PEL: A Predictive Edge Linking Algorithm

Cuneyt Akinlar, Edward Chome

Anadolu University, Computer Engineering Department, Eskisehir, TURKEY

{*cakinlar,edwardchome*}*@anadolu.edu.tr*

Abstract

We propose an edge linking algorithm that takes as input a binary edge map generated by a traditional edge detection algorithm and converts it to a set of edge segments; filling in one pixel gaps in the edge map, cleaning up noisy edge pixel formations and thinning multi-pixel wide edge segments in the process. The proposed edge linking algorithm walks over the edge map based on the predictions generated from its past movements; thus the name Predictive Edge Linking (PEL). We evaluate the performance of PEL both qualitatively using visual experiments and quantitatively within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS 300). Both visual experiments and quantitative evaluation results show that PEL greatly improves the modal quality of binary edge maps produced by traditional edge detectors, and takes a very small amount of time to execute making it suitable for real-time image processing and computer vision applications.

Keywords:
Edge Detection, Edge Linking, Edge Segment Detection, CannySR, Edge Drawing (ED)

1. Introduction

Edge detection is a very important and fundamental first step in many computer vision and image processing applications. A traditional edge detection algorithm [1, 2, 3, 4] takes a grayscale image as input and produces a binary edge map (BEM) as output, where an edge pixel (edgel) is marked (e.g., its value in the edge map is 255), and a non-edge pixel is unmarked (e.g., its value in the edge map is 0).

The binary edge maps produced by traditional edge detectors are usually of low quality, consisting of gaps between the edgels, unattended edgels and noisy notch-like structures, ragged and multi-pixel wide edgels formations etc. An example of such an edge map with low quality artifacts is shown in Fig. 1 for the famous Lena image. This edge map was obtained by the OpenCV implementation of the widely-used Canny [2] edge detector (cvCanny) [5], which is the fastest known Canny implementation. To obtain this edge map, the input image was first smoothed by a Gaussian kernel with $\sigma = 1.5$ (using cvSmooth from OpenCV), and cvCanny was called with low and high threshold values set to 20 and 40 respectively, and the Sobel kernel aperture size set to 3. Fig. 1 also shows the close-up views of two separate sections of the edge map to illustrate the low quality artifacts, which can be grouped in three categories as follows: (1) There are discontinuities and gaps between edgel groups as can clearly be seen in the close-up views of the two enlarged sections of the edge map. Some of these gaps need to be filled up. (2) There are noisy, unattended edgel formations and notch-like structures. This is more evident in the close-up view of the upper-left corner of the edge map. These noisy artifacts needs to be removed. (3) There are multi-pixel wide edgel formations in a staircase pattern especially around the diagonal edgel formations (both 45 degree and 135 degree diagonals). Such formations can be seen in many places in the edge map, and they need to be thinned down to 1-pixel wide chains.

To improve the modal quality of the binary edge maps produced by traditional edge detectors, edge linking methods have been proposed in the literature [7]-[24]. The goals of these methods are commonly to remove noisy edgel formations and clean up the edge map, and to fill in gaps between edgels to form longer edgel groups.

Snyder et al. [7] is one of the first researchers to present a method to deal with errors in edge detector results. The authors propose a method to close gaps in edge maps while preserving edge structure and connectedness based on the concept of a chamfer map. Xie [6] presents a method to link edge pixels for the purpose of line segment detection. The method makes use of the concepts of horizontal edge element and causal neighborhood window to realize edge linking and consequently line segment detection. Given a binary edge map, the author performs linking of pixels aligned on the same image line into what is called a horizontal edge element, which is then converted to a line segment. The algorithm performs poorly especially in highly textured regions. Basak et al. [8] also consider the problem of line detection via edge linking and propose two neural network architectures.

Farag and Delp [9] define edge linking as a graph search problem. Beginning at a start node, they use the A^* search algorithm to reach a goal node. The search makes use of the gradient magnitude and angles, and the swath of edge information estimated by the zero crossings of Laplacian of Gaussian operator. They propose a linear path metric function to guide the

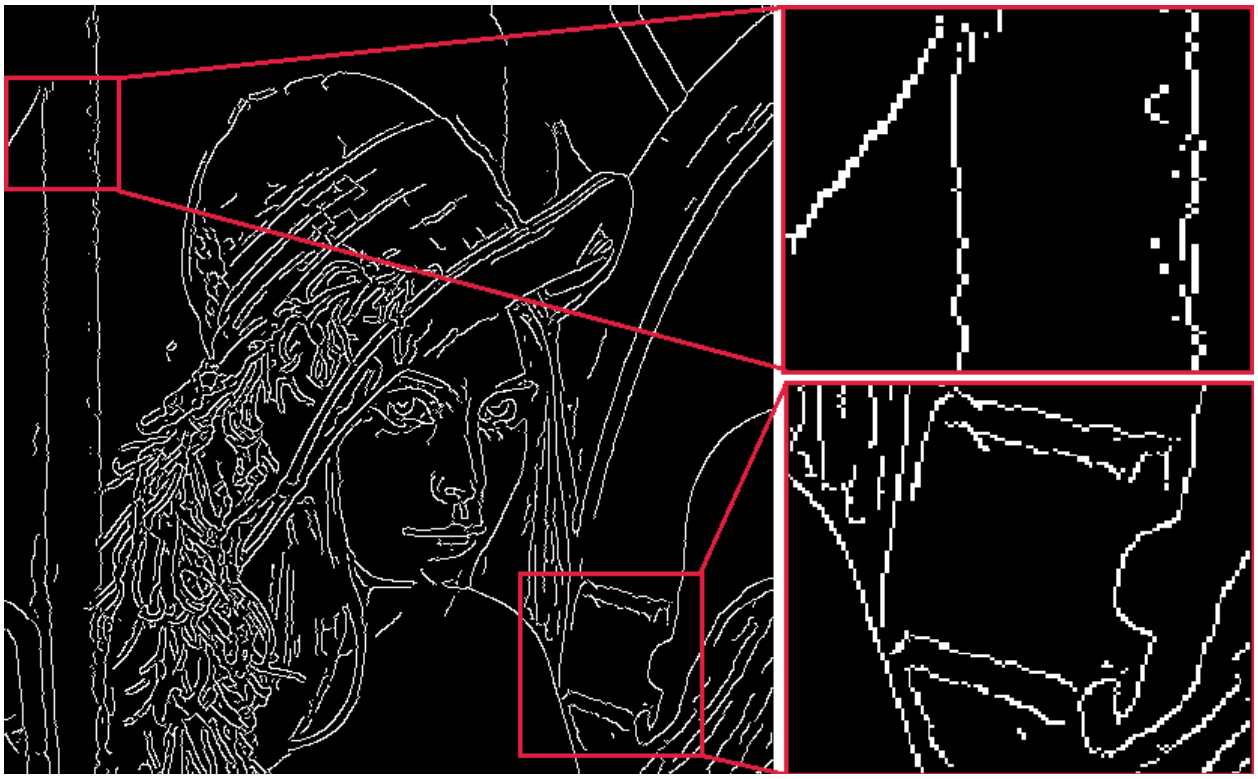

Figure 1: Left: Lenas edge map result by OpenCV Canny with low and high threshold values of 20 and 40 respectively. The image was first smoothed by a Gaussian kernel with $\sigma = 1.5$. Right: Close-up view of two sections of the edge map showing low quality edge group formations: Discontinuities and missing edgels, noisy, notch-like edge group formations, and multi-pixel wide staircase edgel structures.

A^* search. To limit the search space, the algorithm only looks at three neighbors for transitioning, which reduces the running time but affects the accuracy of the algorithm. The biggest problem with this algorithm is the large number of parameters that needs to be adjusted. Also some statistical parameters and a priori information about the edge map needs to be known for the algorithm to work correctly. Zhu et al. [10] present an algorithm to link discontinuous edge segments. They model an edge map as a global potential field with energy dispositions at detected edges. A directional potential function measures the energy charges and guides the linking process to fill the gaps between the broken edge fragments. Their method can fill small gaps between broken edge fragments successfully, but produces unacceptable results when the gaps are large or the image is noisy.

Saber et al. [11] propose a method for segmentation of the color images. They model the segmentation map as a Gibbs random field and use the gradient magnitude of the 3-channel color image to find the regions. The boundaries of the closed regions constitute the final linked edge map. Maeda et al. [13] also concentrate on image segmentation and propose incorporating a linking algorithm based on a directional potential function into an edge preserving smoothing filter. They show that the unnecessary details of the image are smoothed out before region growing is performed. Hajjar and Chen [12] propose an edge linking algorithm and its VLSI implementation for real-time edge linking. Their method is based on the break points' direction and weak level points and try to fill the gaps between edge groups. Ghita and Whelan [14] also propose an algorithm to close the gaps in an edge image by local analysis of the edge break points (terminators). They mark the edge termination points, and determine the connection path between different edge termination points by an analysis of the local edge structure. Their algorithm requires a single pass through the edge map and can be applied without a priori information. Shih and Cheng [15] apply mathematical morphology to edge linking to fill in the gaps between edge segments. Broken edge segments are extended along their slope directions by using the adaptive dilation operation with suitable elliptical structuring elements. They also apply thinning and pruning as a post-processing step. Although their algorithm is shown to fill gaps in the edge map of an ellipse, it is not clear how it can be applied without a priori information about the edge map.

Sappa and Vintimilla [18] present an algorithm that takes in an edge map and the original intensity image, and generates closed contours using graph theoretical approaches. In the process, they close small gaps and remove spurious edge pixels, but their algorithm is more tailored towards image segmentation than edge linking. Lu and Chen [16] apply ant colony optimization to compensate broken edges. They propose four moving policies for ants and apply a finite number of iterations to fill up the broken edge segments. Jevtic et al. [19] propose a similar ant based edge linking algorithm to combine broken edges. Wang and Zhang [17] propose the use of application-specific lo-

cal neighborhood to compute edge direction and geodesic distance for measuring proximity between candidate edge points to be linked. Lin et al. [20] make use of an edge linking algorithm with directional edge-gap closing to produce complete edge-links that are then used for lane detection.

Flores et al. [21] present a method for the segmentation of intensity images that combines an optical contouring technique with edge linking. Ji et al. [22] present a method for segmentation of satellite imagery. Their idea is to make use of the gradient magnitude and directions and perform a heuristic A^* search to link the original edge points. Their algorithm is also tailored toward image segmentation. Guan et al. [23] recently proposed a Partial Differential Equation (PDE)-based method to link the edges to obtain closed contours for image segmentation. They then show their algorithm's performance on the segmentation of cell images.

We can classify most of the edge linking algorithms found in the literature in two categories: (1) Those that try to fill the gaps between broken edge groups in the edge map, which is only part of the edge linking problem that we attempt in this paper, (2) those that use edge linking to find closed contours for image segmentation, which is outside the scope of this paper.

To overcome the problems associated with binary edge maps from the get-go, authors in [25] propose a completely new edge and segment detection algorithm called Edge Drawing (ED). Rather than detecting edgels individually, which causes the above-mentioned problems in the first place, the authors conceive edge detection as a boundary detection problem and model it similar to childrens dot completion games, where the child is given a set of marked anchor points (numbered dots) in a picture and is asked to connect the dots to reveal the hidden picture boundaries. To emulate a dot completion game, ED first computes a set of anchor points, which are pixels that are definitely assumed to be edgels, and then uses an efficient method called the Smart Routing (SR) to join these anchors. Since ED draws the boundaries by attaching contiguous pixels one after the other, it outputs not just a binary edge map, but a set of edge segments, each of which is a contiguous chain of pixels.

Recently, Akinlar and Chome [24] proposed an edge linking algorithm named CannySR that computes a binary edge map using the Canny edge detection algorithm and then uses the Smart Routing step of Edge Drawing to convert Canny's binary edge map to a set of edge segments. CannySR is shown to produce good results, but it is limited to working only in the context of the Canny edge detector. That is, for CannySR to work correctly, it requires the original grayscale image and the Gaussian smoothing kernel used to smooth the image before Canny edge detection. This is not a problem when the original image is available and we know how it was smoothed before edge detection, but in the general case where we only have the binary edge map and we do not know how it was obtained, CannySR does not work. Ideally, one wants to have an edge linking algorithm that requires only the binary edge map to work without needing to know how it was obtained.

In this paper we propose an edge linking algorithm that has the following goals:

(1) Close small gaps (1-pixel gaps in our case) between edgel groups

(2) Clean-up noisy, unattended and notch-like structures from the edge map

(3) Thin down multi-pixel wide staircase edgel formations to 1-pixel wide chains

(4) Convert the binary edge map into a set of edge segments, each of which is a clean, contiguous, 1-pixel wide chain of pixels.

Traditional edge linking algorithm proposed in the literature mostly restrict their attention to the first goal with some emphesis on the second and third goals without considering the last goal, which we believe is very important. This is due to the fact that having a set of edge segments rather than a binary edge map enables us to perform many higher-level processing jobs such as line [26], arc, circle, ellipse [27], polygon and general shape detection, image registration and segmentation, edge segment validation [28] etc. among many others.

In this paper we propose an edge linking algorithm that satisfies all four goals listed above. The proposed algorithm just takes in a binary edge map generated by any traditional edge detection algorithm and converts it to a set of edge segments; filling in one pixel gaps in the edge map, cleaning up noisy edgel groups and thinning multi-pixel wide edgel formations in the process. It walks over the edge map based on the predictions generated from its past movements; thus the name Predictive Edge Linking (PEL) [37]. We give the details of PEL in section 2, and evaluate its performance both qualitatively using visual experiments and quantitatively within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS 300) [29, 30] in section 4. Section 5 concludes the paper.

2. Predictive Edge Linking (PEL)

In this section we describe the details of our Predictive Edge Linking (PEL) algorithm, which takes as input only the binary edge map (BEM) produced by a traditional edge detector and returns a set of edge segments; filling in one pixel gaps in the edge map, cleaning up noisy edgel groups and thinning multi-pixel wide edgel formations in the process.

The pseudocode for PEL is given in algorithm 1. PEL only takes a BEM as input and returns a set of edge segments (ES) as output. The algorithm consists of 4 steps: In the first step, one pixel gaps in BEM are filled. In the second step, the edge segments are created. In the third step, edge segments whose endpoints are close to each other are joined together to form longer edge segments. In the fourth and the last step, the multi-pixel wide edge segments are thinned down to one-pixel wide edge segments. The minimum segment length supplied by the user specifies the length of the shortest segment to be returned by PEL. Any segment shorter than the minimum segment length are removed and not returned to the user. In the following, we explain each step in detail.

Algorithm 1 Predictive Edge Linking (PEL)

Symbols used in the algorithm:
BEM: Binary Edge Map
ES: Edge Segments

PEL(BEM, MIN_SEGMENT_LEN)
I. FillGaps(BEM);
II. ES = CreateSegments(BEM);
III. JoinSegments(ES);
IV. ThinSegments(ES, MIN_SEGMENT_LEN);
V. return ES;

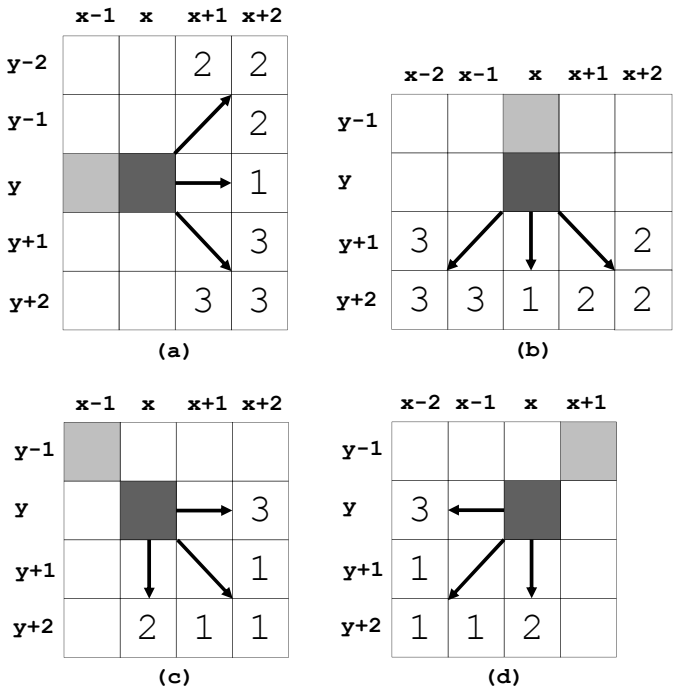

Figure 2: Filling one pixel gaps between the edgel groups. The dark gray pixel at (x, y) is the tip pixel of an edgel group, and the light gray pixel in each case is its neighbor. We perform a check along the direction where the tip is moving, and connect the tip to a neighboring edgel by filling the appropriate pixel.

2.1. FillGaps: Filling One Pixel Gaps in the Binary Edge Map

Recall from Fig. 1 that edge maps produced by traditional edge detectors contain gaps between edgel groups. Many proposals for edge linking found in the literature concentrate on this problem, and propose solutions to fill these gaps. Although our concentration in this paper is on obtaining edge segments, filling the gaps is also important. In this section, we propose a heuristic method to fill one pixel gaps between edgel groups.

Fig. 2 illustrates our heuristic for filling one pixel gaps between the edgel groups. Our idea is to first find the tips of the edgel groups, and then connect each tip to a neighbouring edgel that is one pixel away. The tip an edgel group is defined to be a pixel with only one neighbor. Of the eight possible scenarios, Fig. 2 depicts four cases, where the tip pixel, marked in dark gray, is located at (x, y), and its only neighbor is marked in light gray in each case. The other four cases (left, up, up-left, up-right) are simply symmetric versions of these four cases.

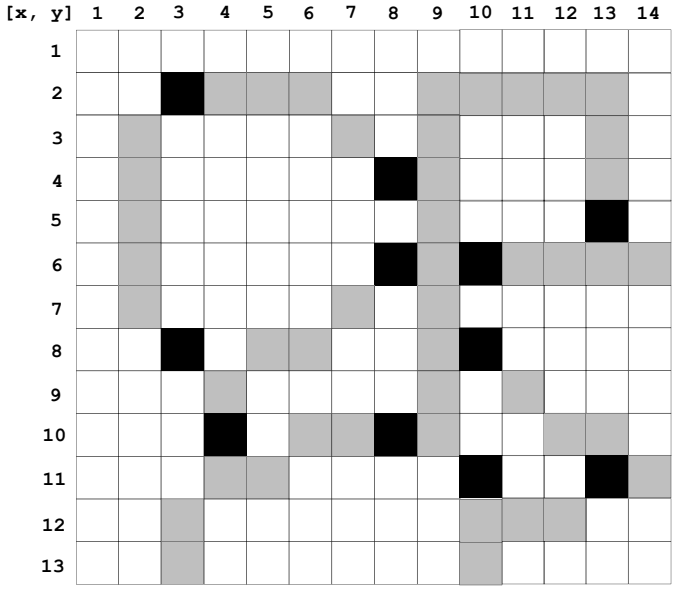

Figure 3: How the proposed heuristic fills up one pixel gaps in the edge map. Light gray pixels are the original edgels in the input BEM, and the dark pixels are the filled pixels.

Our heuristic for connecting a tip pixel takes into account the direction the tip pixel is moving towards and connects it to a neighboring edgel group in that direction. For example, consider the case in Fig. 2(a), where the tip pixel is moving to the right because its neighbor, marked in light gray, is located on the left at (x-1, y). Here we perform three checks to the right marked as 1, 2 or 3 in the figure. If the pixel at (x+2, y) is an edgel, then we connect the tip pixel to (x+2, y) by filling the pixel at (x+1, y). Otherwise, we check if one of the pixels at (x+1, y-2), (x+2, y-2) or (x+2, y-1) is an edgel, and if yes, then we connect the tip to one of these pixels by filling the pixel at (x+1, y-1). Finally, we check if one of the pixels at (x+1, y+2), (x+2, y+2) or (x+2, y+1) is an edgel, and if yes, then we connect the tip to one of these pixels by filling the pixel at (x+1, y+1). Although the other seven directions are not elaborated, we follow a similar procedure for each direction, and connect the tip pixel to a neighboring edgel in the direction that the tip is moving.

Fig. 3 illustrates how the proposed heuristic fills up one pixel gaps in the edge map. In the figure, light gray pixels are the original edgels in the input BEM, and the dark ones are the filled pixels. Just to explain why the pixel (8, 10) is filled, consider the tip pixel at (7, 10), which is moving to the right because its only neighbor is located at (6, 10). According to Fig. 2(a), we need to first check the pixel at (9, 10), which is an edgel. Therefore, (7, 10) is connected to (9, 10) by filling (8, 10).

It is important to note that, instead of the heuristic described in this section, filling the gaps in the input BEM can be done by any other method proposed in the literature. Edge segment creation method presented in section 2.2, which is the heart of the solution proposed in this paper, just uses the filled in BEM to create a set of edge segments, and does not care how the input BEM was filled in.

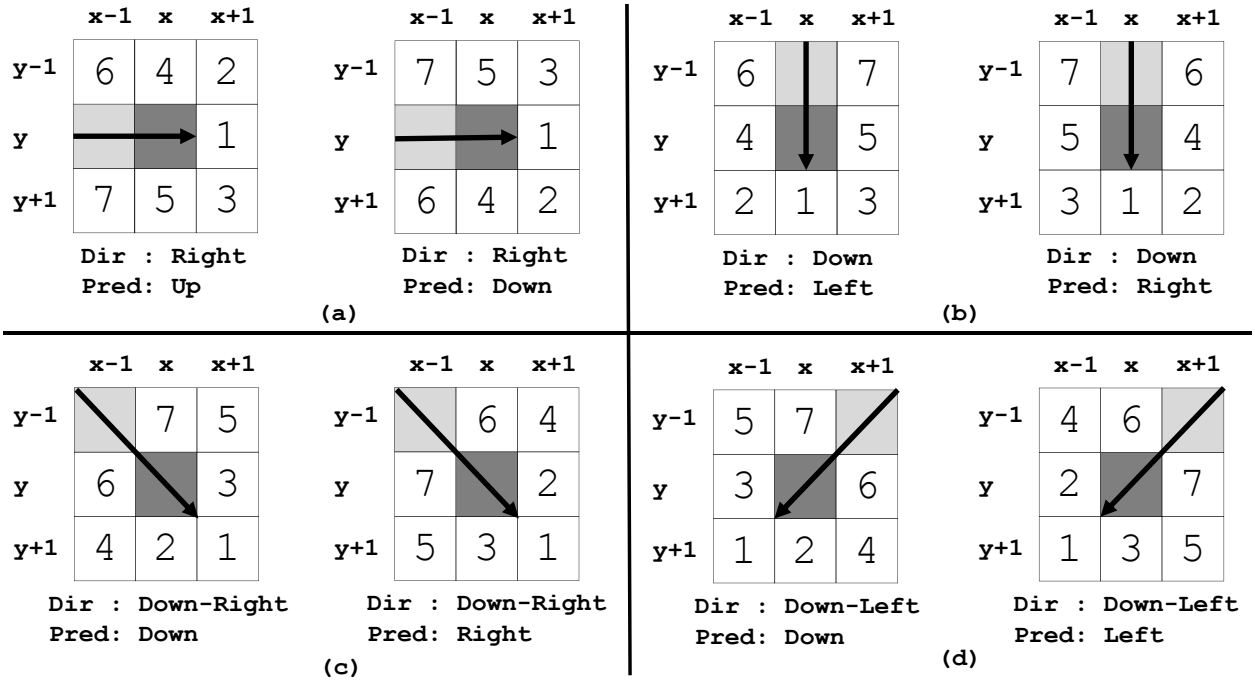

Figure 4: Walking in four directions with prediction. We are currently at pixel (x, y), marked with dark gray color, and moving towards (a) Right, (b) Down, (c) Down-Right, (d) Down-Left. The other four directions, i.e., Left, Up, Up-Left, Up-Right, are simply symmetrical versions of these four directions respectively.

2.2. CreateSegments: Linking Contiguous Edgels to Create Pixel Chains

Now that some of the missing pixels in BEM have been filled up, we move on to the heart of the problem: that of linking the edgels in BEM and creating the edge segments, each a contiguous chain of pixels. The heuristic that we employ at this step is to start at an arbitrary edgel in BEM and create potentially two chains starting at that edgel: One in the forward direction and one in the reverse direction. We then combine these two chains together to create a single chain of pixels, which essentially makes up for one edge segment. We then start at another edgel and do the same thing until all edgels in BEM are converted to edge segments.

To create an edge segment, we perform eight-directional walk with prediction, therefore the name Predictive Edge Linking (PEL). The prediction is used when the current direction changes. By taking the last eight directions into account, the prediction engine tells us which direction to move on to after the current direction changes.

Fig. 4 illustrates the walk in four directions: (a) Right, (b) Down, (c) Down-Right and (d) Down-Left. The other four directions, i.e., Left, Up, Up-Left and Up-Right are simply symmetrical versions of these directions respectively and are not illustrated.

Starting with Fig. 4(a), we see a walk to the right. That is, we are currently at pixel (x, y), marked with dark gray color, and we moved here from pixel (x-1, y), marked with light gray color. Since the current direction is "right", we immediately check the pixel to the right, i.e., (x+1, y), regardless of our past moves. If there is an edgel at (x+1, y), then we add it to the current chain and move there. The current walk direction continues to be "right". If there is no edgel at (x+1, y), then we need to change the current direction and check the other six neighbors in some order. This is where the prediction comes into play. We can first check the "up-right" pixel at (x+1, y-1) or the "down-right" pixel at (x+1, y+1). To make this decision, we consult the prediction engine, which, taking the last eight moves into account, tells us to either check the "up-right" or the "down-right" pixel first. If the prediction is "up", then we first check the "up-right" pixel at (x+1, y-1) and move there if there is an edgel. The current direction then changes from "right" to "up-right". If the prediction is "down", then we first check the "down-right" pixel at (x+1, y+1) and move there if there is an edgel. The current direction then changes from "right" to "down-right". Fig. 4(a) shows in detail the order in which the neighbors of the current pixel (x, y) are checked depending on the current prediction. While checking the neighbors in the depicted order, as soon as we encounter an edgel we move there and change the current direction accordingly. For example, if the current prediction is "up", and there is no edgel at pixels marked 1, 2, 3, 4 but there is an edgel at pixel marked 5, then we move to (x, y+1) and change the current direction as "down". Then in the next iteration of the loop, we will check the neighbors of the current pixel using the order shown in Fig. 4(b). The current pixel chain will come to an end when we check all 6 neighbors of the current pixel (x, y) and none has an edgel.

Although walking in all eight directions follow a logic similar to the one described in the previous paragraph, diagonal moves need a little more explanation. Consider the "down-right" walk depicted in Fig. 4(c). The first neighbor to be checked in the "down-right" pixel at (x+1, y+1) regardless of the prediction. If there is an edgel there, then we will move to

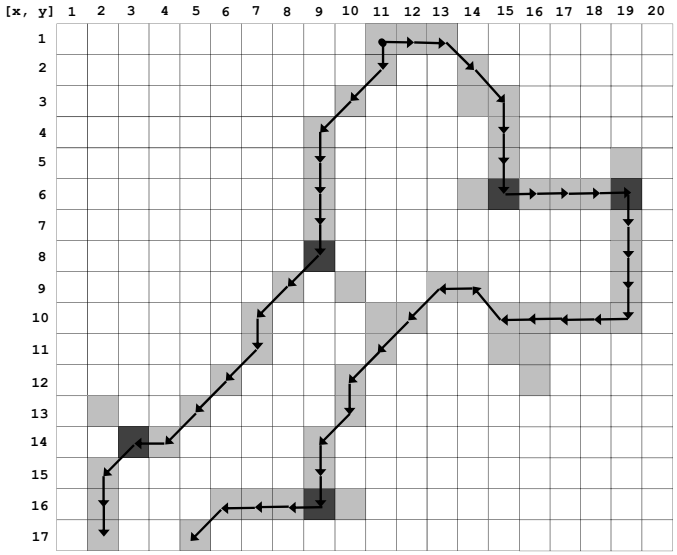

Figure 5: An example illustrating one edge segment creation starting at pixel (11, 2). Two chains are created, which are then combined together to create one edge segment. Colored pixels have edgels. Dark gray pixels are places where the prediction guides us in the correct direction.

that pixel and the current walk direction continues to be "down-right". In this case however, there is a little caveat that needs to be taken into account as follows. As we move to the "down-right" pixel marked as 1 in Fig. 4(c), most of the time we would also have an edgel at pixels marked 2 or 3 because the traditional edge detectors usually generate staircase edgel structures for diagonal edge groups. This can clearly be observed in Canny's Lena edge map shown in Fig. 1. Since we need to collect all pixels in the input BEM during edge segment generation, as we make the "down-right" move in Fig. 4(c), we also check the pixels to the right and down, i.e., pixels (x+1, y) and (x, y+1), and if one of them contains an edgel, we also pick it up and add it to the current chain. If both neighbors have an edgel, only one if picked depending on the current prediction. It is important to note that this approach applies not only to the "down-right" move, but to all four diagonal moves. To be more specific, if we are making a "down-left" move and there is an edgel at the "down-left" pixel (x-1, y+1) marked as 1 in Fig. 4(d), then we check the neighbors (x, y+1) and (x-1, y). If any one of these contain an edgel, then it is picked up and added to the current chain before we move "down-left" to (x-1, y+1).

Fig. 5 illustrates the creation of an edge segment. Assume that we start the segment creation at pixel (11, 1). Looking at this pixel's neighbors, there are two possible walks: One going "down" through (11, 2), one going "right" through (12, 1). The arrows depict how PEL walks over the edgels (denoted with gray color) and obtain two chains. At each pixel, the determination of the next pixel to move on to is made by the moves shown in Fig. 4. The dark gray pixels in Fig. 5 are where the prediction guides the walk one way rather than the other, and illustrates the importance of using the prediction to obtain longer chains.

Table 1: How PEL creates the chain going down from (11, 1)

Current Pixel	Direction	Comment
(11, 1)	Down	Check (11, 2): Full
(11, 2)	Down	Check (11, 3): Empty. Check (10, 3): Full
(10, 3)	Down-Left	Check (9, 4): Full
(9, 4)	Down-Left	Check (8, 5): Empty. Pred: Down. Check (9, 5): Full
(9, 5)	Down	Check (9, 6): Full
(9, 6)	Down	Check (9, 7): Full
(9, 7)	Down	Check (9, 8): Full
(9, 8)	Down	Check (9, 9): Empty. Pred: Left. Check (8, 9): Full
(8, 9)	Down-Left	Check (7, 10): Full
(7, 10)	Down-Left	Check (6, 11): Empty. Pred: Down. Check (7, 11): Full
(7, 11)	Down	Check (7, 12): Empty. Pred: Left. Check (6, 12): Full
(6, 12)	Down-Left	Check (5, 13): Full
(5, 13)	Down-Left	Check (4, 14): Full
(4, 14)	Down-Left	Check (3, 15): Empty. Pred: Down. Check (4, 15): Empty. Check (3, 14): Full
(3, 14)	Left	Check (2, 14): Empty. Pred: Down. Check (2,15): Full
(2, 15)	Down	Check (2, 16): Full
(2, 16)	Down	Check (2, 17): Full
(2, 17)	Down	End of chain

To better understand why PEL makes the moves depicted in Fig. 5, Table 1 gives the details of the decision engine as PEL creates the chain going down from (11, 1). During this chain creation, the prediction engine helps PEL make the correct decision at two locations as follows: At pixel (9, 8), PEL is walking down and the pixel downstairs, i.e., (9, 9), is empty. At this point there are two alternatives: PEL can walk down-left to (8, 9) or down-right to (10, 9). PEL asks the prediction engine to guide it to the left or to the right. The prediction engine performs an analysis of the last 8 moves (in fact there are only 7 moves to this point, i.e., down, down-left, down-left, down, down, down, down) and concludes that PEL should check the down-left pixel before the down-right pixel and guides PEL to the down-left pixel at (9, 9). Similarly, at pixel (3, 14) PEL is moving left and the pixel to the left, i.e., (2, 14), is empty. At this point there are two alternatives: PEL can walk down-left to (2, 15) or up-right to (2, 13). Again PEL consults the prediction engine, which recommends PEL to check the down-left pixel before the up-right pixel because over the last 8 moves, PEL made 2 "down" and 5 "down-left" moves but no "up" moves. Although not given in Table 1, a similar analysis can be performed for the chain starting at (11, 1) and going right from (12, 1) as follows. At pixel (15, 6), the prediction engine guides PEL to the right rather than to the left; at (19, 6) PEL is guided down rather than up, and at (9, 16) PEL is guided to the left

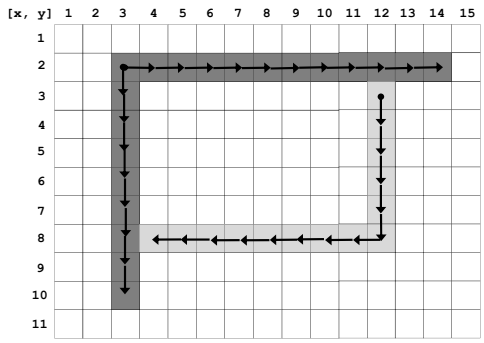

Figure 6: An example illustrating PEL creating two edge segments that should be joined together into one edge segment.

rather than to the right.

It is also important to note that after the two chains shown in Fig. 5 are obtained, they are joined together into one chain by taking the pixels from one chain in the forward direction and the pixels from the other chain in the backwards direction. This new chain makes up for the actual edge segment to be returned by PEL.

2.3. JoinSegments: Extending Nearby Edge Segments

Although PEL uses a prediction engine in an attempt to make the correct decisions during a walk and obtain as long edge segments as possible, the structure of the input BEM may lead to two or more edge segments during segment creation that in fact should have been combined together into one edge segment.

To understand the problem better, consider the illustration in Fig. 6. When this BEM is fed into PEL, two edge segments are created: one marked with dark gray color and the other marked with light gray color. We can imagine however that this BEM belongs to the boundary a rectangular object and a single edge segment that traces the entire boundary of the rectangle would be better to return rather than two edge segments as PELs segment creation algorithm would do. Therefore, after segment creation we have a new simple step named JoinSegments that groups neighbor edge segments and joins them together. We define that two edge segments are neighbors if the end point of one segment touches the other segment at some point and their end points are at most five pixels away from each other. Using this definition, the two edge segments in Fig. 6 are neighbors and can be combined together. To combine the two edge segments in Fig. 6, we first cut the superfluous pixels from the first segment; that is, pixels (13, 2), (14, 2) from one end, and pixels (3, 9), (3, 10) from the other. These are pixels beyond the point where the second segment touches the first. Joining the two neighbor segments after cutting of the superfluous pixels is simply done by attaching the two chains together. By joining neighbor edge segments together, PEL is able to create longer edge segments, which is important for high-level processing after edge linking.

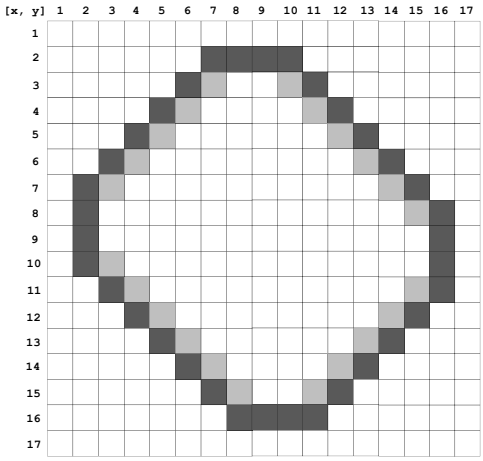

Figure 7: The need for thinning of the edge segments: Traditional edge detectors create multi-pixel wide edgels in a staircase pattern especially around diagonally placed objects. The goal of thinning is to remove superfluous edgels (light gray colored ones) and return one-pixel wide edge segments.

2.4. ThinSegments: Thinning down and cleaning up edge segments

The final step of PEL is to thin down the created edge segments and to remove the short ones from further consideration.

To see why thinning is necessary refer to Fig. 7, which shows a BEM output by a traditional edge detector for a rectangular object placed diagonally. As seen from the figure, there are multi-pixel wide edgels in a staircase pattern around the diagonals, which must be thinned down to one-pixel wide edgels. Specifically, the goal of thinning is to remove the superfluous edgels (light gray colored pixels in Fig. 7) from the chain and to return a contiguous but one-pixel wide chain, i.e., the edge segment consisting only of the dark gray pixels in Fig. 7. This is an easy procedure to perform: We simply walk over the pixels of the edge segment and remove a superfluous pixel when we see the staircase pattern depicted in Fig. 7. After the edge segment goes through thinning, the last step is to check its length and remove the segment from consideration if it is shorter than the minimum segment length supplied by the user. This is important for cleaning up noisy edgel formations in a BEM. Our observation is that a minimum segment length of eight or ten pixels produces good, clean results without omitting any important details.

3. Discussion

In this section we compare and contrast the edge detection and linking methods utilized in this paper, i.e., Canny, ED, and PEL, to make PEL's contributions clearer.

Fig. 10 depicts the typical processing pipeline for PEL and ED [25]. As seen from Fig. 10(a), PEL is a pure edge linking algorithm in that it takes in as input a binary edge map produced by a traditional edge detection algorithm such as Canny, Nalwa, Deriche, etc. [1, 3, 4], and converts the edge pixels to edge segments, filling one pixel gaps between the edgels, cleaning up noisy edgel groups and thinning down multi-pixel wide

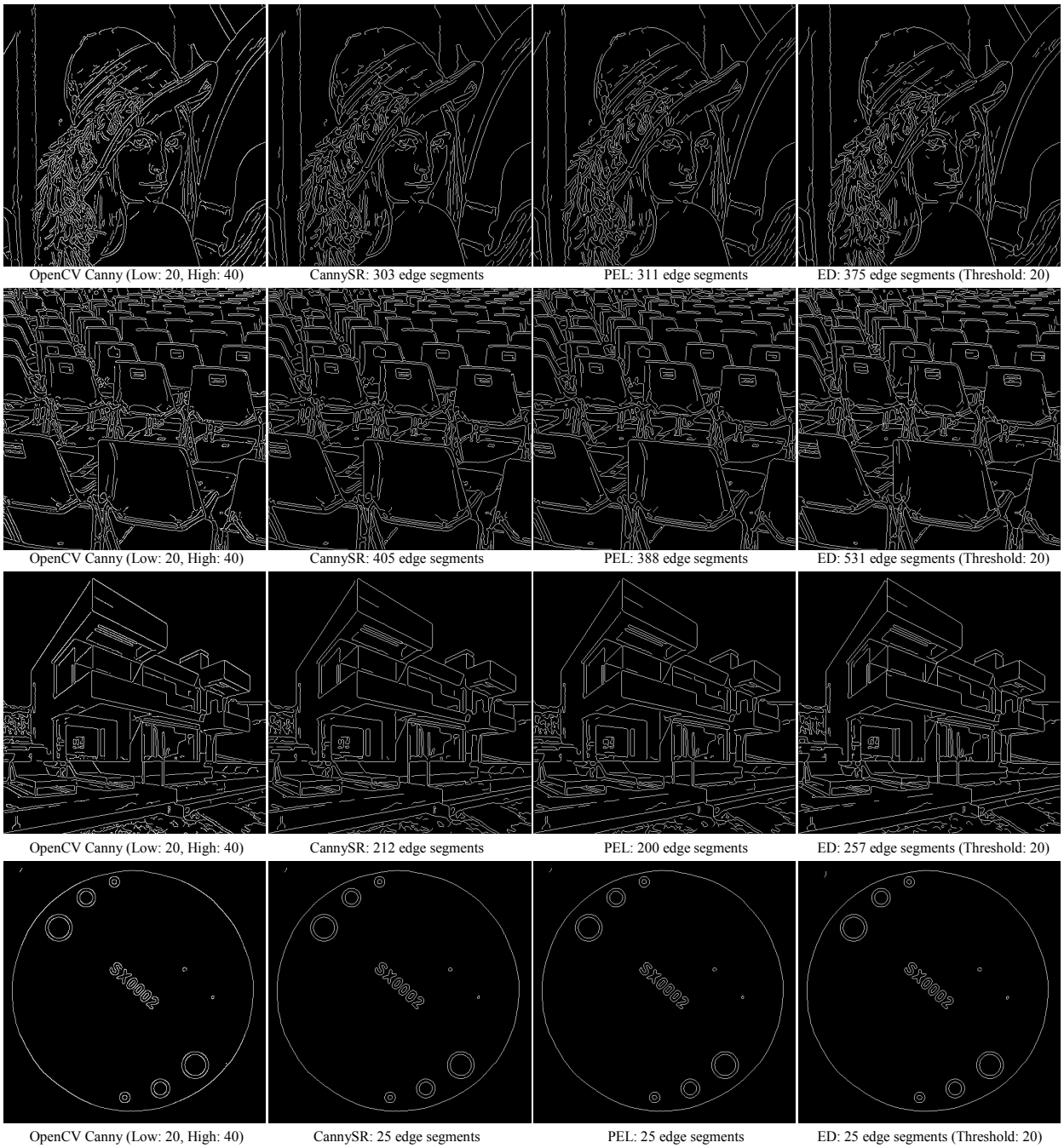

Figure 8: Canny edge maps, CannySR, PEL and ED edge segments for 4 images. The Canny edge maps were obtained by OpenCV Canny with low and high threshold parameters set to 20 and 40 respectively. The images were first smoothed by a Gaussian kernel with $\sigma = 1.5$. For CannySR and PEL, edge segments shorter than 8 pixels have been removed. ED edge segments were obtained by setting the gradient threshold to 20.

edgel formations in the process. This is illustrated in Fig. 8, which shows that PEL greatly improves the modal quality of Canny's edge maps during this post-processing step. Alternatively, PEL can also be used to post-process the binary contour maps produced by a complex multi-scale contour detector such as APD [33], gPb [34], or scg [35] to convert their binary contour maps to edge segments as depicted in Fig. 10(b). The edge segments thus produced can be used in higher level object detection applications as we show in Fig. 13. Notice that PEL by itself is not an edge or contour detection algorithm, but it is a post-processing step to convert the binary edge or contour maps produced by traditional algorithms to edge segments.

Fig. 10(c) depicts the processing pipeline for Edge Drawing (ED) [25]. As seen, ED is a single-scale edge detection algorithm similar to Canny in that ED also takes in as input a grayscale image together with the sigma of the Gausian smoothing kernel, and detects the image's edges. But unlike Canny, which outputs a binary edge map, ED outputs a set of

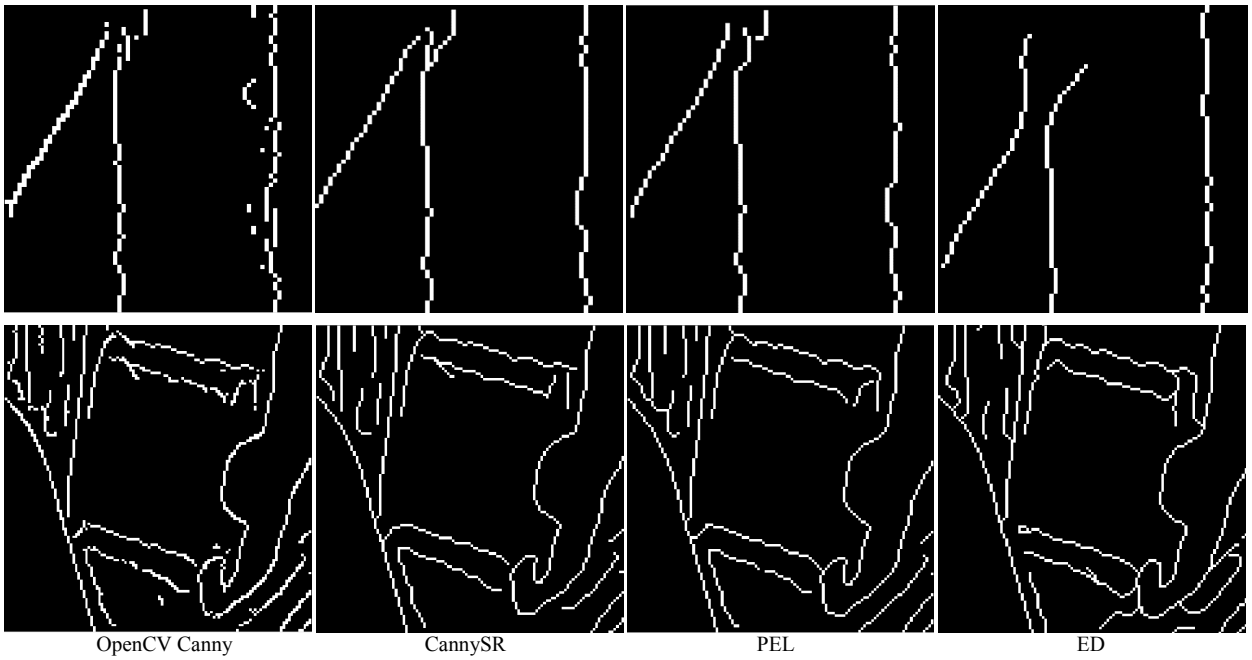

Figure 9: A close-up view of the two sections of Canny's edge map and the resulting edge segments by CannySR and PEL. ED's edge segments are also shown at the last column for comparison.

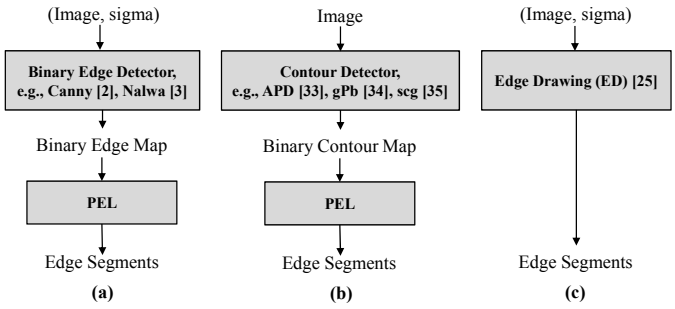

Figure 10: The processing pipeline for PEL and ED [25].

edge segments. Since ED directly outputs a set of edge segments instead of a binary edge map, there is no need to post-process ED's output by PEL. It is important to stress one more time that ED is nothing more than a single-scale edge detector, and therefore, it cannot be used to post-process neither the binary edge maps produced by Canny nor can it be used to post-process the binary contour maps produced by contour detectors. That is, ED can't be used in the place of PEL in the processing pipelines depicted in Fig. 10(a) and (b). This is the most important reason for an edge linking algorithm such as PEL.

4. Experimental Results

In this section, we evaluate the performance of PEL both qualitatively using visual experiments, and quantitatively within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS 300) [29, 30]. We use Canny [2] for edge detection since it is the most widely used binary edge detection algorithm. We compare an contrast PEL's performance to that of CannySR [24], which is a recently proposed edge linking algorithm to convert Canny's binary edge maps to edge segments, and to Edge Drawing (ED) [25], which is a natural edge segment detection algorithm.

Fig. 8 shows the Canny edge maps and the corresponding edge segments obtained by CannySR and PEL for 4 images. The Canny edge maps were obtained by OpenCV Canny implementation (cvCanny) with low and high threshold values set to 20 and 40 respectively, and the Sobel kernel aperture size set to 3. The images were first smoothed by a Gaussian kernel with σ = 1.5 (cvSmooth) before edge detection. For CannySR and PEL, edge segments shorter than 8 pixels have been considered to be noisy edgel formations and removed. Although CannySR and PEL results look similar visually, it is very clear that they both improve the modal quality of Cannys binary edge maps tremendously filling one pixel gaps and thus connecting disjoint edgel groups, removing noisy edgel formations, and thinning down multi-pixel wide edgel formations to 1-pixel wide edge segments. Last but not least, Canny's BEM has been converted to edge segments, which can now be used for higher level processing. Specifically, concentrate on the circles image (the last row), where PEL returns the result as 25 edge segments, where each edge segment is a contour of an object in the image. That is, the pixels corresponding to the periphery of each circle, number, and letter in the edge map is returned as a closed contour, which can then easily be used for such higher level detection applications as circle/ellipse detection and optical character recognition. Fig. 8 also shows the edge segments extracted by ED for comparison. To obtain ED's results, the input images were first smoothed by a Gaussian kernel with σ = 1.5, and a gradient threshold value of 20 was used during edge segment detection. Notice that ED usually extracts more

Table 2: Running times of OpenCV Canny, CannySR, PEL and ED for the 4 test images in Fig. 8 on a Core i7-3770 CPU.

Image (512x512)	Canny (ms)	CannySR (ms)	PEL (ms)	Canny + PEL (ms)	ED (ms)
Lena	5.20	10.84	2.62	7.82	4.32
Chairs	5.40	10.56	2.74	8.14	4.57
House	4.40	9.56	1.98	6.38	3.80
Circle	3.80	7.70	1.23	5.03	3.13

edge segments from the image compared to Canny for the same gradient threshold; but realise that ED's edge segments are of high-quality consisting of clean, contiguous, 1 pixel wide chain of pixels by default.

To better see the modal improvements made possible by CannySR and PEL to Canny's BEMs, consider Fig. 9 that shows a close-up view of the two sections of Canny's Lena BEM along with the resulting edge segments by CannySR and PEL. All three problems that we mentioned for binary edge maps have been solved: (1) One pixel gaps between edge groups have been filled up and long edge segments have been obtained. This is more evident in the vertical bar in the first row of Fig. 9. In Canny's BEM, this section contains a lot of 1 pixel wide gaps, all of which have been filled up and linked together by both CannySR and PEL as seen from their results. (2) Noisy edgel formations have been removed. This can clearly be seen in both Canny BEMs with many unattended, noisy edgel formations. All of these noisy edgel formations have been removed by both CannySR and PEL as seen in the second and third columns of Fig. 9. Recall that PEL has a minimum segment length parameter. Segments shorter than this threshold are removed after edge segment creation. In Fig. 9, edge segments shorter than 8 pixels have been removed as noise. (3) Multi-pixel wide edgel structures have been thinned down to one-pixel wide edge segments. This is more evident especially in diagonal egdel formations (both 45 degree and 135 degree diagonals). Looking at these diagonal edgel formations, we see the staircase pattern in Canny's BEMs, whereas both CannySR and PEL thin these edgel groups to 1-pixel wide edge segments as seen in the second and third columns of Fig. 9. All and all, it is very obvious from Fig. 8 and Fig. 9 that PEL greatly improves the modal quality of BEMs of traditional edge detectors. It is also important to stress once again that in addition to improving the modal quality of BEMs, PEL returns the result as a set of edge segments, each of which is a contiguous chain of pixels. This makes it possible to post-process these segments for such higher level applications as line, arc, circle, ellipse detection, image segmentation etc. Fig. 9 also shows the edge segments extracted by ED for the same two image sections. Notice that ED's edge segments are of good quality by default.

Table 2 shows the running time of OpenCV Canny, CannySR, PEL and ED for the 4 test images in Fig. 8. The running times reported for CannySR include both the time for edge detection by Canny and the time for the ensuing edge linking by SR. The running times were obtained on a PC with a Core i7-3770 CPU running at 3.40 GHz. We see from the table that CannySR doubles the running time of Canny with about half of the time being spent on edge detection by Canny and the remaining half spent on edge linking by SR. As for PEL, we see that PEL takes a very small amount of time to execute increasing the total running time of Canny by only 50%. Given that PEL's performance is as good as CannySR, if not better, and PEL requires nothing but the binary edge map to be linked, it is obvious that PEL is preferable over CannySR. Table 2 also gives the running time of Edge Drawing (ED) for comparison. We see that ED is faster than OpenCV Canny in all cases and is a natural edge segment detector. To obtain the edge segments for an image by first running Canny to get a binary edge map and then PEL to convert the binary edge map to edge segments is obviously costlier. But in any case, PEL takes a very small amount of time to execute, and is very useful in converting binary edge maps to edge segments.

Our goal now is to quantitatively evaluate the performance of PEL to see the amount of improvements made possible by PEL over Canny, and compare PEL's performance to that of CannySR and ED. To this end, we make use of the Berkeley Segmentation Dataset and Benchmark (BSDS 300) [29, 30] and its precision-recall evaluation framework. BSDS has 300 images with 5 to 8 human annotated boundary ground truth information for each image. 200 of these are training images and are used to tune up an algorithm's parameters. The other 100 images are used for testing an algorithm's performance.

Let the boundaries returned by an algorithm for an image be A, and the ground truth boundary information be GT. Then precision P, recall R and F-score are defined as follows, where F-score is essentially the harmonic mean of precision and recall.

$$\begin{aligned} P &= \frac{(A \cap GT)}{A} \\ R &= \frac{(A \cap GT)}{GT} \\ F-score &= \frac{(2PR)}{P+R} \end{aligned} \tag{1}$$

Fig. 11 shows the precision, recall and F-score curves for Canny, CannySR, PEL and ED as the gradient threshold is increased. Each column in the figure represents the results for a Gaussian smoothing kernel with a different sigma value. Specifically, in the first column, an input image is smoothed with a kernel with $\sigma = 1.5$ before edge detection is performed. In the second and third columns, the smoothing sigma is increased to 2.0 and 2.5 respectively. The x-axis in the graphs, i.e., the gradient threshold, is the threshold used to suppress the pixels having a gradient value smaller than the threshold. To obtain the results, we fix the gradient threshold at a specific value and use the same threshold for all images in BSDS test set for Canny and ED. Canny's binary edge maps are then fed into CannySR and PEL to obtain their results. Then the threshold is increased and a new set of results are obtained until the maximum gradient value is reached.

As seen from Fig. 11, ED produces the best F-score values with CannySR and PEL being close but much better than Canny

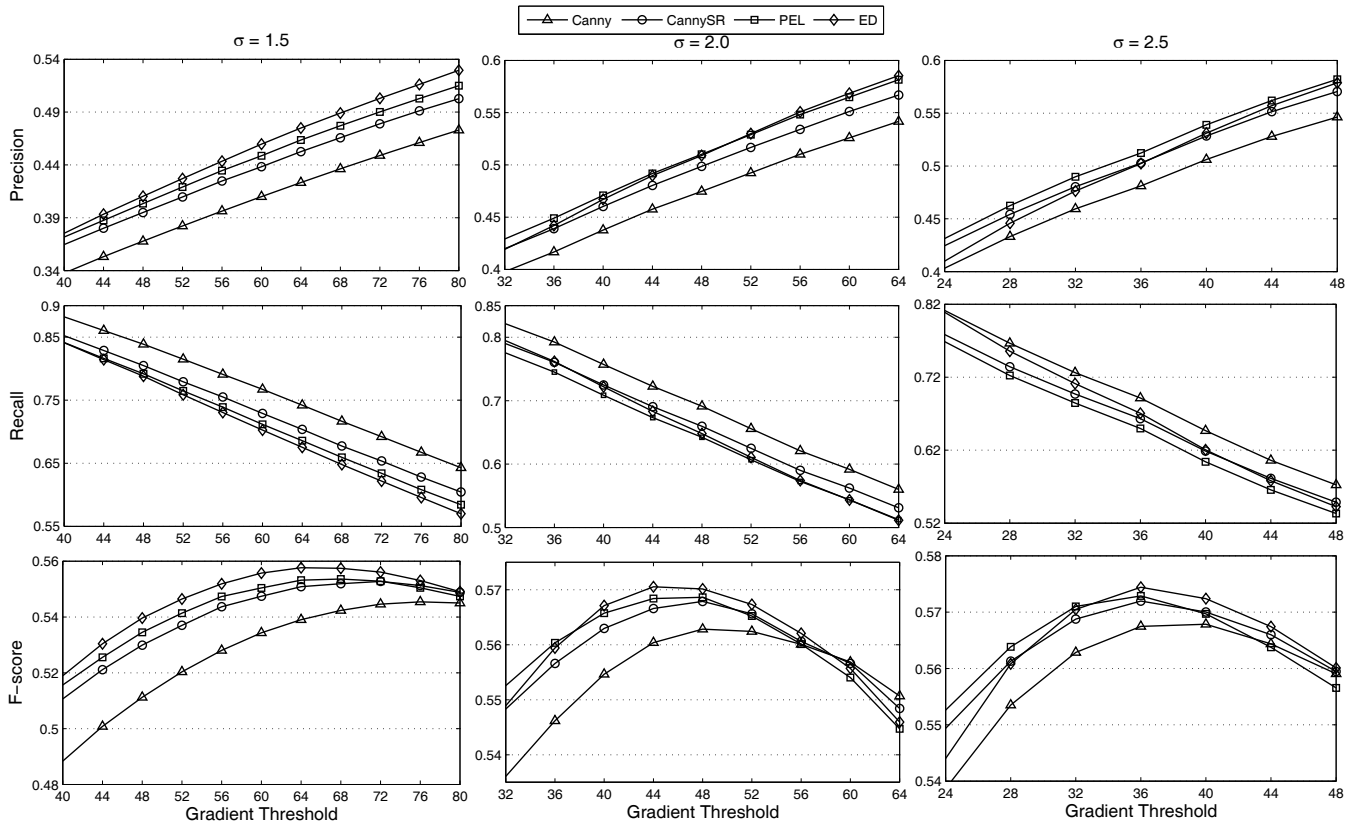

Figure 11: Precision, Recall and F-score curves for three different Gaussian smoothing kernels as the gradient threshold changes for Canny, CannySR, PEL and ED. In all cases, ED produces the best F-score values with CannySR and PEL being close but much better than Canny.

Table 3: Best F-scores values for each algorithm for three different Gaussian smoothing kernels.

Gaussian	Best F-score Value			
Sigma	Canny	CannySR	Canny+PEL	ED
1.5	0.5454	0.5527	0.5536	0.5576
2.0	0.5628	0.5678	0.5686	0.5705
2.5	0.5678	0.5719	0.5728	0.5744

in all cases. The reason for CannySR and PEL's performance improvement can be seen in the precision curves. Since CannySR and PEL clean up Canny's edge maps to a great extent (as can also be visually observed in Fig. 8 and Fig. 9), the precision curves jump up for CannySR and PEL compared to Canny. Although the recall performance for CannySR and PEL drops a little compared to Canny, the big improvements in precision performance compensates the loss in recall resulting in a much better F-score. We can also observe that ED outperforms all other algorithms for most threshold values.

Table 3 lists the best F-score values for each algorithm for three Gaussian smoothing kernels. We see from the table that both CannySR and PEL substantially improve the performance of Canny while ED performs the best.

To show that PEL can be used to process not only the binary edge maps produced by single-scale edge detectors such as Canny, but also the boundary maps produced by complex multi-scale contour detection algorithms such as the ones proposed in [33, 34, 35], Fig. 12 shows the contours detected by the global Probability of boundary (gPb) [34] (the second row) and the corresponding edge segments by PEL (the third row). As can be seen from the figure, PEL not only thins down gPb's contour maps, but it also converts them to edge segments. Table 4 lists the F-score values obtained by gPb for three cases: (1) gPb contour maps without thinning, (2) gPb contour maps after being processed by morphological thinning, (3) gPb contour maps after being processed by PEL, which not only thins down gPb's contour maps but also converts them to edge segments. The quantitative results given in Table 4 show that PEL greatly improves the performance of raw gPb contour maps, and also outperforms traditional morphological thinning. Furthermore, the comparison of the F-score values listed in Table 4 with those listed in Table 3 reveals that the multi-scale contour detection algorithm gPb, when combined with PEL, greatly outperforms all single-scale edge detectors, i.e., Canny and ED. Although gPb is quite slow compared to Canny and ED, it should be the algorithm of choice for boundary detection if the detection performance is more important than the running time.

Given that the high quality contour maps produced by gPb can be converted to edge segments by PEL, they can then be used in object detection applications and make it possible to detect objects that cannot be detected with the edge segments of a single-scale edge segment detector such as Canny+PEL or

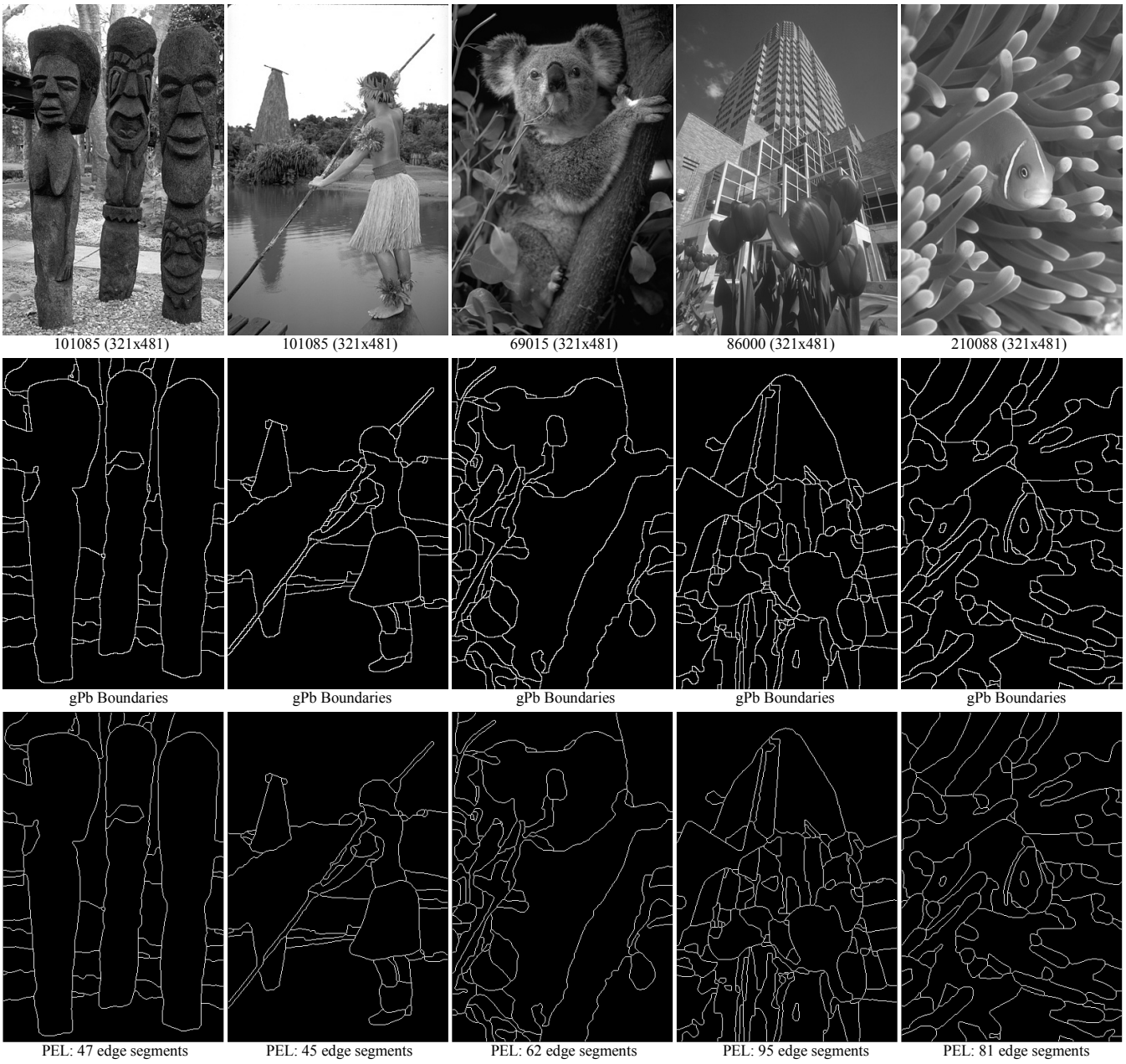

Figure 12: (First row) 5 images from the BSDS test set, (Second row) Boundaries computed by global Probability of boundary (gPb) algorithm [34], (Third row) The corresponding edge segments by PEL.

Table 4: F-scores values by gPb, gPb+morphological thinning, and gPb+PEL.

gPb (no thinning)	0.6870
gPb + morphological thinning	0.7087
gPb + PEL	0.7106

ED. As an example for this, Fig. 13 shows 5 images, each containing one or more circular objects. Specifically, the first two images contain circular objects with cluttered background, and the last three are microscopic images containing stems cells or circular diatoms [36] with fuzzy boundaries, which make detecting long, clean boundaries difficult with a single-scale edge segment detector such as ED. Fig. 13 also shows the line segments and the circles detected for all images using gPb+PEL's and ED's edge segments. In the case of gPb+PEL, the image is first fed into gPb to obtain a binary contour map, which is then fed into PEL for cleanup and edge segment generation. After PEL converts gPb's contour map to edge segments for an image, we first transform the edge segments to line segments using the method described in [26], and then post-process the line segments to detect circles using the method described in [27]. The candidate circles finally go through validation by the Helmholtz principle [31, 32] to eliminate false detections. In the case of ED, the edge segments are extracted directly from the image using the edge segment extraction algorithm described in [28],

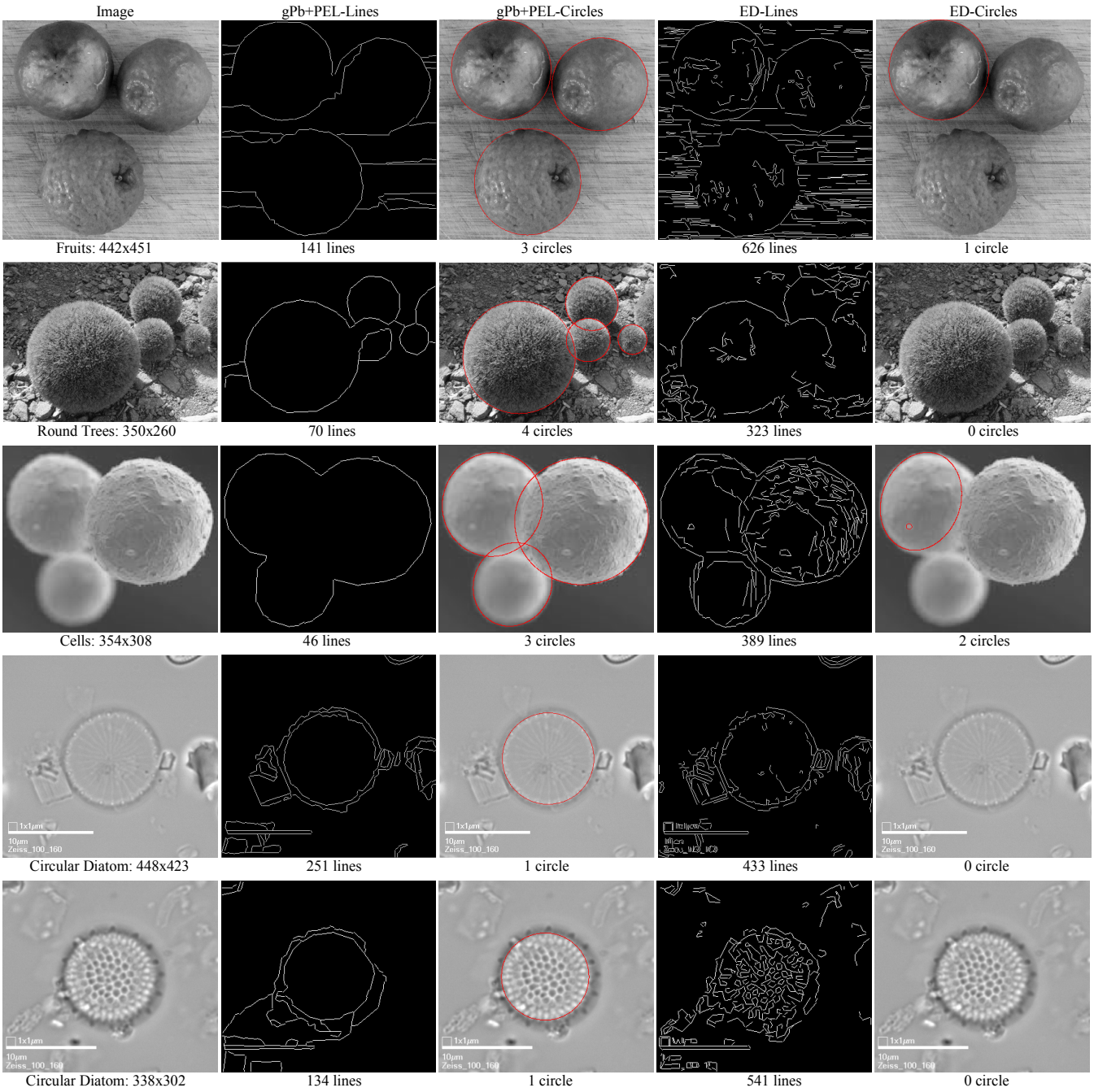

Figure 13: Line segments and circles detected for 5 images using gPb+PEL's and ED's edge segments. In the case of gPb+PEL, the image is first processed by the global Probability of boundary (gPb) [34] algorithm to obtain a binary contour map, which is then fed into PEL for cleanup and edge segment generation. Circle detection from the detected edge segments is done by the algorithm proposed in [27]. The detected circles are overlayed on top of the original images with red color.

and are then converted to line segments. As seen from Fig. 13, the circle detection performance of gPb+PEL's edge segments greatly outperform that of ED's edge segments. While it is possible to detect all circles in the images shown in Fig. 13 by gPb+PEL edge segments, most of the valid circles are not detected with ED's edge segments. This is expected since gPb, being a complex multi-scale contour detector, produces very high quality contour maps, and extracts boundaries not possible to detect with a single-scale edge segment detector such as ED. After gPb's contour map is converted to edge segments with PEL, most of the circles can now be detected. Pay special attention to the cells image (the third row), where ED is able to detect a very small circle on top the left cell while missing the circles tracing the boundaries of the two big cells; whereas, gPb+PEL detects all circles representing the boundaries of the big cells but misses out the small circle on the left cell. The reason for this can easily been seen from the line segments produced by gPb+PEL (the second column), and by ED (the fourth column). gPb, being a multi-scale contour detector, cleanly detects the boundaries of the prominent objects in an image while

wiping out fine grained details, i.e., the boundaries of the small circle on the left cell. Whereas, ED, being a single-scale edge segment detector, detects most of the fine grained details in the image including the small circle on the left cell, but is not able to cleanly extract the boundaries of the large cells (or of the objects in other images with fuzzy boundaries), which makes it impossible to detect these circles by ED-Circles.

5. Conclusions

In this paper we propose an edge linking algorithm named Predictive Edge Linking (PEL) that takes in a binary edge map generated by any traditional edge detection algorithm and converts it to a set of edge segments; filling in one pixel gaps in the edge map, cleaning up noisy edgel groups and thinning multi-pixel wide edgel formations in the process. PEL starts at an arbitrary edgel in the edge map and walks over the neighboring edgels until the end of an edgel chain is reached. During a walk, PEL consults a prediction engine that, based on the last several movements, makes a recommendation for the next move. Both qualitative and quantitative experimental results show that PEL substantially improves the modal quality of the input binary edge maps, takes very small amount of time to execute and returns its result as a set of edge segments, each of which is a chain of pixels. The edge segments can then be used in many high-level processing applications. We believe that PEL will be very useful in many real-time image processing and computer vision algorithms. Interested readers can download PEL's source code from its Web site [37].

References

[1] D. Marr, E. Hildreth, Theory of Edge Detection, Proceedings of the Royal Society of London, Biological Sciences, 207(1167) (1980), pp. 187-217.

[2] J. Canny, A computational approach to edge detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6) (1986), pp. 679-698.

[3] V.S. Nalwa, T.O. Binford, On detecting edges, IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6) (1986), pp. 699-714.

[4] E. Deriche, Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector, International Journal of Computer Vision, 1(2) (1987), pp. 167-187.

[5] OpenCV Web site, `http://opencv.org`, Accessed: July, 2015.

[6] M. Xie, Edge linking by using causal neighborhood window, Pattern Recognition Letters, 13(19) (1992), pp. 647-656.

[7] W. Snyder, R. Groshong, M. Hsiao, K. Boone, T. Hudacko, Closing gaps in edges and surfaces, Image and Vision Computing, 10(8) (1992), pp. 523-531.

[8] J. Basak, B. Chanda, D.D. Majumder, On edge and line linking with connectionist models, IEEE Transactions on Systems, Man and Cybernetics, 24(3) (1994), pp. 413-428.

[9] A. Farag, E. Delp, Edge Linking by sequential search, Pattern Recognition, 28(5) (1995), pp. 611-633.

[10] Q. Zhu, M. Payne, V. Riordan, Edge linking by a directional potential function (DPF), Image and Vision Computing, 14(1) (1996), pp. 59-70.

[11] E. Saber, A.M. Tekalp, G. Bozdagi, Fusion of color and edge information for improved segmentation and edge linking, Image and Vision Computing, 15 (1997), pp. 769-780.

[12] A. Hajjar, T. Chen, A VLSI architecture for real-time edge linking, IEEE Transactions on Pattern Analysis and Machine Intelligence, 21(1) (1999), pp. 89-94.

[13] J. Maeda, T. Iizawa, T. Ishizaka, C. Ishikawa, Y. Suzuki, Segmentation of Natural Images Using Anisotropic Diffusion and Linking of Boundary Edges, Pattern Recognition, 31(12) (1998), pp. 1993-1999.

[14] O. Ghita, P.F. Whelan, Computational approach for edge linking, The Journal of Electronic Imaging (JEI), 11(4) (2002), pp. 479-485.

[15] F.Y. Shih, S. Cheng, Adaptive mathematical morphology for edge linking, Information Sciences, 167(1-4) (2004), pp. 9-21.

[16] D.S. Lu, C.C. Chen, Edge detection improvement by ant colony optimization, Pattern Recognition Letters, 29(4) (2008), pp. 416-425.

[17] Z. Wang, H. Zhang, Edge Linking Using Geodesic Distance and Neighborhood Information, Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics, (2008), pp. 151-155.

[18] A.D. Sappa, B.X. Vintimilla, Edge Point Linking by Means of Global and Local Schemes, Signal Processing for Image Enhancement and Multimedia Processing, 31 (2008), pp. 115-125.

[19] A. Jevtic, I. Melgar, D. Andina, Ant based edge linking algorithm, Proceedings of 35th Annual Conference of the IEEE Industrial Electronics Society (IECON), (2009), pp. 3353-3358.

[20] Q. Lin, Y. Han, H. Hahn, Real-time Lane Detection Based on Extended Edge-linking Algorithm, 2nd International Conference on Computer Research and Development, (2010), pp. 725-730.

[21] J.L. Flores, G.A. Ayubi, J.R. Alonso, A. Fernandez, J.M. Di Martino, J.A. Ferrari, Edge linking and image segmentation by combining optical and digital methods, Optik, 124(18) (2013), pp. 3260-3264.

[22] X. Ji, X. Zhang, L. Zhang, Sequantial Edge Linking Method for Segmentation of Remotely Sensed Imagery based on Heuristic Search, 1st International conference on Geoinformatics, 2013.

[23] T. Guan, D. Zhou, K. Peng, Y. Liu, A Novel Contour Closure Method using Ending Point Restrained Gradient Vector Flow Field, Journal of Information Science and Engineering, 31(1) (2015), pp. 43-58.

[24] C. Akinlar, E. Chome, CannySR: Using Smart Routing of Edge Drawing to Convert Canny Binary Edge maps to Edge Segments, IEEE Int. Symposium on Innovations in Intelligent Systems and Applications, (2015).

[25] C. Topal, C. Akinlar, Edge Drawing: A Combined Real-Time Edge and Segment Detector, Journal of Visual Communication and Image Representation, 23(6) (2012), pp. 862-872.

[26] C. Akinlar, C. Topal, EDLines: a real-time line segment detector with a false detection control, Pattern Recognition Letters, 32(13) (2011), pp. 1633-1642.

[27] C. Akinlar, C. Topal, EDCircles: A real-time circle detector with a false detection control, Pattern Recognition, 46(3) (2013), pp. 725-740.

[28] C. Akinlar, C. Topal, EDPF: A Real-time Parameter-free Edge Segment Detector with a False Detection Control, International Journal of Pattern Recognition and Artificial Intelligence, 26(1) (2012).

[29] D. Martin, C. Fowlkes, D. Tal, J. Malik, A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics, IEEE International Conference on Computer Vision (ICCV), (2001), pp. 416-423.

[30] The Berkeley Segmentation Dataset and Benchmark Web site, `http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds`, Accessed: December, 2015.

[31] A. Desolneux, L. Moisan, J.M. Morel, Gestalt theory and Computer Vision, Book chapter in Seeing, Thinking and Knowing, Kluwer Academic Publishers, (2004), pp. 71-101.

[32] A. Desolneux, L. Moisan, J.M. Morel, From Gestalt Theory to Image Analysis: A Probabilistic Approach, Springer, (2008).

[33] G. Papari, N. Petkov, Adaptive Pseudo Dilation for Gestalt Edge Grouping and Contour Detection, IEEE Transactions on Image Processing, 17(10) (2008), pp. 1950-1962.

[34] P. Arbelaez, M. Maire, C. Fowlkes, J. Malik, Contour Detection and Hierarchical Image Segmentation, IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(5) (2011), pp. 898-916.

[35] X. Ren, L. Bo, Discriminatively Trained Sparse Code Gradients for Contour Detection, Advances in Neural Information Processing Systems (NIPS), (2012).

[36] Q. Luo, Y. Gao, J. Luo, C. Chen, J. Liang, C. Yang, Automatic Identification of Diatoms with Circular Shape using Texture Analysis, Journal of Software, 6(3) (2012), pp. 428-435.

[37] Predictive Edge Linking (PEL) Web site, `http://ceng.anadolu.edu.tr/cv/PEL`, Accessed: December, 2015.