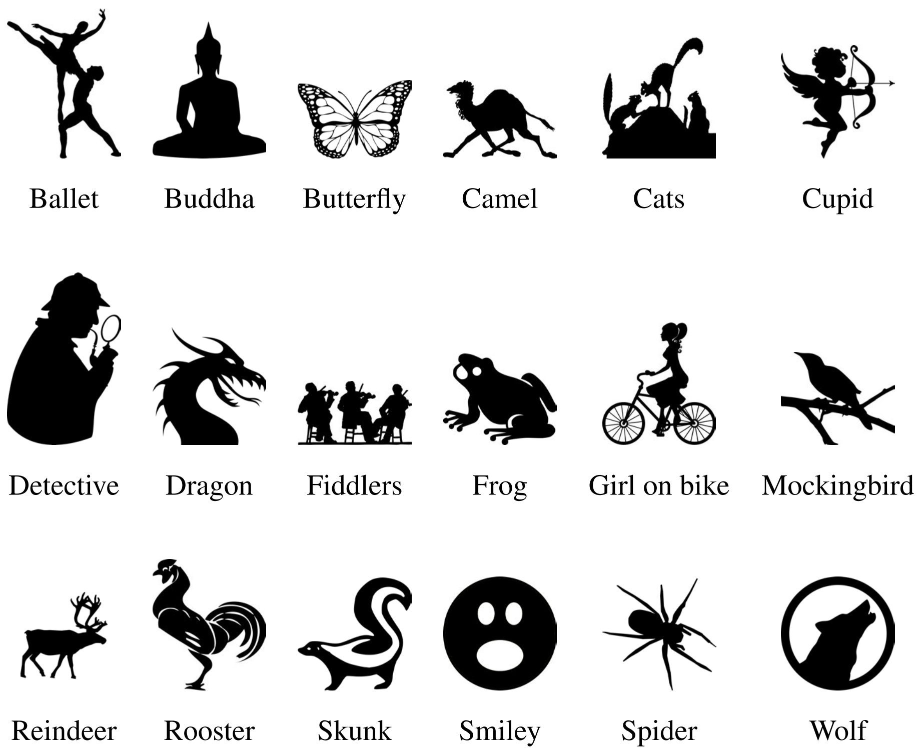

Figure 10. *A representative set of rasterized geometric objects.*

The number of Type_A and Type_B self-intersections that occur during the quadtree construction is given in Table 2.

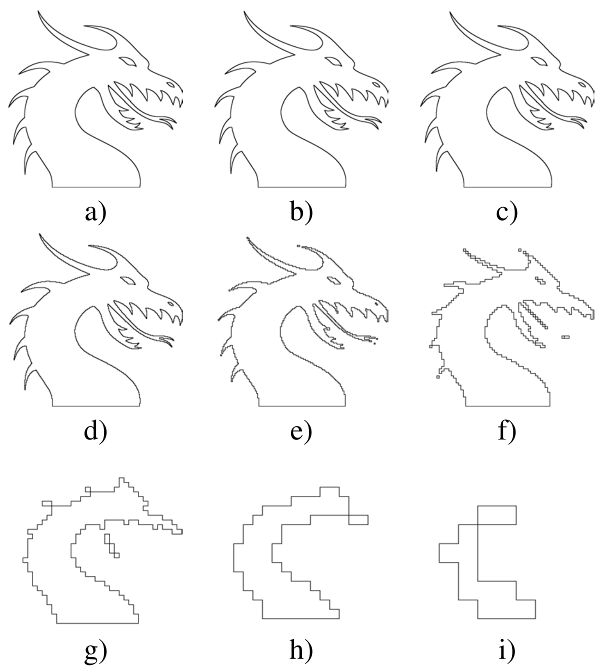

Figure 11. *Results of the algorithm, where the length of F4 symbols reduces as follows: 26336 (a), 13072 (b), 6436 (c), 3142 (d), 1456 (e), 582 (f), 198 (g), 72 (h), 26 (i).*

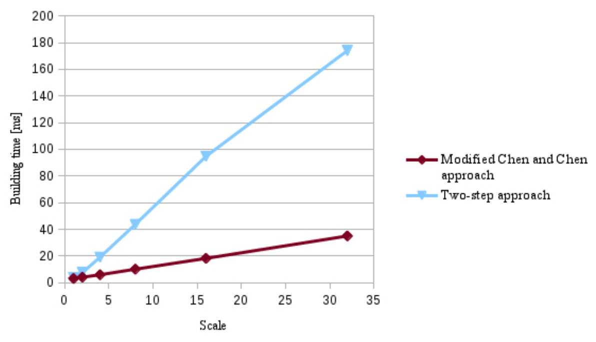

Figure 12. *Quadtree construction times for the object Girl on bike.*

Finally, the efficiency of the F4 to quadtree conversion algorithm is considered. As explained, Chen and Chen's approach did not consider the self-intersections of Type_B, and, therefore, the improved approach could not be compared with their approach. Since Chen and Chen compared their algorithm to the two-phase approach according to Samet (1980), we also used the Samet's approach as a reference. As this approach does not work with the objects containing holes, only outside borders were used. Each object was scaled by factors 1, 2, 4, and 8 in order to analyse the trend of execution times as the object's size increased. The results are shown in Table 3. A graphical comparison of spent CPU time for the object Girl on bike is given in Fig. 12.

Table 2. *Frequency of self-intersections.*

Chain code	Depth	Type_A	Type_B
Ballet	12	114	31
Buddha	12	36	0
Butterfly	12	695	89
Camel	12	126	18
Cats	12	93	11
Cupid	12	142	22
Detective	12	39	4
Dragon	12	170	54
Fiddlers	12	242	25
Frog	12	77	9
Girl on bike	12	646	117
Mockingbird	12	78	17
Reindeer	11	110	24
Rooster	12	305	77
Skunk	12	61	14
Smiley	12	6	0
Spider	12	153	59
Wolf	12	28	5

DISCUSSION

An improved solution for forming quadtrees from chain code representation of rasterized objects was presented in this paper. The original algorithm presented in Chen and Chen (2001) was extended by resolving the problem with a non-considered type of self-intersections that led to an incorrect quadtree. Such self-intersections cannot be detected by comparing the neighbouring symbols of the chain code, and require comparison of the pixels' coordinates. For the sake of efficiency, the space-filling Z-order curve was used for this purpose. Additionally, an extension of the method to handle the objects with holes was developed.

The experiments have shown that the self-intersections of Type_A are more frequent in general. However, although the self-intersections of Type_B are less frequent, they cannot be neglected for the safe execution of the algorithm, since they occur at least once in around 80–90% of test cases. It was determined that their occurrence is least likely with geometrically simple shapes (*e.g.*, circle, ellipse, square, etc.) as shown with the object "Smiley". The likelihood of occurrence increases with complexity of the shape and

Table 3. *Quadtree construction times [ms] for different scale factors of chain codes.*

Object	Modified Chen and Chen approach				Two-step approach			
	1	2	4	8	1	2	4	8
Ballet	3.093	3.966	5.871	9.541	3.332	7.162	17.181	42.588
Buddha	2.035	2.669	3.899	6.386	2.746	4.935	10.282	23.624
Butterfly	3.138	4.034	5.937	9.514	3.826	7.574	15.536	34.331
Camel	3.706	4.895	7.065	11.718	4.237	8.941	20.516	55.592
Cats	3.947	5.340	7.371	12.101	4.382	9.561	24.663	63.152
Cupid	4.305	5.703	8.277	13.570	4.915	10.537	24.929	74.018
Detective	2.181	2.895	4.220	6.896	2.499	5.755	11.227	26.814
Dragon	5.359	6.996	9.989	16.111	5.733	12.586	38.416	90.256
Fiddlers	3.387	4.457	6.696	10.980	4.123	8.421	22.143	57.883
Frog	3.551	4.612	6.751	11.113	4.042	9.323	20.154	56.229
Girl on bike	3.102	4.119	5.977	10.135	3.760	7.551	19.143	43.756
Mockingbird	3.061	4.003	5.822	9.255	3.705	7.688	16.537	44.963
Reindeer	3.457	4.609	6.685	11.070	3.967	8.451	18.955	53.784
Rooster	2.447	3.223	4.745	7.779	2.806	6.035	12.814	31.142
Skunk	2.658	3.482	5.148	8.165	3.044	6.502	14.683	36.235
Smiley	1.903	2.470	3.568	5.935	2.246	4.673	10.008	21.758
Spider	4.948	6.383	9.319	15.168	5.454	11.987	33.133	84.351
Wolf	1.910	2.477	3.621	6.094	2.344	5.065	10.560	22.558

they are almost unavoidable with objects that have long sloped sections (*e.g.*, tongue in the object "Dragon" as shown in Fig. 11).

While the shape of the geometric object should, intuitively, indicate whether Type_B self-intersection appears during the construction process, the experiments did not confirm this hypothesis. Actually, the shape of the considered geometric object is not the only factor where Type_B self-intersections could or could not appear. For example, object Buddha, which shape can be considered as geometrically demanding, does not contain any Type_B self-intersections during the quadtree building process according to the results from Table 2. However, Type_B self-intersections appeared when Buddha was translated by a few pixels. This feature was also confirmed also with other objects where the number of Type_B self-intersections depends on the objects' positions. As a rule, however, Type_B self-intersection occurs in objects with long slanted sections like legs on the objects Ballet, Camel, Reindeer, or antennae on the Butterfly. Other good indicators about the possibility that Type_B self-intersection may occur are also concave shapes like on the relatively simple object Wolf. Except for the basic convex objects (*i.e.*, circles, ellipses, rectangles), there is no guarantee that at some position Type_B self-intersections cannot occur at some position.

Despite the added step of searching for self-intersections on each level of quadtree construction, this solution is still considerably more efficient than the two-step approach. While the experiments did show that a large number of self-intersections can adversely affect the runtime, with large objects this effect diminishes and the efficiency of our approach is evident as shown in Fig. 12. Table 3 shows that, for all other objects, the trends are similar.

Our implementation of the algorithm, together with testing F4 chain codes, can be downloaded from https://gemma.feri.um.si/chain-code-to-quadtree-conversion/.

ACKNOWLEDGEMENTS

This work was supported by the Slovenian Research Agency under Grants J2-8176 and P2-0041. In addition, this work was financed in part by the European Union from the European Fund for Regional Development and the Ministry of Education, Science and Sport of the Republic of Slovenia under the Grant C3330-17-529005.

REFERENCES

Anand S, Knott K (1991). An algorithm for converting the boundary representation of a CAD model to its octree representation. Comput Ind Eng 21:343–7.

Bader M (2012). Space-filling curves: an introduction with applications in scientific computing. Berlin: Springer.

Bribiesca E (1999). A new chain code. Pattern Recogn 32:235–51.

Bribiesca E, Bribiesca-Contreras F, Ángel Carrillo-Bermejo, Bribiesca-Correa G, Hevia-Montiel N (2019). A chain code for representing high definition contour shapes. J Vis Commun Image R 61:93–104.

Bribiesca E, Bribiesca-Contreras G (2014). 2d tree object representation via the slope chain code. Pattern Recogn 47:3242–53.

Chen Z, Chen IP (2001). A simple recursive method for converting a chain code into a quadtree with a lookup table. Image Vision Comput 19:413–26.

Finkel RA, Bentley JL (1974). Quad trees: a data structure for retrieval on composite keys. Acta Inform 4:1–9.

Freeman H (1961). On the encoding of arbitrary geometric configurations. IRE T Electr Comput EC-10:260–8.

Hoffmann CM (1989). Geometric and solid modeling: An introduction. San Francisco: Morgan Kaufmann.

Krishnan R, Das A, Gurumoorthy B (1996). Octree encoding of *B*-rep based objects. Comput Graph 20:107–14.

Lattanzi M, Shaffer C (1991). An optimal boundary to quadtree conversion algorithm. CVGIP Imag Understan 53:303–12.

Lemus E, Bribiesca E, Garduño E (2014). Representation of enclosing surfaces from simple voxelized objects by means of a chain code. Pattern Recogn 47:1721–30.

Mäntylä M (1987). An introduction to solid modeling. New York: Computer Science Press.

Mark D, Abel D (1985). Linear quadtrees from vector representations of polygons. IEEE T Pattern Anal 7:344–9.

Mortensen ME (1985). Geometric modeling. New York: John Wiley & Sons.

Sagan H (1994). Lebesgue's space-filling curve. New York: Springer, 69–83.

Sagan H (2012). Space-filling curves. New York: Springer.

Samet H (1980). Region representation: quadtree from chain codes. Commun ACM 23:163–70.

Shih FY, Wong WT (2001). An adaptive algorithm for conversion from quadtree to chain codes. Pattern Recogn 34:631–9.

Sánchez-Cruz H, López-Valdez HH, Cuevas FJ (2014). A new relative chain code in 3D. Pattern Recogn 47:769–88.

Sánchez-Cruz H, Rodríguez-Dagnino RM (2005). Compressing bilevel images by means of a three-bit chain code. Opt Eng 44:1–8.

Webber R (1984). Analysis of quadtree algorithms. PhD Thesis. College Park, MD: University of Maryland.

Žalik B, Mongus D, Žalik KR, Lukač N (2016). Chain code compression using string transformation techniques. Digit Signal Process 53:1–10.

Žalik B, Mongus D, Liu YK, Lukač N (2016). Unsigned Manhattan chain code. J Vis Commun Image R 38:186–94.

Žalik B, Mongus D, Lukač N, Žalik KR (2018). Efficient chain code compression with interpolative coding. Inform Sciences 439/440:39–49.

Žalik B, Mongus D, Žalik KR, Lukač N (2017). Boolean operations on rasterized shapes represented by chain codes using space filling curves. J Vis Commun Image R 49:420–32.