Improving the Canny Edge Detector Using Automatic Programming: Improving Hysteresis Thresholding

Lars Vidar Magnusson[1] and Roland Olsson[2]

[1]Østfold University College , lars.v.magnusson@hiof.no
[2]Østfold University College , roland.olsson@hiof.no

Abstract

We have used automatic programming to improve the hysteresis thresholding stage of the popular Canny edge detector—without increasing the computational complexity or adding extra information. The F-measure has been increased by 1.8% on a test set of natural images, and a paired student-t test and a Wilcoxon signed rank test show that the improvement is statistically significant.

This is the first time evolutionary computation and automatic programming has been used to improve hysteresis thresholding. The new program has introduced complex recursive patterns that make the algorithm perform better with weak edges and retain more detail.

The findings provide further evidence that an automatically designed algorithm can outperform manually created algorithms on low level image analysis problems, and that automatic programming is ideally suited for inferring suitable heuristics for such problems.

1 Introduction

Edge detection is a fundamental image processing step, and there are a number of algorithms available that attempt to solve the problem. The Canny edge detector [3] is a popular edge detector available in most image analysis platforms. The algorithm offers great runtime performance, but it is inferior to more complex algorithms [1, 19] in terms of accuracy. Textured regions are typical problem areas, where either edges are found within a texture or missed between two textures.

There have been proposed several improvements to the Canny edge detector. *Surround suppression* was added to improve the performance in textured regions [5]. An adaptive filter has been proposed that could be used in the first stage of the algorithm [18], and there has been proposed an improvement to *nonmax suppression*, the second part of the algorithm [4]. Common for all the proposed improvements are that extra processing is required.

Automatic programming is a machine learning technique related to *Inductive Logic Programming* (ILP) and *Genetic Programming* (GP). *Automatic Design of Algorithms*

This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

Through Evolution (ADATE) [14, 15] is a state-of-the-art automatic programming system that employs an evolutionary computation strategy, in the form of a systematic search process, to create new programs or improve existing ones. The system has been used to infer novel heuristics for image analysis algorithms [12, 13, 11, 7, 2] and other complex algorithms [9, 16] in the past.

We have employed automatic programming and the ADATE system in order to improve *hysteresis thresholding*, the third stage of the Canny edge detector. Others have used evolutionary computation to improve the filter used in the first stage of the algorithm [6], and it has been used to generate logic to identify edges in sliding windows [17, 20]. This is the first time the strategy has been used to improve hysteresis thresholding. We have successfully used a similar approach to improve other aspects of the Canny algorithm [12, 13] already, but improving hysteresis thresholding is more challenging.

The remainder of this paper is structured as follows. Section 2 gives an introduction to the inner workings of the ADATE system, and Section 3 describe the relevant parts of the Canny edge detector. Section 4 provide the details relating to how we conducted the experiments. Section 5 presents the results of the experiments, and Section 6 contains the conclusions that can be drawn from our findings.

2 ADATE

ADATE is a general system for evolutionary computation that can be used both to create new algorithms and to improve existing algorithms. The algorithms are synthesized using a minimal purely functional subset of Standard ML called ADATE ML. The target algorithm is specified in a so called *specification* file, along with any custom data types and auxiliary functions needed. The specification file also contains the fitness function and the data needed for training and validating the performance of the generated programs.

Unlike most other evolutionary computation strategies, ADATE does not rely on random mutations. Instead, the system employs a systematic search inspired by natural evolution. A full explanation of the process is beyond the scope of this article, but a full description can be found in [14, 15].

Population Management

Each separate run with ADATE starts with just one program, the starting program. As new individuals are created they are organized in a tree structure called a *kingdom*, inspired by Linnean taxonomy [8]. At the top level of the kingdom there are *families* of *genera* that are similar in terms of syntactic complexity, and each *genus* contains one or more potential parent programs. A *species* is all individuals created from the same parent program.

Each program has an associated *cost limit* that is essential to the search process. On each iteration, the program with the lowest syntactic complexity and the lowest cost limit is selected for expansion. A number of new programs are synthesized based on the current cost limit and the syntactic complexity of the program. Any new program that is considered better than the existing individuals is inserted into the kingdom, and all the programs that are bigger, but not better, are removed. This produces genealogical chains of gradually bigger and better programs. The cost limit of the parent program is then doubled before continuing by finding a new program to expand. In this way we end up with an iterative deepening search process.

Program Transformations

At each iteration during evolution, a number of new programs are synthesized using a sensible sequence of atomic transformations, referred to as *compound transformations*. The transformations in a sequence are found by systematically searching the program space given by type correct transformations.

The are the following four kinds of atomic transformations. A *Replacement* (**R**) is responsible for exchanging an expression with a new one. The new expression can either be entirely new, or it can include the original expression, or parts of it, as an argument. Replacements are the only transformation type capable of changing the semantics of a program, and as such they are essential to the search process. They are also by far the most combinatorially challenging.

The remaining three transformations are all just syntactic rewrites, but they play an important part for the overall search process. An *Abstraction* (**ABSTR**) is responsible for creating new auxiliary functions by moving a chosen expression into a new function and placing a function call with the appropriate arguments at the location of the source expression. A *Case-Distribution* (**CASE-DIST**) changes the scope of variables and functions, and an *Embedding* (**EMB**) changes the domain or range of an auxiliary function.

3 The Canny Edge Detector

The Canny edge detector [3] is a filter based edge detector in that it convolves the input image with a custom filter to produce a gradient magnitude image. This image provides the input to the second stage of the algorithm, called *non-max suppression*, which was designed to reduce multiple responses to a single edge. This stage of the algorithm has already been improved [12], but we have used the original in this paper. Hysteresis thresholding is the last stage of the algorithm, and it is the target of our investigation in this paper. It was designed to identify the *true* edges amongst all the *weak* edges, and as such reduce the number of false positives.

There are many implementations of the Canny algorithm available, but we decided to base our experiments on the implementation in Matlab. It offers a good match to the original description of the algorithm, and the implementation has been widely used in academia. The entire algorithm has been ported to SML and tested thoroughly to ensure that the implementation produces the same results as the original.

Hysteresis Thresholding

In addition to the non-max suppressed gradient magnitudes image, hysteresis thresholding takes two thresholds as input. The two thresholds *high* and *low* are used to identify the strong and weak edges respectively. All gradient magnitudes greater than the high threshold are considered strong edges, and the ones greater than the low threshold are considered weak edges. The final edge map contains all the strong edges, along with all weak edges that are connected to a strong edge.

In the Matlab implementation, hysteresis thresholding is done using vector operations not available in SML. The algorithm has instead been implemented as a recursive depth-first-search from all strong edges to find all the weak edges connected. The runtime for this implementation is linear in the number of pixels in the image.

4 Experimental Setup

This section will cover the most important aspects of the experimental setup. The ADATE specification and all relevant resources are available at our website [10].

The Original Program

```
fun f( g, e, et, h, l, p, m )
let
  fun follow( pf, us ) =
  let
    fun filterValid ns =
      case ns of
        [] => []
      | n::ns' =>
          case nav( g, pf, n ) of
            invalid => filterValid ns'
          | valid pv => pv::filterValid ns'
  in
    follow'(
      filterValid
        [ up, upRight, right, downRight, down, downLeft, left, upLeft ],
      us )
  end

  and follow'( ps, us ) =
    case ps of
      [] => us
    | p::ps' =>
        case checkUpdate( g, e, et, p, l, us ) of ( u, us' ) =>
        case u of
          false => follow'( ps', us )
        | true => follow'( ps', follow( p, us' ) )

  val ( u, us ) = checkUpdate( g, e, et, p, h, [] )
in
  case u of
    false => []
  | true => follow( p, us )
end
```

Figure 1: The original program written in Standard ML.

The original program can be seen in Figure 1. The first three input arguments are not intended for direct use, but they are needed by some the auxiliary functions. *g* is a representation of the gradient magnitude image, and *e* and *et* are the final edge map and a temporary edge map respectively. The temporary edge map is used to keep track of edges found during the current invocation. The input arguments *h* and *l* contain the high and low thresholds respectively, while *p* and *m* contain the current position and magnitude. The function returns a list of positions to update.

There are two auxiliary functions defined outside the learning environment. The first, *checkUpdate*, is used to check if the magnitude of a specified position is greater than a specified value, and to mark the position as an edge if that is so. The second auxiliary function, *nav*, is used to navigate the image. It takes a position and a direction to move, and it returns either *invalid* or the valid position.

Figure 2: A selection of images from the BSDS500 [1]

The function f is invoked for all pixels in the image not already marked as an edge. Each invocation starts with a call to *checkUpdate* to compare the magnitude of the current position to h. If the magnitude is greater, the auxiliary function *follow* is invoked, if not, the function simply returns. The *follow* function takes a position *pf* and generates a list of all the valid neighbors and invokes the *follow'* function on that list. The *follow'* function compares the magnitude of each of the positions in the list against l using *checkUpdate*, and invokes *follow* on the positions where the magnitude is greater.

The Dataset

The training data used for our experiments were taken from BSDS500 [1]—a dataset specifically designed to train and evaluate contour detectors and image segmentation algorithms. The set contains high quality ground truth annotations by on average five subjects for each of the images. There are in total 500 images in the set; 200 training images, 100 validation images and 200 test images. Figure 2 shows a few example images from the dataset.

We used the training and validation images both for finding appropriate constants and for evolving new programs. In order to reduce the time needed to evaluate each image during evolution, we decided to reduce the cardinality of the ground truth sets by only using the *best* ground truth from each set. The best ground truth was selected by evaluating each truth against the others in the set. This reduces the time needed to evaluate each image to about one fifth of what is originally needed, and previous experience with the dataset has shown that this approach is capable of producing results that correlate with the results from a full evaluation.

The Fitness Function

We have used the average F-measure as the performance metric in our experiments. We evaluate the quality of an edge map using the exact same approach as [1]. A graph matching algorithm is used to match edges in an edge map against edges in the ground truth. But, where [1] accumulate the count for all the images and calculate the F-measure for the entire set, we instead calculate the F-measure on each image, and use the average to determine the overall performance on the set. This approach makes each image equally important to the final score, regardless of the number of edge pixels in the image.

Selecting the Constant Values

The Canny algorithm has three constants that control the algorithm. The first, σ, specifies the standard deviation of the noise reduction filter used in the first stage of the algorithm. The last two are the *high* and *low* thresholds used in hysteresis thresholding.

The first constant σ was chosen by simple grid search. We evaluate values for σ in the interval $[0.25, 4.25]$ using a small 4×4 grid for the thresholds. The two thresholds were found using the best σ and a 32×32 grid. Initial experiments showed that using thresholds optimized for the entire set produced programs that used only one of the thresholds, meaning that it is unlikely that any benefit is gained from using a dual threshold setup when the thresholds have been tuned for the entire set. Therefore we decided instead to use constants optimized per image.

5 Results

This section presents the new and improved program along with the results of the benchmarks, and a visual inspection of the improvement on a small set of images.

The Improved Program

The improved program is listed in Figure 3. There are a number of changes, but most of the overall structure from the original program remains.

The first difference is the comparison of h to m. If h is greater or equal to m the function returns an empty update list. If h is less than m, the function makes a recursive call with $h = l$, $l = m$ and $m = h$. The program then continues to invoke *checkUpdate* to check the magnitude at position p against $h \tanh h$, and to update the edge maps if necessary. Note that the function call is sent the update list returned from the recursive call, and that *et* has been replaced by *e*.

The *follow* function has been modified so that it invokes *follow'* two times. The first invocation will check only the current position, and the second will check the neighbors of the current position, in the following order; *downRight*, *upRight*, *right*, *downLeft*, *down*, *upLeft*, *left*, *upLeft* and *up*. Note that *upLeft* is repeated twice. The internal auxiliary function *filterValid'* has changed so that it stops if any of the neighbors are invalid.

The *follow'* function has also changed. The function now uses the update list sent to *follow* (*us*)—instead of the update list in us'—as an argument in the function call to *checkUpdate*. A new case-expression has been inserted where *u* matches *false*. This invokes *nav* in an attempt to return the *downLeft* neighbor. If this neighbor is invalid, the function returns *us*, if it is valid, normal operation is resumed. Lastly, the function has been modified where *u* matches *true*. The order of the function calls has changed so that *follow'* is invoked first, and the *pos* list has changed to *ps*—the position list originally sent to *follow'*.

Benchmarks

The final benchmark of the accuracy of the algorithms were conducted using the accumulated F-measure and the full ground truth sets as in [1], and we used the dedicated test set with 200 images. This allows fair comparison between the algorithms, and it makes the result comparable with the result of a number of other algorithms that have been benchmarked using the same setup.

The benchmark results can be seen in Table 1. We have evaluated the algorithms using constants optimized for the entire dataset (**OD**) and constants optimized per image

```
fun f( g, e, et, h, l, p, m ) =
let
  fun follow( fp, us ) =
  let
    fun filterValid' ns =
      case ns of
        [] => []
      | n::ns' =>
      case nav( g, fp, n ) of
        invalid => []
      | valid pv => pv::filterValid' ns'

    fun follow'( ps, us' ) =
      case ps of
        [] => us'
      | p::ps' =>
      case checkUpdate( g, e, et, p, l, us ) of ( u, us'' ) =>
      case u of
        false => (
          case nav( g, p, downLeft ) of
            invalid => us
          | valid _ => follow'( ps', us' ) )
      | true => follow( p, follow'( ps, us'' ) )
  in
    follow'(
      filterValid'
        [ downRight, upRight, right, downLeft,
          down, upLeft, left, upLeft, up ] ,
      follow'( [ p ], us ) )
  end
in
  case realLess( h, m ) of
    false => []
  | true =>
      case f( g, e, et, l, m, p, h ) of us =>
      case checkUpdate( g, e, e, p, h*tanh h, us ) of
        ( _, us' ) => follow( p, us' )
end
```

Figure 3: The improved program written in Standard ML.

(**OI**). The F-measure with **OD** constants is listed under **ODF**, and the F-measure with **OI** constants is listed under **OIF**.

The SCG algorithm [19] has been included for comparison. This is currently the best known algorithm for the dataset, but it is far more complex than the other two algorithms and requires much longer runtimes.

The ADATE-improved edge detector has been improved by 1.8%, or 1.1 percentage point, with OD constants, and by 1.4%, or 0.9 percentage points, using OI constants. This is a significant improvement since it closes 11.5% and 11.9% of the gap up to the SCG algorithm with OD and OI constants respectively—without increasing the computational complexity or adding more information. Note that the improvement is greater when using OD constants—despite the fact that the algorithm was evolved using OI constants. This is a strong indication that the algorithm offers a general improvement over the original Canny algorithm.

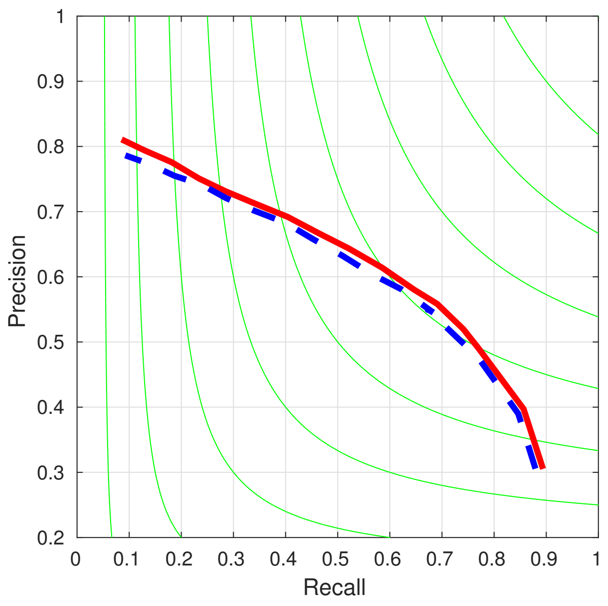

Figure 4: The ROC-curves of both algorithms. The ADATE-improved curve in blue, and the original in dotted blue.

The **ROC**-curves of the two algorithms can be seen in Figure 4; the ADATE-improved algorithm is plotted in red, and the original Canny algorithm in dotted blue. The curves show a comfortable improvement across the entire range.

Table 1: The results of the benchmarks.

	ODF	**OIF**
SCG	0.71	0.73
ADATE-Improved	0.618	0.656
Canny	0.606	0.646

We performed a paired student-t test and a Wilcoxon signed rank test on the results of the two algorithms on the test set using OD constants. The student-t test gave a p-value of 4.685×10^{-09}, and the Wilcoxon signed rank test gave a p-value of 2.357×10^{-10}. Therefore both tests confirm that the improvement made to the algorithm is statistically significant on the test set. We have included a histogram plot of the difference in F-measure between the ADATE-improved and the original Canny algorithm in Figure 5.

Visual Analysis

We have provided a few example images, along with the edge maps produced by the two algorithms, for visual analysis of the improvement. The images and corresponding edge maps can be seen in Figure 6, where the results produced by the ADATE-improved algorithm are in the middle, and the results from the original in the right.

The first image has only a single foreground object, but most of the edge pixels around the duck are missing in the image on the right. The middle image is also missing quite a few edges—in particular the weak edges over the back of the bird—but the image is a significant improvement over the right image. The improved algorithm is able to correctly

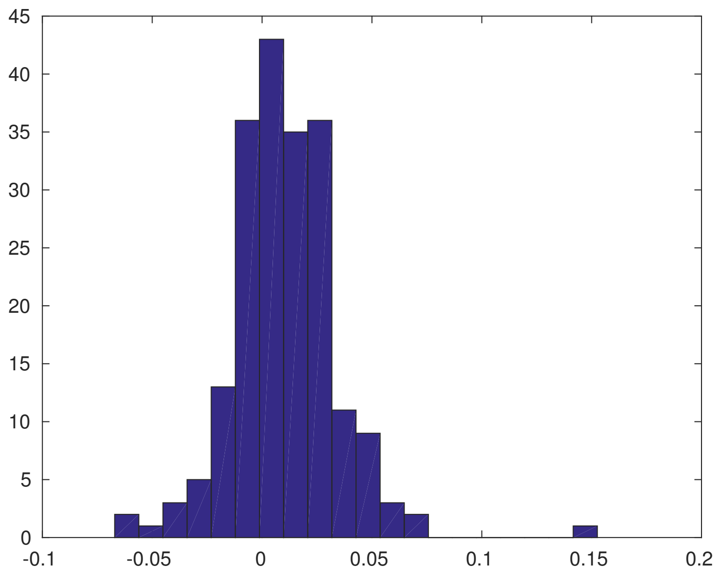

Figure 5: A histogram plot of the difference in F-measure between the two algorithms on the test set.

identify most of the edges around the head, and it has found some of the tail and second leg. There is also more detail in the edge along the reflection in the water.

Both algorithms are capable of identifying the primary object in the second image, but their performance differ in the surrounding areas. The improved algorithm has found more of the cracks in front of the bear, it has been able to detect more of the ice in the water in front, and there is more detail in the background.

In the second image, we can easily see that the improved algorithm does a considerably better job with the emblem on the side of the fuselage. The original algorithm has identified more of the edges around the clouds, but these edges are neglected in all but one of the ground truth images.

The fourth image is an example where a considerable amount of weak edges are marked as true edges. The improved algorithm does a better job at neglecting these edges—both in the fur of the goat and the blurry background.

The improved algorithm has identified more of the left wing of the penguin in the last image than the original, and it has identified the edges around the eye. The rocky region under the penguin is troublesome for both algorithms. The improved looks slightly better on the right hand side of the image, and the original looks better on left.

6 Conclusions

We have successfully improved the hysteresis thresholding stage of the Canny edge detector. The F-measure has increased with 1.8 percent on a widely used test set of natural images. A paired student-t test and a Wilcoxon signed rank test gives p-values of 4.685×10^{-09} and 2.357×10^{-10} respectively, which show that the improvement is extremely statistically significant.

The improved algorithm has introduced several complex recursive patterns that exhibit an increased ability to distinguish true weak edges from false weak edges, and to produce

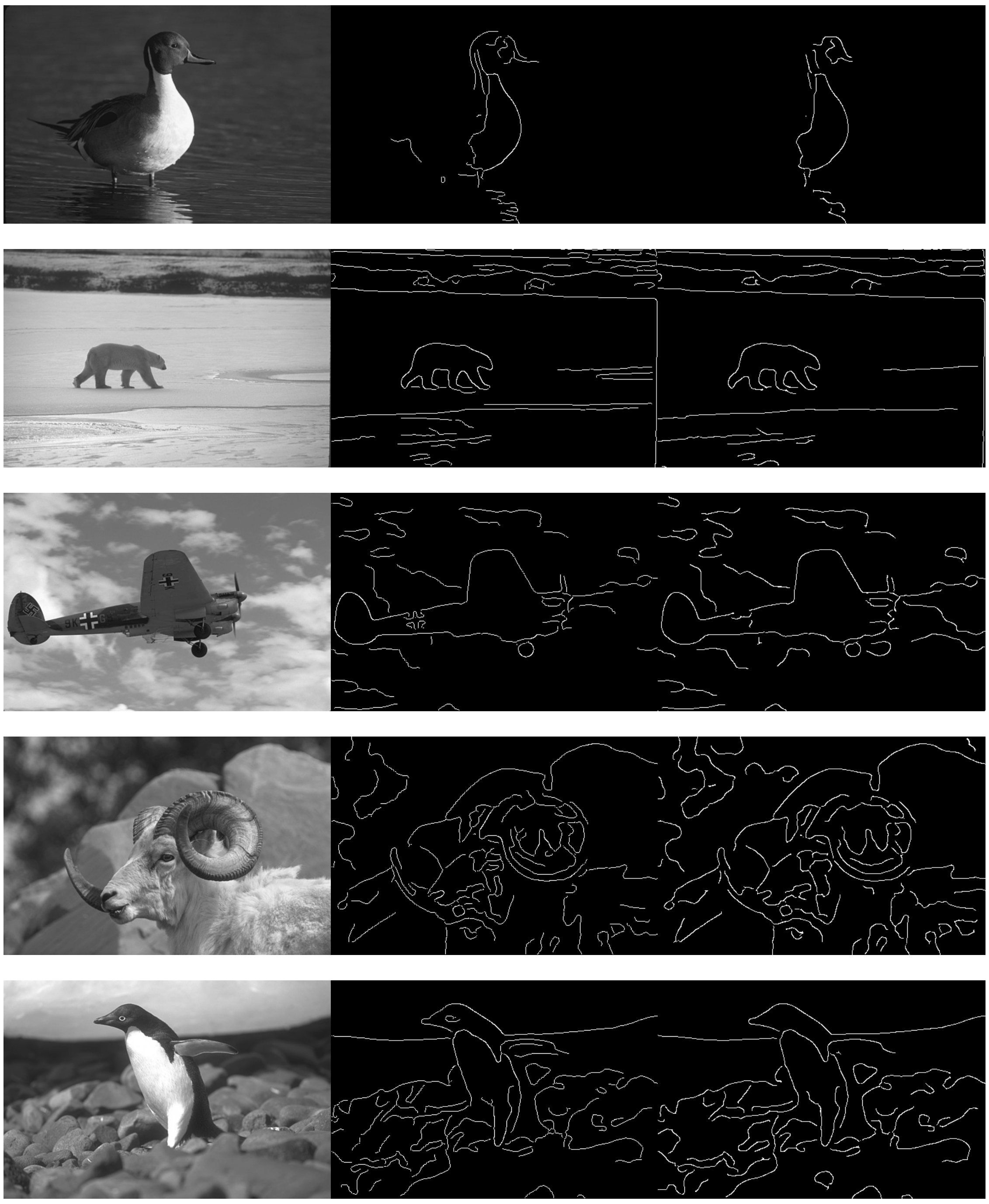
Figure 6: A selection of test images and the results produced by the two algorithms using OD constants.

more accurate representations of edges in detailed areas. This has been achieved without adding any extra processing or information to the algorithm.

The algorithm offers the greatest improvement when using constants optimized for the entire test set, even though the program was evolved using constants optimized per image. This indicates that the algorithm offers a general improvement over the original, but more testing is required on other data sets to make sure.

The fact that we have improved hysteresis thresholding is further evidence that automatic programming can infer heuristic algorithms capable of outperforming popular image analysis algorithms created manually. The approach should work equally well in other problem areas where exact solutions are impractical or impossible.

Future Work

There are a number of promising candidates for future investigations. There has been a number of efforts to find an adaptive threshold mechanism for the Canny edge detector, but a heuristic algorithm created using automatic programming could provide an improvement over the current state-of-art.

We have been successful in improving the stages of the Canny algorithm separately. The separate improvements need to be tested together to figure out how that affects the performance. We would also like to investigate the possibility of evolving the different stages together, which would allow for complex collaborative patterns to be created.

There are several other image analysis algorithms that we would like to try to improve using a approach similar to the one presented in this paper. We have successfully improved several non-trivial image analysis algorithms, but it remains to see if we could improve the state-of-the-art algorithms.

References

[1] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33:898–916, 2011.

[2] H. Berg, R. Olsson, T. Lindblad, and J. Chilo. Automatic design of pulse coupled neurons for image segmentation. *Neurocomputing*, 71(10-12):1980–1993, 2008. Neurocomputing for Vision Research; Advances in Blind Signal Processing.

[3] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):679–698, 1986.

[4] L. Ding and A. Goshtasby. On the canny edge detector. *Pattern Recognition*, 34(3):721–725, 2001.

[5] C. Grigorescu, N. Petkov, and M. A. Westenberg. Contour and boundary detection improved by surround suppression of texture edges. *Image and Vision Computing*, 22(8):609–622, 2004.

[6] C. Harris and B. Buxton. Evolving edge detectors with genetic programming. In *Proceedings of the 1st annual conference on genetic programming*, pages 309–314. MIT Press, 1996.

[7] K. Larsen, L. V. Magnusson, and R. Olsson. Edge pixel classification using automatic programming. *Norsk Informatikkonferanse (NIK)*, 2014.

[8] C. Linnaeus. *Systema naturae per regna tria naturae secundum classes, ordines, genera, species,...*, volume 1. impensis Georg Emanuel Beer, 1788.

[9] A. Løkketangen and R. Olsson. Generating meta-heuristic optimization code using ADATE. *Journal of Heuristics*, 16:911–930, 2010.

[10] L. V. Magnusson. Image analysis & machine learning. `http://www.it.hiof.no/iaml/`. Accessed 2016-08-20.

[11] L. V. Magnusson and R. Olsson. Improving graph-based image segmentation using automatic programming. In *Applications of Evolutionary Computation*, pages 464–475. Springer, 2014.

[12] L. V. Magnusson and R. Olsson. Improving the canny edge detector using automatic programming: Improving non-max suppression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 461–468, New York, NY, USA, 2016. ACM.

[13] L. V. Magnusson and R. Olsson. Improving the canny edge detector using automatic programming: Improving the filter. In *Image, Vision and Computing (ICIVC), International Conference on*, pages 36–40. IEEE, 2016.

[14] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74:55–81, 1995.

[15] R. Olsson. Population management for automatic design of algorithms through evolution. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 592–597. IEEE, 1998.

[16] R. Olsson and A. Løkketangen. Using automatic programming to generate state-of-the-art algorithms for random 3-sat. *Journal of Heuristics*, 19(5):819–844, 2013.

[17] R. Poli. Genetic programming for feature detection and image segmentation. In *Evolutionary Computing*, pages 110–125. Springer, 1996.

[18] B. Wang and S. Fan. An improved canny edge detection algorithm. In *2009 second international workshop on computer science and engineering*, pages 497–500. IEEE, 2009.

[19] R. Xiaofeng and L. Bo. Discriminatively trained sparse code gradients for contour detection. In *Advances in neural information processing systems*, pages 584–592, 2012.

[20] Y. Zhang and P. I. Rockett. Evolving optimal feature extraction using multi-objective genetic programming: a methodology and preliminary study on edge detection. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 795–802. ACM, 2005.