Image and Vision Computing 54 (2016) 60–70

Contents lists available at ScienceDirect

Image and Vision Computing

journal homepage: www.elsevier.com/locate/imavis

CEDContours: A high speed contour detector for color images ☆

Cuneyt Akinlar

Anadolu University, Computer Engineering Department, Eskisehir, Turkey

ARTICLE INFO

Article history:
Received 2 March 2016
Received in revised form 8 June 2016
Accepted 9 August 2016
Available online 30 August 2016

Keywords:
Edge detection
Contour detection
Image segmentation
Color Edge Drawing

ABSTRACT

We propose a high-speed contour detector for color images that produces its contours as set of edge segments, each a chain of pixels. The proposed algorithm performs a multi-scale analysis of the input image by combining the edge segments produced by Color Edge Drawing (CED) at different scales; thus, the name CEDContours. We evaluate the performance of CEDContours both qualitatively by presenting visual experimental results, and quantitatively within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS300 and BSDS500). Experimental results show that CEDContours with the DiZenzo gradient operator, named CEDContours-DiZenzo, surpasses many of the prominent contour detectors found in the literature (0.70 and 0.71 F-score for BSDS300 and BSDS500 respectively), and gives comparable results to the leading contour detectors, i.e., the global Probability of boundary ultrametric contour maps (gPb-ucm: 0.71 and 0.73), and the sparse code gradients (scg: 0.72 and 0.74), but runs up to 100 times faster than these contour detectors (700ms for 481×321 images as opposed to 40s for gPb-ucm and 70s for scg), making it suitable for high-speed image processing and computer vision applications.

1. Introduction

Contours are boundaries that separate different objects in an image from each other [1]. Efficiently and correctly finding the contours in an image is a basic and fundamental problem in computer vision, and forms the basis for many applications ranging from object detection [2–4], segmentation and recognition [5], object matching and registration [6], tracking [7], scene understanding [8], etc.

A contour is a higher level feature than an edge in that it represents the change of pixel ownership from one object to another in the image, whereas an edge is a low level entity representing an abrupt change in some low level feature such as the brightness or color [1]. However, edges are usually good indicators of boundaries and to efficiently perform boundary detection in images, many grayscale and color edge detection algorithms have been proposed in the literature [9–22]. Although these edge detectors run very fast and produce acceptable boundary detection results for a wide range of images, their performance drops dramatically for natural images containing textured regions and complex cluttered backgrounds. For such images, traditional edge detectors usually produce many false detections or miss out on some important boundaries.

To cope with the contour detection problem in such complex natural images, many contour detection algorithms have been proposed in the literature. Grigorescu et al. [23] propose a contour detector for grayscale images that aims to improve the performance of classical edge detectors by trying to inhibit the response of the gradient operator on textured regions of the image. The authors employ what is called the non-classical receptive field inhibition inspired from biology for texture edge suppression, and propose a 40 image dataset for grayscale contour detection with human annotated ground truth contours [24] for quantitative evaluation. The authors in Ref. [25] incorporate the surround suppression operator to the Canny [10] edge detector and show that it performs better than the classical Canny and the SUSAN [13] edge detectors. Papari et al. [26] propose a biologically motivated multiresolution contour detector that uses Bayesian denoising together with surround inhibition. The authors improve upon this work in Ref. [27] to propose a new morphological operator that tries to identify long curvilinear structures in the edgemap by grouping of the edge pixels according to the Gestalt law of good continuation. The proposed detector can more effectively suppress texture regions and detect more low-contrast contours. Based on the similar idea of classical receptive field (CRF) inhibition, Wei et al. [28] propose a multi-scale contour detector to extract object contours from disorderly background textures. Recently, Azzopardi et al. [29] propose a grayscale contour detector based on the computational model of a simple cell, called the Combination of Receptive Fields (CORF), and extend it with push-pull inhibition in Ref. [30]. The authors test their detector on the RUG dataset and show that CORF with push-pull inhibition improves the contour detection performance and the signal to noise ratio (SNR).

☆ This paper has been recommended for acceptance by Seong-Whan Lee.

E-mail address: cakinlar@anadolu.edu.tr (C. Akinlar).

http://dx.doi.org/10.1016/j.imavis.2016.08.010
0262-8856/

The authors in Ref. [31] formulate the contour detection problem as deriving an understanding of the global structure of the image by a set of smooth curves. They model the problem using a minimum-cover framework, and use a greedy approximation algorithm to select objects minimizing a 'cost per pixel' measure. Dollar et al. [32] propose a contour detector, named Boosted Edge Learning (BEL), that tries to learn an edge classifier in the form of a probabilistic boosting tree by incorporating features from multiple scales. The algorithm uses a large aperture to provide significant context for each decision. The authors then improve their work in Ref. [33] to propose a contour detector that uses random decision forests to build a structured learning framework. Another contour detector based on supervised learning of mid-level information in the form of hand drawn contour image patches, called the sketch tokens, is presented in Ref. [34]. Maire et al. [35] also present a learning-based algorithm for contour detection and semantic labeling that does not require any hand-designed features or filters. Rather, the algorithm automatically learns the model directly from the image and the human annotated ground truth patches.

Arbelaez [36] studies the boundary extraction and segmentation of images in the context of hierarchical classification. The author represents the geometric structure of an image as a soft boundary image associated to a set of nested segmentations, called the ultrametric contour map (ucm). The root of the map represents the entire image with the leaves representing the finest details. Martin et al. [1] concentrate on contour detection in color images and propose detecting the boundaries in an image by combining features from many local cues such as the brightness, color and texture. To optimally combine the information from these local cues, the authors train a classifier using human annotated ground truth contours, and use this classifier to mark each pixel in the image with a Probability of being on the boundary (Pb). The performance of the proposed contour detector is quantitatively tested within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS300) [37,38], which contains 200 training and 100 test images with 5 to 8 human annotated ground truth boundaries for each image. Ren [39] improves upon this contour detector by combining information from multiple scales of the local cues. Arbelaez et al. [40] improve their previous contour detector proposed in Ref. [1] by using spectral clustering as a globalization technique to ensure consistency and continuity of contours. This algorithm is called the global Probability of Boundary (gPb). Authors then extend this work in Ref. [41] by making use the ultrametric contour maps proposed in Ref. [36], and name the new detector gPb-ucm, which is among the best contour detectors found in the literature. Recently, Ren and Bo [42] have proposed a contour detector to improve the performance of gPb [40] by replacing Pb's fixed gradients with sparse code gradients (scg) that are automatically learned from local patches of data. Both gPb-ucm and scg are high performance contour detectors, but they are very slow and require a lot of memory to run. To make gPb faster, the authors in Ref. [44] present a parallel implementation of gPb on the CUDA [45] architecture. But even on modern CUDA cards, gPb takes several seconds to execute. Gupta et al. [8] generalize the gPb-ucm contour detector to make use of the depth information, and test the new detector on the recently proposed NYU-Depth V2 (NYUD2) [46] dataset for contour detection, grouping, and semantic segmentation. The authors then extend this work in Ref. [47] to study the object detection problem for RGB-D images. A summary of some of the edge and line based contour detectors can be found in Ref. [48].

In this paper we propose a high-speed contour detector for color images that works by performing an analysis of the image at multiple scales using the Color Edge Drawing with Validation algorithm [15,16,22], thus the name CEDContours, which gives its output as a set of edge segments, each a chain of pixels. The edge segments can then be used for such applications as line segment detection [3], arc, circle, ellipse detection [4], object matching, tracking, recognition, image segmentation, etc. The proposed algorithm is different than edge focusing techniques [49,50], which find the edges at a large scale, and track them down to a small scale where the edges are better localized. Two variants of CEDContours have been proposed: One based on the high-speed DiZenzo color gradient operator [17,18], named CEDContours-DiZenzo, and one based on the slow Compass [19,20] gradient operator, named CEDContours-Compass. We evaluate the performance of CEDContours both qualitatively by presenting visual experimental results, and quantitatively using BSDS300 and its extended version BSDS500 [37,38], which contains 200 extra test images.

2. CEDContours: multi-scale contour detection using Color Edge Drawing

In this section, we describe the details of the proposed contour detector, CEDContours. Given a three channel color image in (R, G, B) format, our idea for contour detection is to compute the image's

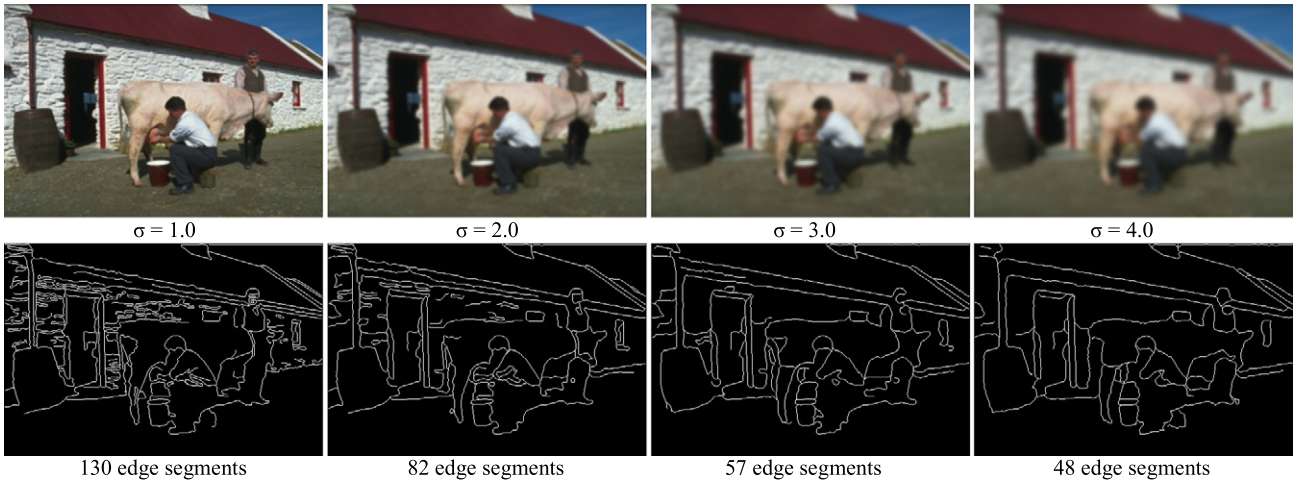

Fig. 1. Smoothed versions of the image 385039 from BSDS300 test set with increasing scale, and the corresponding edge segments detected by Color Edge Drawing with Validation (CEDV) [22]. As the scale is increased, only the prominent boundaries of the large objects get detected with finer details getting wiped out.

edge segments at different scales and fuse them together. The rationale behind this idea is the following: At smaller scales, most of the fine-grained details in the image can be detected. At higher scales, however, only the prominent boundaries of the large objects will be detected with smaller details getting wiped out by Gaussian smoothing. By combining the fine grained details detected from smaller scales of the image with the coarse grained details detected from larger scales appropriately, our goal is to design a balanced contour detector that wipes out unnecessary small details while preserving prominent boundaries separating the major objects from each other.

To illustrate this idea, Fig. 1 shows the smoothed version of the image 385039 from BSDS300 test set with increasing scale, and the corresponding edge segments detected by Color Edge Drawing with Validation (CEDV) [22]. Notice that as the scale is increased, only the prominent boundaries of the large objects (e.g., the boundaries of the cow, the humans, and the house) get detected with finer details (boundaries between the stones on the house's wall) getting wiped out.

Based on the above-mentioned idea, we propose two different variants of CEDContours: One based on the DiZenzo [17,18] color gradient operator, and one based on the Compass [19,20] color gradient operator.

Algorithm 1 gives the details of CEDContours using the DiZenzo's gradient operator, named CEDContours-Dizenzo, and Algorithm 2 gives the details of CEDContours using the Compass gradient operator, named CEDContours-Compass. Both algorithms take as input a color image in (R, G, B) format and the gradient threshold to be used for contour detection, and start by converting the (R, G, B) image to the CIE Lab [51] format (step I). Our experiments both in this work and in Ref. [22] show that the uniform CIE Lab color space gives the best performance for color edge and contour detection, which is in compliance with some of the state of the art contour detectors [1,39-42], all of which use the CIE Lab color space for contour detection. In the two ensuing steps (steps III and IV), we simply compute the edge segments from the color image at different scales using the Color Edge Drawing with Validation (CEDV) [22] algorithm, and add up the results together collecting the added result in a ContourMap image. At the end of the for loop, we thus have a soft contour map, which in a sense marks each pixel with a probability of being on the boundary of an object. The higher the probability, the more likely it is for a pixel to be output as a boundary pixel. The only difference between the two CEDContours algorithms, i.e., CEDContours-DiZenzo and CEDContours-Compass, is the operator used for gradient computation of the color image with everything else being the same.

Algorithm 1. CEDContours by DiZenzo gradient operator.

Symbols used in the algorithm:
R, G, B: Red, green, blue channels of the input color image
thresh: Gradient threshold
ES: Edge Segments

CEDContours-DiZenzo(R, G, B, thresh)
I. (L, a, b) = RGB2Lab(R, G, B);
II. ContourMap[x, y] = 0;
for sigma = 0.25 *to* MaxSigma *increment by* 0.25 **do**
  III. ES = *ColorEDV-DiZenzo*(L, a, b, sigma, thresh);
  IV. ContourMap[x, y] += ES[x, y];
end for
V. SkeltonES = *ComputeContourSkeleton*(ContourMap);
VI. *BoostSkeletonES*(SkeletonES, ContourMap);
VII. return SkeletonES;

Algorithm 2. CEDContours by Compass gradient operator.

Symbols used in the algorithm:
R, G, B: Red, green, blue channels of the input color image
thresh: Gradient threshold
ES: Edge Segments

CEDContours-Compass(R, G, B, thresh)
I. (L, a, b) = RGB2Lab(R, G, B);
II. ContourMap[x, y] = 0;
for radius = 2 *to* MaxRadius *increment by* 1 **do**
  III. ES = *ColorEDV-Compass*(L, a, b, radius, thresh);
  IV. ContourMap[x, y] += ES[x, y];
end for
V. SkeltonES = *ComputeContourSkeleton*(ContourMap);
VI. *BoostSkeletonES*(SkeletonES, ContourMap);
VII. return SkeletonES;

Fig. 2 shows the cumulative soft contour map for the image in Fig. 1 computed by CEDContours-DiZenzo at the end of the for loop, i.e., at the beginning of step V. Notice that at higher scales, the edge segments will be off by a couple of pixels from the actual location of the boundaries (called the localization error), which leads to multi-pixel wide cumulative boundaries. The bottom half of Fig. 2 shows a sample zoomed-in section of the soft contour map, e.g., the head of the standing person. The number in each box represents the number of times an edge segment has passed through that pixel at all scales. The dark gray pixels having the maximum total count represent the actual boundaries with the neighboring light gray pixels representing boundaries computed due to localization errors. Our goal is to return a soft contour map consisting of one-pixel wide edge segments rather than the multi-pixel wide contour map obtained so far. Therefore, in the rest of the algorithm, i.e., steps V and VI, we first compute a skeleton over the soft contour map, i.e., the dark gray pixels in Fig. 2 (step V), and then boost the skeleton by heaping up the neighboring pixels, i.e., the light gray pixels in Fig. 2, on top of the skeleton pixels (step V).

Algorithm 3 details how the skeleton edge segments are computed given the soft contour map. First, we compute the edge directions through each contour pixel (step I). Next a set of anchor pixels are computed (step II). Anchors are pixels of high value on the soft contour map, and are used as the starting points for edge segment generation. Finally, a new edge segment is created starting at an anchor using an 8-directional linking algorithm (step IV), and the detected edge segment is added to the set of edge segments (step V). At the end, the algorithm returns all detected edge segments (step VI), which are what is returned to the user by the contour detector after boosting (refer to Algorithm 1).

Computation of the skeleton edge segments over the contour map outlined in Algorithm 3 resembles the problem of computing the edge segments over the gradient map of an image by Smart Routing (SR) of the Edge Drawing (ED) algorithm [15], but there are several differences: (1) SR requires precise edge directions at each pixel, which are readily and correctly available during gradient computation. For our problem, however, the edge directions are not available, and must be computed heuristically; therefore, they are not as reliable as the edge directions computed during gradient computation. Furthermore, SR makes use of the edge directions at each step during linking, whereas we make use of the edge direction only once at the start of linking to guide the initial walk in a certain direction (see Algorithm 6). (2) SR employs a 4-directional linking methodology, whereas we perform an 8-directional walk over the soft contour map as described below.

Algorithm 3. Compute the skeleton edge segments over the soft contour map.

Map: Soft contour map
Dirs: Edge directions
Anchors: Anchors
Mark: The buffer that marks the skeleton pixels
SkeletonES: Skeleton edge segments

ComputeContourSkeleton(Map)
I. Dirs = `ComputeEdgeDirections`(Map);
II. Anchors = `ComputeAnchors`(Map);
III. Set all Mark[x, y] to false;
for each (x, y) in Map **do**
  if Mark[x, y] == true **or** Anchors[x, y] == false **then**
    continue;(b) Skeleton edge segments
  end if
  IV. s = `CreateEdgeSegment`(x, y, Dirs[x, y], Mark);
  V. Add s to SkeletonES;
end for
VI. **return** SkeletonES;

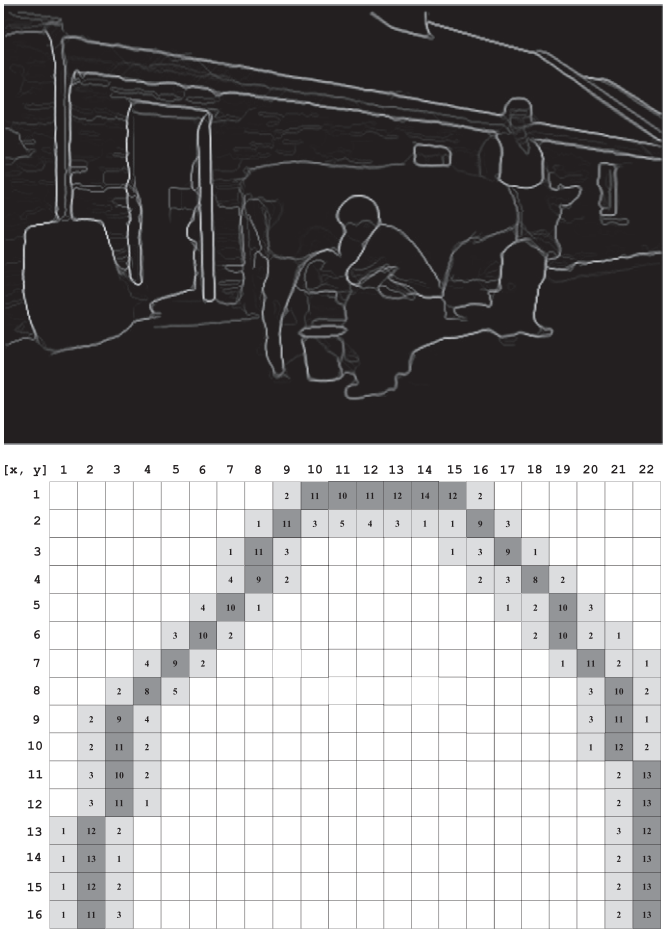

Fig. 2. (Top) Cumulative soft contour map for the image in Fig. 1 computed by CEDContours-DiZenzo. (Bottom) A sample zoomed-in section of the contour map, e.g., the head of the standing person.

Algorithm 4 gives the details of how the edge directions are heuristically computed. As seen from the algorithm, we simply assume that the edge passing through a contour pixel is in the direction of the neighbor having the maximum contour value. Therefore, for each contour pixel, we compute the neighbor pixel (*mx*, *my*) having the maximum value, and if the maximum is the left or the right neighbor, then the edge direction is assumed to be horizontal; if the maximum is the up or the down neighbor, then the edge direction is assumed to be vertical; if the maximum is the Up-Right or the Down-Left neighbor, then the edge direction is assumed to be through the 45 degree diagonal; and finally, if the maximum is the Up-Left or the Down-Right neighbor, then the edge direction is assumed to be through the 135 degree diagonal. Fig. 3 illustrates the edge directions for each contour pixel for the sample soft contour map shown in Fig. 2 computed by Algorithm 4.

Algorithm 4. Compute edge directions over the soft contour map.

Map: Soft contour map
Dirs: Edge directions

ComputeEdgeDirections(Map)
for each (x, y) in Map **do**
  Let (mx, my) be the neighbor pixel with the max value
  if ((mx == x-1 **and** my == y) **or** // *Left*
    (mx == x+1 **and** my == y)) // *Right* **then**
    Dirs[x, y] = HORIZONTAL_EDGE;
  else if ((mx == x **and** my == y-1) **or** // *Up*
      (mx == x **and** my == y+1)) // *Down* **then**
    Dirs[x, y] = VERTICAL_EDGE;
  else if ((mx == x+1 **and** my == y-1) **or** // *Up-Right*
      (mx == x-1 **and** my == y+1)) // *Down-Left* **then**
    Dirs[x, y] = 45DEG_EDGE;
  else
    Dirs[x, y] = 135DEG_EDGE; // *Up-Left or Down-Right*
  end if
end for
return Dirs;

Having computed the edge directions, we now move to computing the anchor pixels. Algorithm 5 details how the anchors are computed. The idea is simple: A pixel (x, y) is an anchor only if the

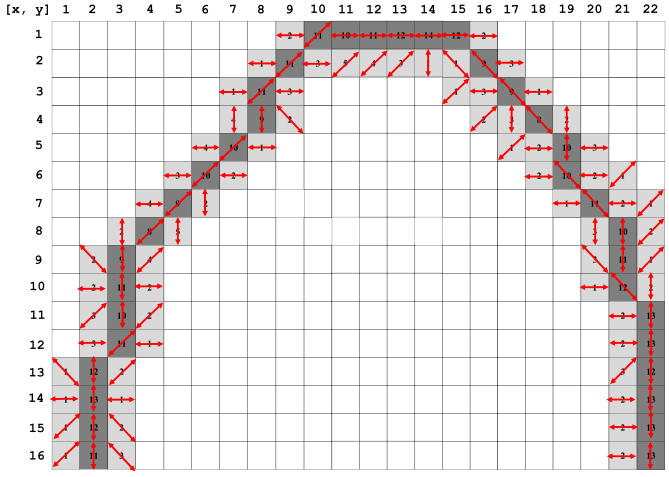

Fig. 3. Edge directions for each contour pixel computed by Algorithm 4.

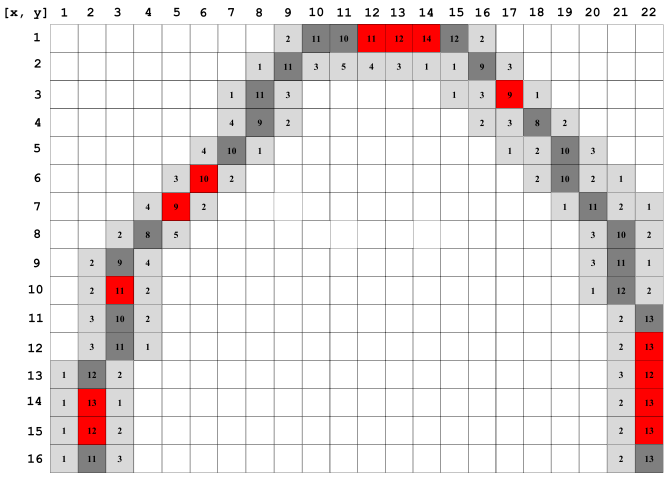

Fig. 4. Anchors (red pixels) computed by Algorithm 5 for the sample soft contour map in Fig. 2.

two neighbors in the direction of the edge passing through (x, y) also have the same edge direction. For example, if there is a horizontal edge passing through (x, y), then for (x, y) to be an anchor, the edge passing through its left, i.e., $(x-1, y)$, and right, i.e., $(x+1, y)$, neighbors must also be horizontal. The 3 other directions follow the same logic. Fig. 4 depicts the anchor pixels computed for the sample soft contour map shown in Fig. 2. As seen, of the many contour pixels, only a few (13 to be exact) are marked as anchors. These pixels will be our stable points to start the edge segment creation.

Algorithm 5. Computing the anchor pixels.

Map: Soft contour map
Dirs: Edge directions
Anchors: Stable pixels to start linking

```
ComputeAnchors(Map, Dirs)
for each (x, y) in Map do
  if (Dirs[x, y] == HORIZONTAL_EDGE) then
    if (Dirs[x-1, y] == HORIZONTAL_EDGE and
        Dirs[x+1, y] == HORIZONTAL_EDGE) then
      Anchors[x, y] = true;
    end if
  else if (Dirs[x, y] == VERTICAL_EDGE) then
    if (Dirs[x, y-1] == VERTICAL_EDGE and
        Dirs[x, y+1] == VERTICAL_EDGE) then
      Anchors[x, y] = true;
    end if
  else if (Dirs[x, y] == 45DEG_EDGE) then
    if (Dirs[x+1, y-1] == 45DEG_EDGE and
        Dirs[x-1, y+1] == 45DEG_EDGE) then
      Anchors[x, y] = true;
    end if
  else if (Dirs[x, y] == 135DEG_EDGE) then
    if (Dirs[x-1, y-1] == 135DEG_EDGE and
        Dirs[x+1, y+1] == 135DEG_EDGE) then
      Anchors[x, y] = true;
    end if
  end if
end for
```

Algorithm 6. Create one edge segment starting at pixel (x, y) with 'dir' as the initial edge direction.

(x, y): Starting anchor position
dir: Edge direction at the anchor pixel
Mark: The buffer that marks the skeleton pixels

```
CreateOneEdgeSegment(x, y, dir, Mark)
if (dir == HORIZONTAL_EDGE) then
  Chain1 = Walk(x, y, Left, Mark);
  Chain2 = Walk(x, y, Right, Mark);
else if (dir == VERTICAL_EDGE) then
  Chain1 = Walk(x, y, Up, Mark);
  Chain2 = Walk(x, y, Down, Mark);
else if (dir == 45DEG_EDGE) then
  Chain1 = Walk(x, y, Up-Right, Mark);
  Chain2 = Walk(x, y, Down-Left, Mark);
else if (dir == 135DEG_EDGE) then
  Chain1 = Walk(x, y, Up-Left, Mark);
  Chain2 = Walk(x, y, Down-Right, Mark);
end if
return Chain1 (in reverse order) + Chain2;
```

Having computed the edge directions and the anchors, we now sort the anchors with respect to their values, and start the creation of an edge segment with the anchor having the maximum value. In Fig. 4, this would be pixel (14, 1) having the maximum value of 14 among all anchors. Algorithm 6 describes how an edge segment is created starting at an anchor point (x, y), whose edge direction is given by 'dir'. Using the initial edge direction, we start two Walks in reverse direction. For example, if the edge direction at (x, y) is horizontal, then we start two Walks: One to the Left and one to the Right. The remaining 3 directions follow a similar logic. At the end of the two walks, we have two chains, which are combined together and returned as an edge segment. Notice that the two chains are in

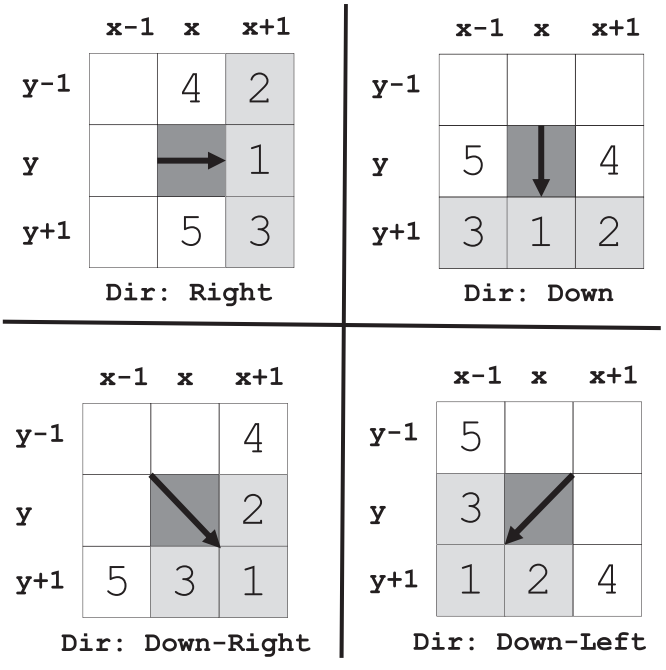

Fig. 5. Four walk directions: Right, Down, Down-Right, Down-Left. The other four directions, i.e., Left, Up, Up-Left, and Up-Right, are the symmetric versions of these.

reverse direction, so one of the chains is reversed and then attached to the other to form the returned edge segment.

Algorithm 7 shows how one chain is created starting at pixel (x, y) with the initial walk direction given by 'dir'. As seen, we perform an 8-directional walk over the soft contour map. At each step of the algorithm, we move to the neighboring pixel (nx, ny) along the current direction with the next direction specified by 'nDir', which may be the same as the current direction or change based on the move. If the end of the chain is reached or if we encounter a previously marked skeleton pixel, then the loop terminates and the chain is returned.

Algorithm 7. Walk until we reach the end of a chain.

```
(x, y): Current chain pixel
(nx, ny): Next pixel to move to
dir: Current edge direction
Mark: The buffer that marks the skeleton pixels
Chain: Chain pixels

Walk(x, y, dir, Mark)
Chain = null;
while (1) do
   Add (x, y) to Chain;
   Mark[x, y] = true;
   if (dir == Left) then
      (nx, ny), nDir = MoveLeft(x, y);
   else if (dir == Right) then
      (nx, ny), nDir = MoveRight(x, y);
   else if (dir == Up) then
      (nx, ny), nDir = MoveUp(x, y);
   else if (dir == Down) then
      (nx, ny), nDir = MoveDown(x, y);
   else if (dir == Up-Left) then
      (nx, ny), nDir = MoveUp-Left(x, y);
   else if (dir == Up-Right) then
      (nx, ny), nDir = MoveUp-Right(x, y);
   else if (dir == Down-Right) then
      (nx, ny), nDir = MoveDown-Right(x, y);
   else if (dir == Down-Left) then
      (nx, ny), nDir = MoveDown-Left(x, y);
   end if
   if (End of the chain or Mark[nx, ny] == true) then
      Break out of the while loop;
   end if
   (x, y), dir = (nx, ny), nDir;
end while
return Chain;
```

Fig. 5 illustrates how the walk proceeds in four directions: Right, Down, Down-Right, and Down-Left. The other four directions, i.e., Left, Up, Up-Left, and Up-Right, are the symmetric versions of these, and are not illustrated. In the figure, we are at pixel (x, y) and moving in the direction of the arrow. At each step, we first check the three light gray neighbor pixels labeled 1, 2 and 3, and move to the one having the largest value. If pixel 1 has the largest value, we move there and the current walk direction stays the same. Otherwise, if we move to one of the pixels labeled 2 or 3, then the current direction changes. For example, assume that the current direction is Right, and the neighbor $(x + 1, y - 1)$ labeled 2 has the biggest value. Then we move there and the current walk direction changes from Right to Up-Right. If all of the pixels labeled 1, 2, and 3 have zero values, then we check the pixels labeled 4 and 5, and move to the one having the larger value. The reason for this check is to make the walk turn and continue when we reach the end of a pixel group. Algorithm 7 shows how the Right-move is implemented. Moves to the other directions are implemented similarly following the logic depicted in Fig. 5.

Algorithm 8. Move one pixel in the Right direction

```
(x, y): Current position of the chain
(nx, ny): Next pixel to move to
dir: Next walk direction

MoveRight(x, y)
max = Map[x+1, y]; // Location marked 1
dir = Right; (nx, ny) = (x+1, y);
if (Map[x+1, y-1] > max) then
   max = Map[x+1, y-1]; // Location marked 2
   dir = Up-Left; (nx, ny) = (x+1, y-1);
else if (Map[x+1, y+1] > max) then
   max = Map[x+1, y+1]; // Location marked 3
   dir = Down-Right; (nx, ny) = (x+1, y+1);
end if
if (max == 0) then
   max = Map[x, y-1]; // Location marked 4
   dir = Up; (nx, ny) = (x, y-1);
   if (Map[x, y+1] > max) then
      max = Map[x, y+1]; // Location marked 5
      dir = Down; (nx, ny) = (x, y+1);
   end if
end if
if (max > 0) then
   return (nx, ny), dir;
else
   return "End of the chain";
end if
```

Fig. 6 exemplifies the computation of one skeleton edge segment starting at pixel (14, 1). Since the edge direction at (14, 1) is a horizontal edge, we start two walks: One to the left, and one to the right (refer to Algorithm 6). During each step of the walk (refer to Algorithm 7), we check the 3 neighbors along the current direction using the rules depicted in Fig. 5, and move to the neighbor having the biggest value. For example, starting at (14, 1), the left walk continues until (10, 1), where the direction changes to Down-Left, then changes to Down at (8, 3), and then back to Down-Left at (8, 4). Following the red arrows over the dark gray pixels, this chain would end up at pixel (2, 16). We then start a walk to the right from (14, 1), which would follow the red arrows in the figure mostly moving Down and Down-Right, and end up at pixel (22, 16). We now have two chains in reverse order. One of the chains will be reversed and attached to the other. Assuming that the first chain is reversed, the final edge segment would start at (2, 16) and end up at (22, 16) consisting of 36 pixels.

Having computed the skeleton edge segments, the last step of the contour detection algorithm (step VI in Algorithms 1 and 2) is to heap up the neighboring pixels of the skeleton on top of the skeleton pixels. The idea is to simply compute the nearest skeleton pixel, called the master pixel, for each non-skeleton pixel and add its value to the

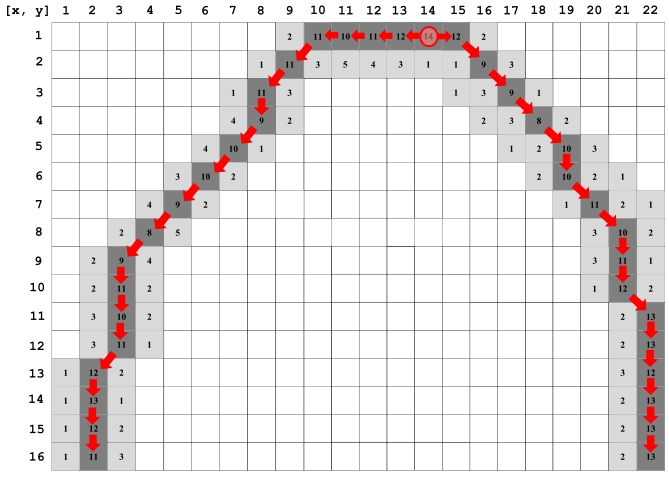

Fig. 6. Generation of the edge segment starting at anchor pixel (14, 1).

master pixel. At the end of this step, the values of the skeleton pixels will be boosted up by the values of the neighboring non-skeleton pixels. To give an example, for non-skeleton pixels (1, 16) and (3, 16), the master skeleton pixel is (2, 16). We simply add the values of the non-skeleton pixels to (2, 16) boosting its value to 15. All the other non-skeleton pixels follow the same idea and end up boosting the value of their nearest skeleton pixel. This is the final output of CEDContours: A set of edge segments, each a chain of pixels with each pixel having a probability of being on the boundary of an object. Fig. 7 shows the final values of the skeleton edge segment pixels after boosting for the sample soft contour map. To obtain a binary contour map from this soft-contour map, the soft contour map is thresholded at a specific value. The skeleton pixels whose values are greater than the threshold survive, while the remaining skeleton pixels are suppressed.

3. Experimental results

In this section, we evaluate the performance of CEDContours both qualitatively by showing visual experimentation results, and quantitatively within the precision-recall framework of the Berkeley Segmentation Dataset and Benchmark (BSDS) [37,38]. BSDS has two

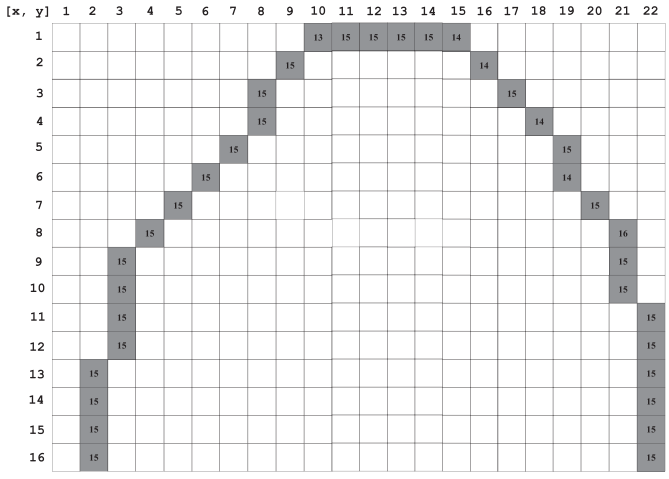

Fig. 7. The final values of the skeleton edge segment pixels after boosting for the sample soft contour map.

versions: BSDS300, with 200 training and 100 test images; BSDS500, with an additional 200 test images. Each image in the dataset has 5 to 8 human annotated ground truth boundary information. The 200 training images are used to tune up an algorithm's parameters, and once the parameters are fixed, the algorithm is run with the test images. The quantitative evaluation is performed by comparing the boundaries produced by an algorithm with the human annotated ground truth boundaries.

Let the boundaries returned by an algorithm for an image be A, and the human annotated ground truth boundary 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.

$$P = \frac{(A \cap GT)}{A}$$
$$R = \frac{(A \cap GT)}{GT}$$
$$F - score = \frac{(2PR)}{P + R} \quad (1)$$

Recall from Algorithm 1 that CEDContours-DiZenzo has one user-supplied parameter, the gradient threshold (thresh), and two internal parameters: the maximum scale (MaxSigma), and the scale step size, which is fixed at 0.25. Our experiments have shown that a step size of 0.25 gives the best results. Step size values smaller than 0.25 does not change the result; bigger values, e.g., 0.50, produce slightly worse results. Therefore, we fix the step size to 0.25 for CEDContours-DiZenzo. Notice that making the step size smaller increases the total computation time; making it smaller reduces it. To determine the optimal (thresh, MaxSigma) parameters for CEDContours-DiZenzo, we have run the algorithm using the training images.

Table 1 shows the best F-score values obtained by CEDContours-DiZenzo for different gradient threshold and MaxSigma values for the BSDS300 training set. Although the results are very close to each other, the optimal parameters seem to be (32, 4.75) for the gradient threshold and MaxSigma parameters respectively. We have repeated the same experiment for CEDContours-Compass, and although we do not present the experiment results, the optimal values turned out to be (60, 15) for the gradient threshold and MaxRadius parameters respectively. In the rest of the experiments, we use these fixed parameters to obtain the results for BSDS300 and BSDS500 test images running CEDContours algorithms.

Fig. 8 shows seven images from BSDS300 test set along with human-annotated ground truth contours, and the corresponding thresholded best contour detection results by two of the best state of the art contour detection algorithms; namely, global Probability of boundary-ultrametric contour maps (gPb-ucm) [41], and sparse code gradients (scg) [42]. The figure also presents the results for the proposed CEDContours-DiZenzo and CEDContours-Compass algorithms. Similarly, Fig. 9 shows seven images from BSDS500 test set along with human-annotated ground truth contours, and the corresponding thresholded best contour detection results by the same contour detection algorithms. To obtain the contour detection results for gPb-ucm, we downloaded its freely distributed code from the BSDS Web

Table 1
Best F-score values for CEDContours-DiZenzo for different gradient threshold and MaxSigma values. (32, 4.75) seems to be the best choice.

	MaxSigma		
Gradient threshold	4.50	4.75	5.00
28	0.7075	0.7080	0.7080
32	0.7078	**0.7084**	0.7084
36	0.7075	0.7079	0.7079

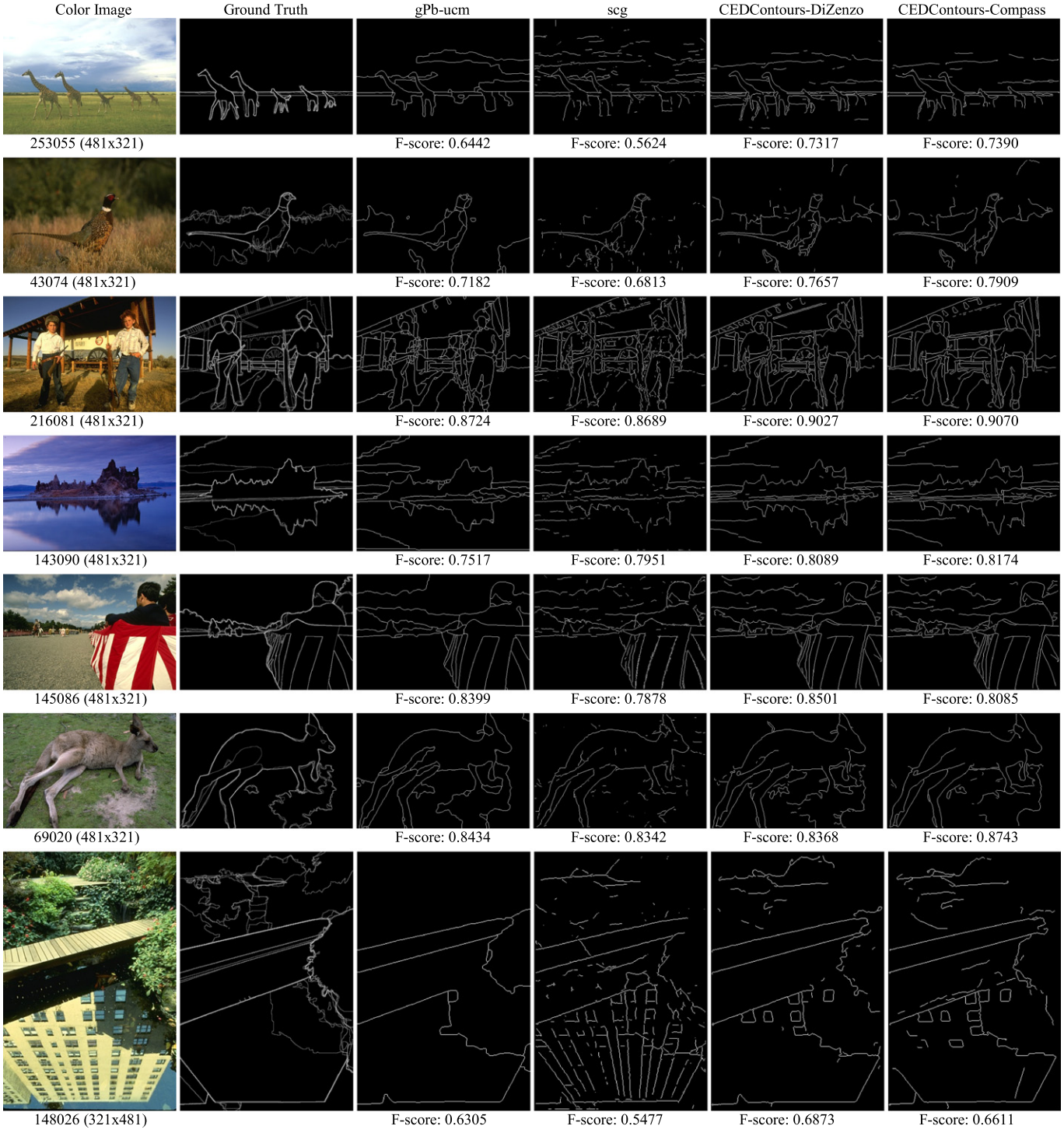

Fig. 8. Seven images from BSDS300 test set along with human-annotated ground truth contours, and the corresponding thresholded best contour detection results by four algorithms, global Probability of boundary-ultrametric contour maps (gPb-ucm) [41], sparse code gradients (scg) [42], CEDContours-DiZenzo and CEDContours-Compass. For all of these images, one of the CEDContours variant produces the best results as measured by the F-score metric.

site [38]. Similarly, the code for 'scg' was downloaded from its Web site [43].

For all of images shown in Figs. 8 and 9, one of the CEDContours variant produces the best results as measured by the F-score metric. Looking at the results visually, we see that the contour maps produced by 'scg' have many noisy formations even though 'scg' gives the best overall F-score as given in Table 2. CEDContours results, however, are very clean, and are returned as a set of edge segments, each consisting of a contiguous chain of pixels. The edge segments can readily be used for higher layer processing for such applications as line segment detection [3], arc, circle and ellipse detection [4], and similar object detection and image segmentation applications.

For the 100 test images in BSDS300, CEDContours-DiZenzo gives the best score on 17 of the 100 images (17%), and CEDContours-Compass gives the best score on 19 of the 100 images (19%) compared to gPb and scg, which are the best two contour detection

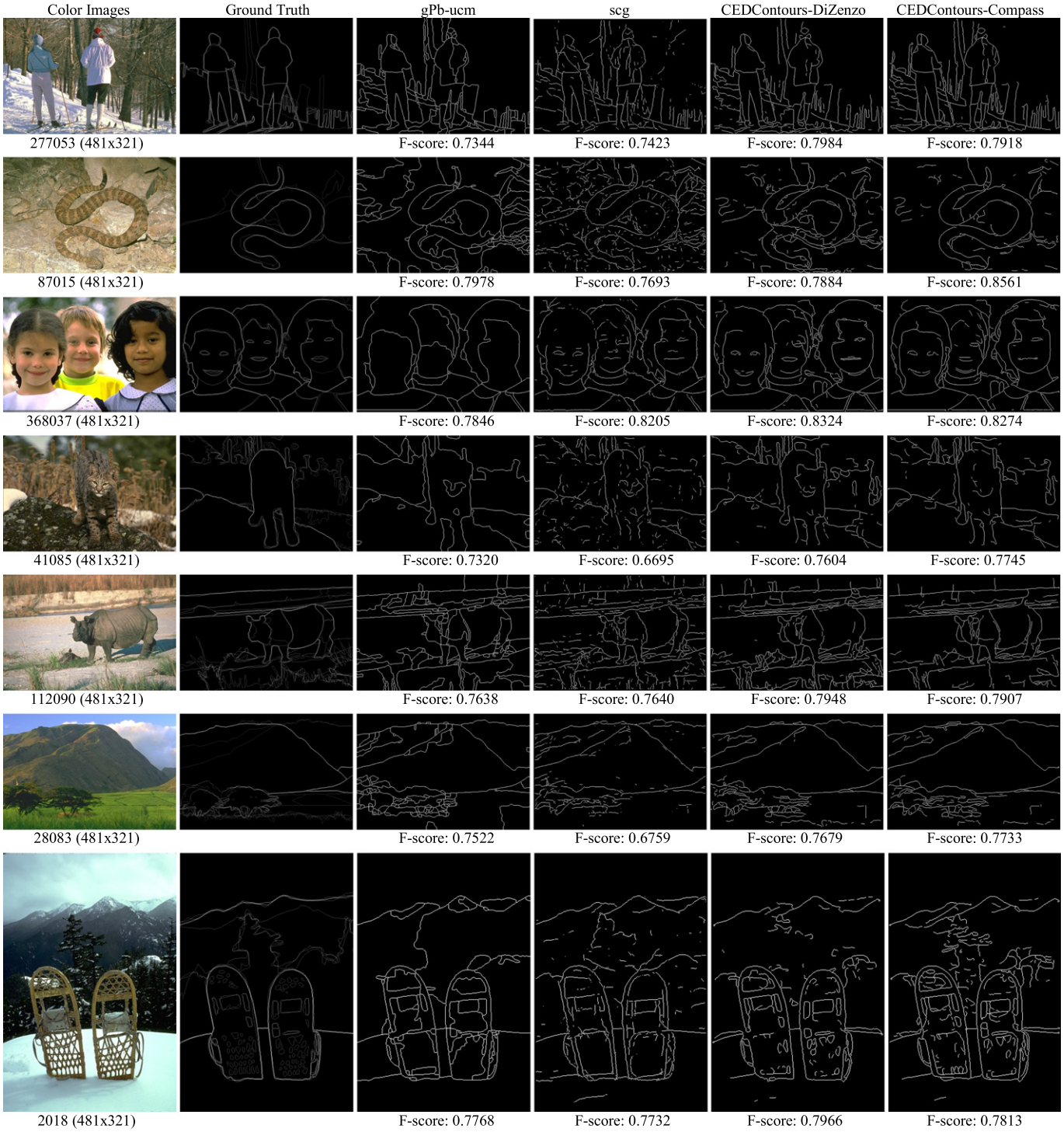

Fig. 9. Seven images from BSDS500 test set along with human-annotated ground truth contours, and the corresponding thresholded best contour detection results by four algorithms, global Probability of boundary-ultrametric contour maps (gPb-ucm) [41], sparse code gradients (scg) [42], CEDContours-DiZenzo and CEDContours-Compass. For all of these images, one of the CEDContours variant produces the best results as measured by the F-score metric.

algorithms in the literature. For the 200 test images in the BSDS500, CEDContours-DiZenzo gives the best score on 27 images (14%), and CEDContours-Compass gives the best score on 37 images (19%).

Figs. 10 and 11 plot the performance of some prominent contour detectors on the BSDS300 and BSDS500 benchmarks [37,38] respectively, for different precision-recall values. The ranking of the detectors is based on the F-score metric given in Eq. (3). Iso-F curves are shown in green, and the average agreement between the human subjects is shown by the green dot. Notice from the figures that CEDContours outperforms most of the contour detectors found in the literature, and finds itself a place among the best contour detectors.

To quantitatively compare and contrast the performance of CEDContours with several state of the contour detectors, Table 2 lists the best overall F-score value produced by each algorithm on the

Table 2
The best F-score values by different algorithms on the BSDS300 test set.

Algorithm	Recall	Precision	F-score
Mincover [31]	0.6968	0.6067	0.6486
BEL [32]	0.7008	0.6313	0.6643
Arbelaez [36]	0.6861	0.6583	0.6719
Xren [39]	0.7239	0.6324	0.6751
CEDContours-DiZenzo	0.7077	0.6836	0.6954
gPb [40]	0.7113	0.6842	0.6975
CEDContours-Compass	0.7125	0.6979	0.7051
gPb-ucm [41]	0.7040	0.7134	0.7087
scg [42]	0.7387	0.6928	0.7150

BSDS300 test set. As seen from the table, CEDContours-DiZenzo gives the fifth best overall score surpassing most of the contour detectors in the literature, and finding a place just behind gPb [40]. CEDContours-Compass gives the third best overall score, surpassed only by gPb-ucm [41] and scg [42]. Table 3 gives the best overall scores by the leading contour detectors on the BSDS500 test set, and we again see a similar ranking of the algorithms.

Table 4 lists the average running time of the most prominent contour detection algorithms for images of size 481 × 321 or 321 × 481 pixels in the BSDS test sets. The running times were obtained on a machine having an Intel Xeon E5-1630 processor running at 3.70 GHz, and 32 GB of memory. As seen from the table, all prominent contour detectors take very long time to execute with gPb-ucm taking 40 s and scg taking 70 s per image on average. These algorithms also require up to 15 GB of memory during execution for a single image in the dataset. Although CEDContours-Compass requires only several MB of memory to execute, it is also very slow due to the enormous amounts of time it takes to compute the Compass gradient. CEDContours-DiZenzo, however, takes a mere 700 ms to execute for an image of size 481 × 321 or 321 × 481, and is thus 57 times faster than gPb-ucm, and 100 times faster than scg.

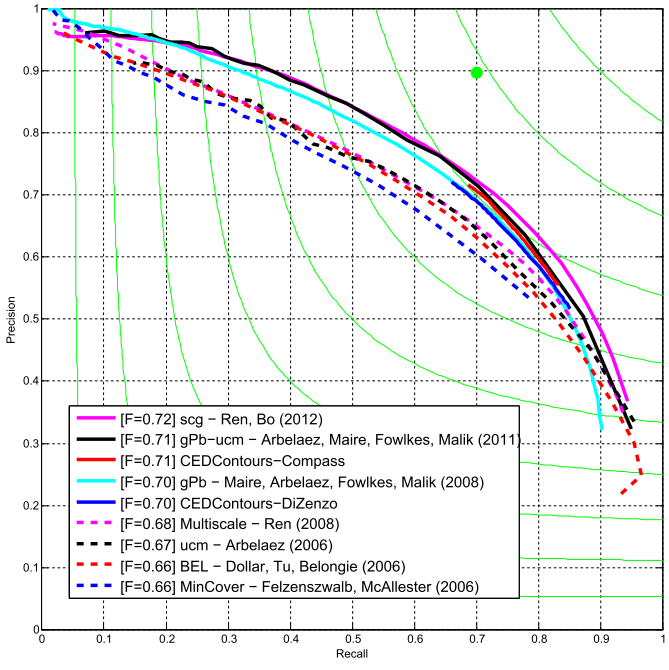

Fig. 10. Performance of the leading contour detectors on the BSDS300 benchmark [37,38] for different precision-recall values. The ranking of the detectors is based on the F-score metric given in Eq. (3). Iso-F curves are shown in green, and the average agreement between the human subjects is shown by the green dot. CEDContours finds itself a place among the best contour detectors.

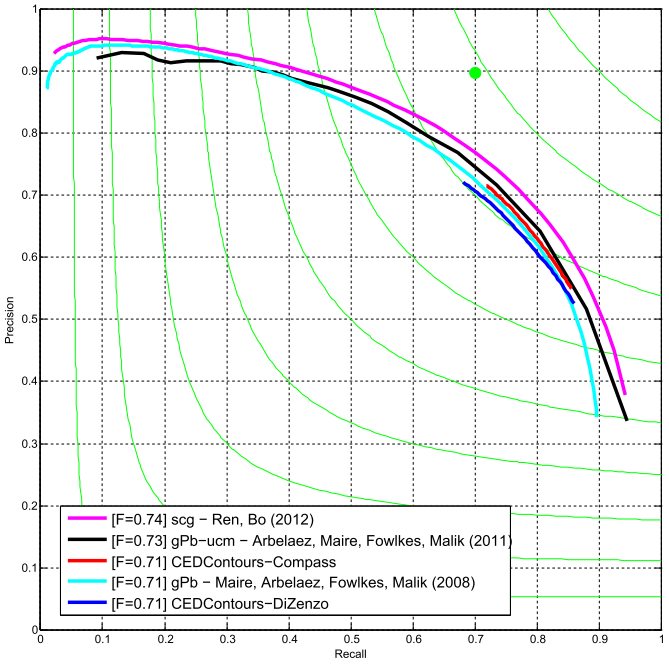

Fig. 11. Performance of the leading contour detectors on the BSDS500 benchmark [37,38] for different precision-recall values. The ranking of the detectors is based on the F-score metric given in Eq. (3). Iso-F curves are shown in green, and the average agreement between the human subjects is shown by the green dot. CEDContours finds itself a place among the best contour detectors.

Although CEDContours-DiZenzo's overall contour detection performance is slightly worse than gPb-ucm and scg as seen from Tables 2 and 3, it is up to 100 times faster than these detectors and requires only megabytes of memory during execution as opposed to gigabytes of memory. This makes CEDContours-DiZenzo very suitable for high-speed image processing and computer vision applications, for which gPb-ucm and scg cannot be utilized.

4. Conclusions

We present a contour detection algorithm for color images that works by incorporating edge segments detected at multiple scales using our recently proposed Color Edge Drawing (CED) algorithm, thus the name CEDContours. Two variants of CEDContours are proposed: One that makes use of the Compass gradient operator, named CEDContours-Compass; and one that makes use of the DiZenzo gradient operator, named CEDContours-DiZenzo. Although CEDContours-Compass produces better results, it is very slow due to the massive amount of time it takes to run the Compass operator. CEDContours-DiZenzo, however, is very high-speed, and runs up to 100 times faster than the best contour detectors in the literature, e.g., gPb-ucm and scg, while showing comparable performance, and surpassing the performance of many of the prominent contour detectors found in the literature. We believe that CEDContours-DiZenzo will be very useful for high-speed image processing and computer

Table 3
The best F-score values by different algorithms on the BSDS500 test set.

Algorithm	Recall	Precision	F-score
CEDContours-DiZenzo	0.7239	0.6885	0.7057
gPb [40]	0.7286	0.6995	0.7138
CEDContours-Compass	0.7338	0.7036	0.7184
gPb-ucm [41]	0.7343	0.7175	0.7258
scg [42]	0.7501	0.7293	0.7396

Table 4
Average running time of some prominent contour detection algorithms for images of size 481×321 or 321×481pixels in the BSDS test sets. The running times were obtained on a machine having an Intel Xeon E5-1630 processor running at 3.70GHz, and 32GB of memory.

Algorithm	Running time (sec)
CEDContours-DiZenzo	0.700
CEDContours-Compass	120
gPb [40]	38
gPb-ucm [41]	40
scg [42]	70

vision applications. Interested readers can download CEDContours's code along with CEDContours's results for BSDS300 and BSDS500 benchmarks from its Web site [52]. Our future work is to look into methods of texture inhibition and incorporation of a texture gradient to the proposed algorithms. Furthermore, we will be looking into ways to automatically determine the optimal gradient threshold for each image.

References

[1] D. Martin, C. Fowlkes, J. Malik, Learning to detect natural image boundaries using local brightness, color and texture cues, Trans. Pattern Anal. Mach. Intell. 26 (5) (2004) 530–549.
[2] V. Ferrari, T. Tuytelaars, L.V. Gool, Object detection by contour segment networks, Eur. Conf. Comput. Vis. (ECCV) (2006) 1428.
[3] C. Akinlar, C. Topal, EDLines: A real-time line segment detector with a false detection control, Pattern Recogn. Lett. 32 (13) (2011) 1633–1642.
[4] C. Akinlar, C. Topal, EDCIrcles: A real-time circle detector with a false detection control, Pattern Recogn. 46 (3) (2013) 725–740.
[5] C. Gu, J. Lim, P. Arbelaez, J. Malik, Recognition using regions, Comput. Vis. Pattern Recognit. (CVPR) (2009) 1030–1037.
[6] H. Lui, B.S. Manjunath, S.K. Mitra, A contour-based approach to multisensor image registration, IEEE Trans. Image Process. 4 (3) (1995) 320–334.
[7] W. Hu, X. Zhou, W. Li, X. Luo, S. Zhang, Maybank, Active contour-based visual tracking by integrating colors, shapes, and motions, IEEE Trans. Image Process. 22 (5) (2013) 1778–1792.
[8] S. Gupta, P. Arbelaez, J. Malik, Perceptual organization and recognition of indoor scenes from RGB-d images, Conf. Comput. Vis. Pattern Recognit. (CVPR) (2013) 564–571.
[9] D. Marr, E. Hildreth, Theory of edge detection, Proceedings of the Royal Society of London, Biol. Sci. 207 (1167) (1980) 187–217.
[10] J. Canny, A computational approach to edge detection, IEEE Trans. Pattern Anal. Mach. Intell. 8 (6) (1986) 679–698.
[11] V.S. Nalwa, T.O. Binford, On detecting edges, IEEE Trans. Pattern Anal. Mach. Intell. 8 (6) (1986) 699–714.
[12] E. Deriche, Using Canny's criteria to derive a recursively implemented optimal edge detector, Int. J. Comput. Vis. 1 (2) (1987) 167–187.
[13] S.M. Smith, J.M. Brady, SUSAN—a new approach to low level image processing, Int. J. Comput. Vis. 23 (1) (1997) 4578.
[14] P. Meer, B. Georgescu, Edge detection with embedded confidence, IEEE Trans. Pattern Anal. Mach. Intell. (PAMI) 23 (12) (2001) 1351–1365.
[15] C. Topal, C. Akinlar, Edge drawing: a combined real-time edge and segment detector, J. Vis. Commun. Image Represent. 23 (6) (2012) 862–872.
[16] C. Akinlar, C. Topal, EDPF: A real-time parameter-free edge segment detector with a false detection control, Int. J. Pattern Recognit. Artif. Intell. 26 (1). (2012)
[17] S. DiZenzo, A note on the gradient of a multi-image, Comput. Vis. Graph. Image Process. 33 (1986) 116–125.
[18] R.C. Gonzales, R.E. Woods, Digital Image Processing, Pearson Prentice-Hall, 2008, 447–451.
[19] M. Ruzon, C. Tomasi, Color Edge Detection with the Compass operator, IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR) (1999) 160–166.
[20] M. Ruzon, C. Tomasi, Edge, junction, and corner detection using color distributions, IEEE Trans. Pattern Anal. Mach. Intell. 23 (11) (2001) 1281–1295.
[21] A. Saez, C.S. Mendoze, B. Acha, C. Serrano, Development and evaluation of perceptually adapted colour gradients, IET Image Process. 7 (4) (2013) 355–363.
[22] C. Akinlar, C. Topal, ColorED: Color Edge and Segment Detection By Edge Drawing (ED), 2016. submitted for publication
[23] C. Grigorescu, N. Petkov, M.A. Westenberg, Contour detection based on non–classical receptive field inhibition, IEEE Trans. Image Process. 12 (7) (2003) 729–739.
[24] The RUG Dataset Web site, 2016, Accessed: February. http://www.cs.rug.nl/imaging/APD/rug/rug.html.
[25] C. Grigorescu, N. Petkov, M.A. Westenberg, Contour and boundary detection improved by surround suppression of texture edges, J. Image Vision Comput. 22 (8) (2004) 609–622.
[26] G. Papari, P. Campisi, N. Petkov, A. biologically motivated multiresolution approach to contour detection, EURASIP J. Adv. Signal Process. Spec. issue Hum. Percept. (2007)
[27] G. Papari, N. Petkov, Adaptive pseudo dilation for gestalt edge grouping and contour detection, IEEE Trans. Image Process. 17 (10) (2008) 1950–1962.
[28] H. Wei, B. Lang, Q. Zuo, Neurocomputing 103 (2013) 247–262.
[29] G. Azzopardi, N. Petkov, A CORF computational model of a simple cell that relies on LGN input outperforms the Gabor function model, Biol. Cybern. 106 (2012) 177–189.
[30] G. Azzopardi, A. Rodríguez-Sánchez, J. Piater, N. Petkov, A push-pull CORF model of a simple cell with antiphase inhibition improves SNR and contour detection, 9 (7) (2014) e98424. pLoS ONE.
[31] P. Felzenszwalb, D. McAllester, A Min-Cover Approach for Finding Salient Curves, IEEE Comput. Vis. Pattern Recogn. Workshop (CVPRW) (2006)
[32] P. Dollar, Z. Tu, S. Belongie, Supervised learning of edges and object boundaries, IEEE Comput. Vis. Pattern Recogn. (CVPR) (2006) 1964–1971.
[33] P. Dollar, C.L. Zitnick, Structured forests for fast edge detection, Int. Conf. Comput. Vis. (ICCV) (2013) 1841–1848.
[34] Joseph J. Lim, C.L. Zitnick, Piotr dollar, IEEE Comput. Vis. Pattern Recogn. (CVPR) (2013) 3158–3165.
[35] M. Maire, S.X. Yu, P. Perona, Reconstructive sparse code transfer for contour detection and semantic labeling, Asian Conf. Comput. Vis. (ACCV) - Lect. Notes Comput. Sci 9006 (2014) 273–287.
[36] P. Arbelaez, Boundary extraction in natural images using ultrametric contour maps, IEEE Comput. Vis. Pattern Recognit. Workshop (CVPRW) (2006)
[37] 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 Int. Conf. Comput. Vis. (ICCV) (2001) 416–423.
[38] The Berkeley Segmentation Dataset Benchmark Web site, 2016, Accessed: January. http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds.
[39] X. Ren, Multi-scale improves boundary detection in natural images, Eur. Conf. Comput. Vis. (ECCV) - Lect. Notes Comput. Sci 5304 (2008) 533–545.
[40] M. Maire, P. Arbelaez, C. Fowlkes, J. Malik, Using contours to detect and localize junctions in natural images, IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR) (2008) 1–8.
[41] P. Arbelaez, M. Maire, C. Fowlkes, J. Malik, Contour detection and hierarchical image segmentation, IEEE Trans. Pattern Anal. Mach. Intell. (PAMI) 33 (5) (2011) 898–916.
[42] X. Ren, L. Bo, Discriminatively trained sparse code gradients for contour detection, Adv. Neural Inf. Proces. Syst. (NIPS) (2012)
[43] Sparse Code Gradients (scg) Web site, 2016, Accessed: January. http://homes.cs.washington.edu/xren/sparse_contour_gradients_v1.1.zip.
[44] B. Catanzaro, B.Y. Su, N. Sundaram, Y. Lee, M. Murphy, K. Keutzer, Efficient, high-quality image contour detection, IEEE Int. Conf. Comput. Vis. (ICCV) (2009)
[45] C.U.D.A. Nvidia, 2016, Accessed: February. http://nvidia.com/cuda.
[46] N. Silberman, D. Hoiem, P. Kohli, R. Fergus, Indoor segmentation and support inference from RGBD images, Eur. Conf. Comput. Vis. (ECCV) (2012) 564–571.
[47] S. Gupta, R. Girshick, P. Arbeláez, J. Malik, Learning rich features from RGB-d images for object detection and segmentation, Eur. Conf. Comput. Vis. (ECCV) - Lect. Notes Comput. Sci 8695 (2014) 345–360.
[48] G. Papari, N. Petkov, Edge and line oriented contour detection: state of the art, Image Vis. Comput. 29 (2-3) (2011) 79–103.
[49] F. Bergholm, Focusing, edge, IEEE Trans. Pattern Anal. Mach. Intell. (PAMI) 9 (6) (1987) 726741.
[50] A. Goshtasby, On edge focusing, Image Vis. Comput. 12 (4) (1994) 247256.
[51] K. McLaren, The development of the CIE 1976 (L*a*b*) uniform colour-space and colour-difference formula, J. Soc. Dye. Colour. 92 (9) (1976) 338–341.
[52] Color EDContours (CEDContours) Web site, 2016, Accessed: February. http://ceng.anadolu.edu.tr/cv/CEDContours.