A Relaxation Algorithm for Segmentation of Real-World Scenes

Steve Dropsho

Department of Computer Science

University of Massachusetts-Amherst

Amherst, MA 01003

Introduction

This report is a synthesis of two areas in computer science: machine vision and computer architecture. The first half of the paper explores a new segmentation algorithm while the second half discusses the computational needs of the algorithm and the architectural features that are required to attain the high performance.

Segmentation is a fundamental operation in machine vision. Grouping image data into regions is often a first step in interpreting scenes in an image. A segmentation algorithm is presented here that has three stages. The first stage provides edge detection through a Hopfield neural network. The second stage is an edge extension phase to complete the boundaries indicated by the first stage. The last stage performs region merging in a novel approach that combines texture classification of regions with multiple merging policies. This combination proves especially effective on natural scene images.

In the latter half of the paper the computational needs of the algorithm are detailed. The algorithm is designed to have a very high degree of data parallelism. This makes the algorithm ideally suited for a SIMD machine and it has been ported to the University of Massachuesetts' Image Understanding Architecture (IUA). Performance results are given along with a discussion of the bottlenecks in the code and architectural features required to alleviate them.

Segmentation

Overview

The core of the segmentation algorithm is a slight modification of the technique in [Cortes, 1989 #1] which uses a relaxation algorithm to separately detect horizontal and vertical edges between pixels. The horizontal and vertical results are later combined to suggest likely regions. Originally tested on synthetic images, the algorithm as stated is insufficient for real outdoor images, however it does provide a good indicator of region boundaries. Using the output of the above algorithm as a starting point, two additional phases are added to complete the segmentation. These are an edge extension phase and a merging phase based on texture classification of regions. The resulting algorithm goes well beyond the original and is able to produce good segmentations of real outdoor scenes.

Edge Detection

In [Cortes, 1989 #1], Cortes and Hertz explored a number of parallel networks for edge detection. They detail the design of the simplest system that they found to work on synthetic images which are blurred and corrupted with gaussian noise. The architecture has two Hopfield neural networks that use the directional second derivative to detect edges in the image, both horizontal and vertical; the networks differ only in the direction that the second derivative is taken. The results are combined in a final network to generate the final segmentation of the synthetic images (fig. 1).

Each network contains image units and edge units (fig. 2). Both types of units can take values +/-1. The image units, denoted S_i, are in one-to-one correspondence with the pixels in the image; i.e., unit S_i maps to pixel i The edge units, denoted $A_{i,j}$, are between adjacent image units i, j and are set to identify discontinuities in the image. Edge unit $A_{i,j}$ is set to +1 if an edge is hypothesized between image units i and j.

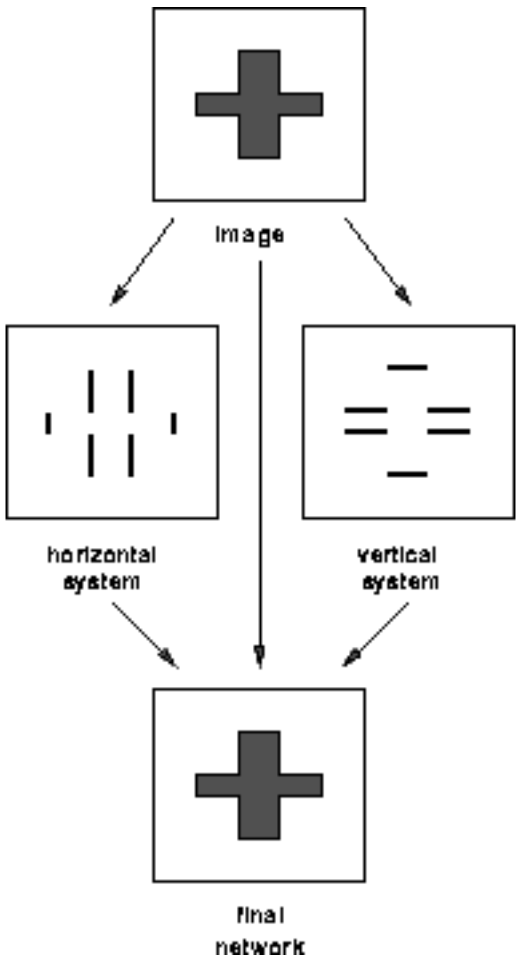


Figure 1: Network Architecture.

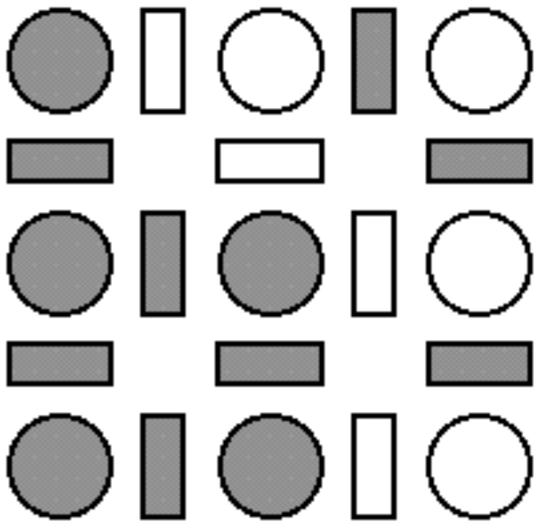


Figure 2: The organization of image units (circles) and edge units (bars).

Horizontal and Vertical Edge Detection

Cortes et al. use the horizontal and vertical edge networks to minimize the energy equation (1). In this phase, h_i^{ext} is not the image itself but, rather, the second derivative of the image in the appropriate direction. A qualitative description of equation (1) is as follows. Since both S_i and $A_{i,j}$ can only take values +/- 1, the second sum is minimized when S_i matches the sign of the second derivative h_i^{ext} at pixel i. In the first sum, the pixels i and j straddling a zero crossing of the second derivative will have values h_i^{ext} and h_j^{ext} of opposite signs so S_i and S_j will have also have opposite signs. Thus, the first sum is minimized if the barrier between the two pixels is set to $A_{i,j} = +1$ at zero crossings and $A_{i,j} = -1$ otherwise.

$$\mathbf{H} = -{}^{1}/_{2} \sum_{<i,j>} [(1 - A_{i,j})/2] S_i S_j - \sum_i S_i h_i^{ext} \qquad (1)$$

The authors describe two methods for numerically solving for the values of image and edge units. The first is simulated annealing where, the value of a unit is randomly selected to be +1 with the probability defined by (2) for image units and -1 otherwise. The probability for edge units is similar. The factor h_i weights the probability to greater than 0.50 that the unit will have the same sign as h_i. The factor B is the inverse ``temperature'' that is slowly lowered to allow exploration for a global optimum state. In the annealing process the units flip between +/- 1 with the actual value being the average over a period of time at the final temperature.

$$P(S_i(t+1) = +1 | h_i(t)) = \exp(B\ h_i(t)) / [\exp(B\ h_i(t)) + \exp(-B\ h_i(t))] \qquad (2)$$

The definition of h_i is shown in (3). It is simply the negative of the partial derivative of (1) with respect to image unit S_i and indicates the direction the value of S_i should be adjusted at time $t+1$ to not increase the energy function given the values of the network at time t. Since a unit is restricted to only two values (+/- 1), the value probabilistically toggles such that the average over a period of time is the final value. This method of stochastically setting the value of the units allows the network to explore the solution space using only local information for a highly parallel algorithm. Equation (4) is the corresponding update equation for the edge units.

$$h_i = \sum_j [(1 - A_{i,j})/2] S_j + h_i^{ext} \qquad (3)$$

$$h_{ij} = -{}^{1}/_{2} S_i S_j \qquad 4)$$

The overall function of the network is to smooth the directional second derivative of the image but not across strong boundaries. Because the network uses the directional second derivative, it is not sensitive to absolute intensity values or constant gradients in the image. The output of the network is the value of the edge units that are used as boundary hints in the final network. Figure 3 shows the state of the edge units and image units for an example image. Notice the spurious edge marker in the middle. This is due to the propagation of the negative values of the image units from the left and the positive values from the right. It is the job of the final network to identify and remove these spurious markers.

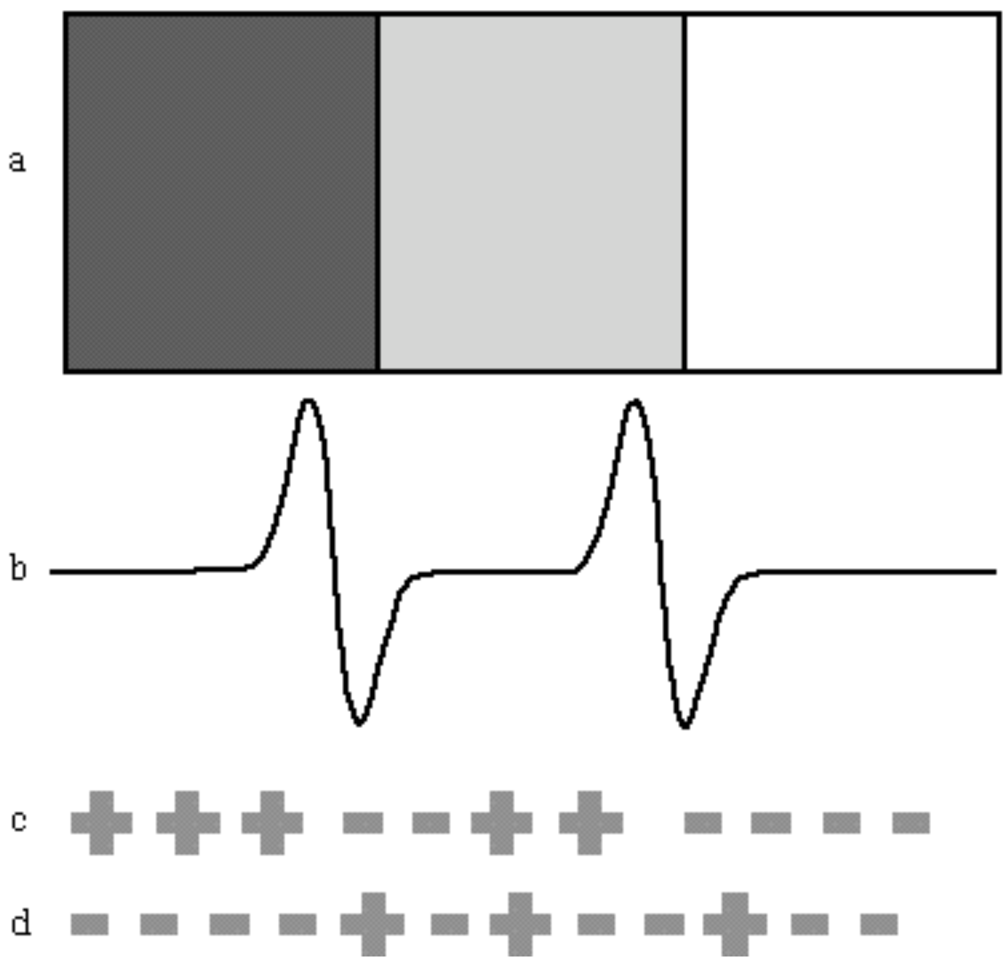


Figure 3: (a) The input image. (b) The second derivative in the horizontal direction. (c) The values of the image units. (d) The values of the edge units.

The second method of solving the energy function, called mean field annealing, replaces the stochastic binary variables S_i, S_j, and $A_{i,j}$ in (3) and (4) with their averages denoted by $<S_i>$, $<S_j>$, and $<A_{i,j}>$. Then from (3) and (4)

$$<S_i> = P(S_i = 1) - P(S_i = -1) = \tanh(B h_i) =$$

$$\tanh[B(\textstyle\sum_j [(1 - <A_{i,j}>)/2]<S_j> + h_i^{ext})] \qquad (5)$$

and

$$<A_{i,j}> = \tanh(B h_{i,j}) = \tanh({}^1/_2 B<S_i><S_j>) \qquad (6)$$

Cortes et al. show that mean field annealing produces comparable results to conventional simulated annealing, but in less time. They also note that although mean field annealing is an approximation for the stochastic network it is the exact theory for the equivalent analog network. They briefly outline a possible VLSI implementation with a tunable gain B as the temperature control.

The Final Network

The final network has the same structure as the previous networks. One difference is that the image units Ω are now *linear* units, the output is not limited by the sigmoid function as in the preceding stage. Equation (7) is the energy function from [Cortes, 1989 #1] that is minimized. The factor μ_{ij} represents the resulting edge unit values from the horizontal and vertical networks. The factor h_i^{ext} is now the original intensity image itself.

$$\mathbf{H = +{}^1/_2 \sum_{<i,j>} [(1 - A\{i,j\})/2](\Omega_i - \Omega_j)^2 + {}^1/_2K \sum_i (\Omega_i - h_i^{ext})^2 - \sum_{<i,j>} \mu_{ij}\ A\_\{ij\}} \qquad \mathbf{(7)}$$

This energy function is easy to interpret. The last term rewards placing boundaries at edge units where the previous networks placed them and inhibits introducing new boundaries. The first term encourages placing a boundary marker between two image units that differ significantly in intensity value. The middle term encourages the network to set the intensity values close to the input image. If the coupling constant K is large then the intensities will tend to track those of the original image. If it is small then the first term will gain influence and the image units will tend to average their values with their neighbors that are not separated by edge units.

For completeness, The equations for h_i and h_{ij} are shown in (8) and (9). They are the negative of the partial derivatives of the energy equation (10).

$$\mathbf{h_{ij} = (\Omega_i - \Omega_j)^2 + \mu_{ij}} \qquad \mathbf{(8)}$$

$$\mathbf{h_i = \sum_{<i,j>} [(1 - A\{i,j\})/2](\Omega_j - \Omega_i) + K(h_i^{ext} - \Omega_i)} \qquad \mathbf{(9)}$$

Observations and Contributions

The above algorithm is the complete segmentation algorithm reported in the paper by Cortes. The following are observations from experimenting with that algorithm.

Reproduction of Results. Using the information from the paper I was not able to reproduce quality segmentations of images similar to their test images. Their test images were synthetic images of rectangular regions labeled with one of three intensities 0, 1, 2. Gaussian noise and blur were added to the images to increase the difficulty of the segmentation. The algorithm I reproduced from the paper did not produce good segmentations of the synthetic images. The problem is that the algorithm is not guaranteed to completely close the boundary around a region. There were many instances where the regions were outlined correctly except between one or two pairs of pixels. As a result, the regions were incorrectly joined. This behavior is not surprising since the alogithm has only local information at each pixel to use. For the results in the paper, one possible explanation that is strictly from conjecture is that since the test images in the original study had only three intensity values the pixels may have been set to the closest value of the three at the end of the algorithm allowing the easy grouping of pixels for likely regions.

Scaling. The authors do not mention how the terms in their algorithm scale for images with intensities with a range greater than 0 to 2. From experimentation, the best results occur when the intensity range is of the same order of magnitude as the range for *tanh(h_i)* which has the range of -1.0 to 1.0. I use a range of 0.0 to 4.0. The minimum and maximum values of the input image are automatically detected and mapped to 0.0 and 4.0 respectively. All values in-between are scaled linearly within the range.

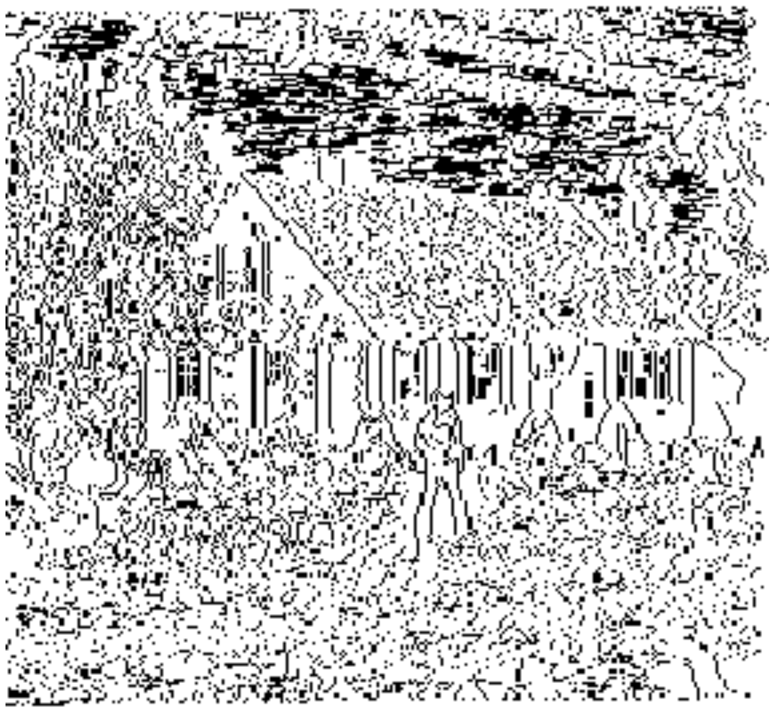
The Elimination of Spurious Edges. Cortes et al. assert that (7) removes spurious edge markers. I found no support for this claim. To demonstrate this, the results of the first and second network are examined for the vertical edges only. The top image in figure 4 is the output from the first network that finds potential vertical edges. Due to printing limitations a whole pixel is dark if the vertical edge on the left side of the pixel is selected. As you can see, the first network generates a significant number of potential edges. The second image in figure 4 is the result from the second network using Equation (8) from the paper. This network does not eliminate any of the potential edges and actually adds additional edges. Looking more closely at the energy function one can see why. A spurious edge marker will have a positive value for μ_{ij} so the third term of the energy function is minimized by placing an edge marker in the final network at every position where there is a potential edge from the first network. The first term does not penalize this action as claimed by the authors. The first term is always positive; the smallest value it can take is zero. Thus, the energy equation is actually minimized if all potential edges are maintained. The number of edges can actually increase due to the first term adding edges at gradients ignored by the first network.

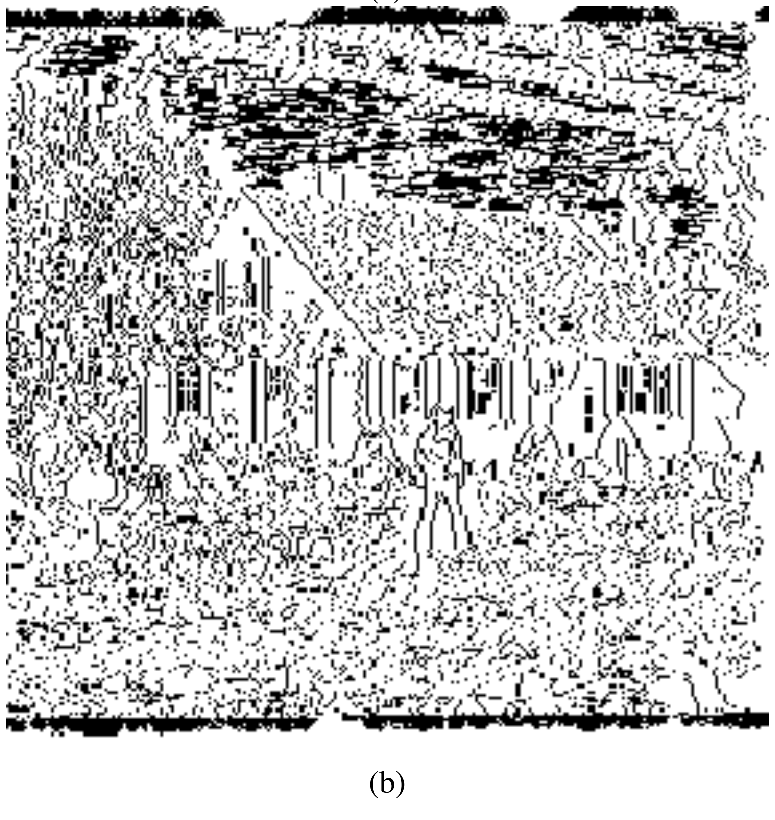
The goal is to have an energy function that maintains the edge markers from the input networks except where the intensity gradient is ``small'' to sift out spurious markers. A test has to be included in the energy function to quantify what constitutes a ``small'' gradient. Adding a threshold value *T* as in (10) performs this function.

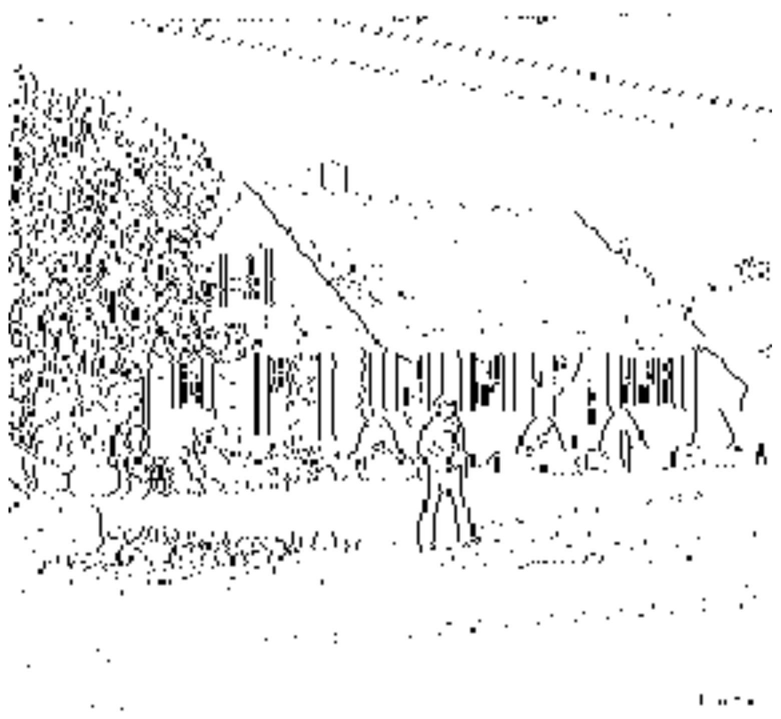
$$\mathbf{H = +{}^{1}/_{2} \sum_{<i,j>} [(1 - A_{i,j})/2][(\Omega_i - \Omega_j)^2 - T] + {}^{1}/_{2}K \sum_i (\Omega_i - h_i^{ext})^2 - \sum_{<i,j>} \mu_{ij} A_{ij} \quad (10)}$$

Unfortunately, *T* is another arbitrary parameter and is dependent on the image. From empirical experimentation it was found that the values for *T* that performed well were in a tight range of (1.07, 1.10). In general, smaller values produced over-segmentation and higher values produced significant under-segmentation. Interestingly enough, the 'value' that performed the best was actually a graduated scale with the highest intensity regions using a T=1.09 and the lowest intensity regions using T=1.18 and interpolated values for all intensities in between. The intuition being that we have keen interest in the brighter regions of the image and but less in the dark regions. This is strictly a heuristical rule, but it helps to minimize the number of regions in the final segmentation with minimal effects on the final segmentation across the images tested. The third image in figure 4 is the output of the second network using the energy function in (10). Notice that there are many fewer edges. Many of the missing edges along borders such as the roof line would be found in the horizontal edge representation.

As an implementation note, the value of *T* must be large enough to discourage spurious edge markers from being added by the third term. Thus, a value near 1.0 is large enough to offset the third term. An additional amount defines what is a ``small'' intensity gradient. However, the actual value relies on the two factors as well as the intensity range of the image which is 0.0 to 4.0 in this implementation. The algorithm seems robust enough that binning the graduated scale of *T* into only four values does not affect the results and replaces the interpolation calculation with a small table lookup. Figure 5 is the original image and figure 6 is the result from the final network showing the placement of horizontal and vertical edge markers. Due to printing restrictions the state of the edge markers are shown via the image units. An image unit is highlighted if either the vertical edge marker to its left is set or the horizontal marker above is set.


(a)


(b)



(c)

Figure 4: (a) Result from first network for potential vertical edges. (b) Resulting vertical edges from second network using original equations. (c) Resulting vertical edges from second network using modified equations.

A Mathematical Simplification. To simplify the calculation of h_{ij} in the final network the absolute value is used instead of the square of the difference of Ω_i and Ω_j. In addition, the constant that would be associated with the first term in equation (11) is set to 1. The values for threshold T stated above account for these adjustments. The resulting equation for h_{ij} h_{ij} is shown by (11). The purpose of these adjustments is to simplify the calculations for a SIMD array of simple PEs to which this algorithm has been ported. As will be discussed in a later section, multiplication is a very time-consuming operation on the particular SIMD machine that is used. Minimizing arithmetical operations such as multiplication and division is critical to achieving high performance.

$$\mathbf{h_{ij} = |\Omega_i - \Omega_j| - T + \mu_{ij}} \qquad \mathbf{(11)}$$

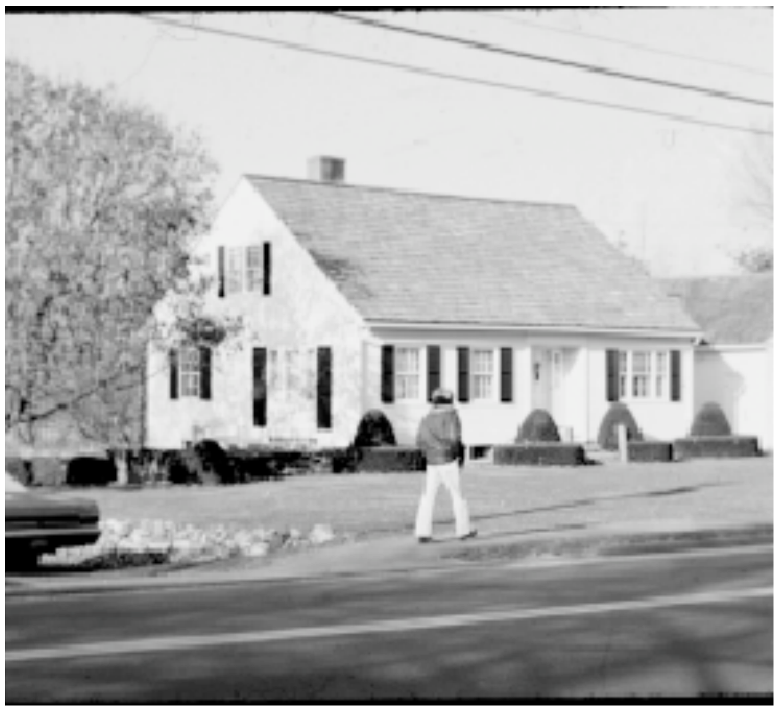


Figure 5: House Image

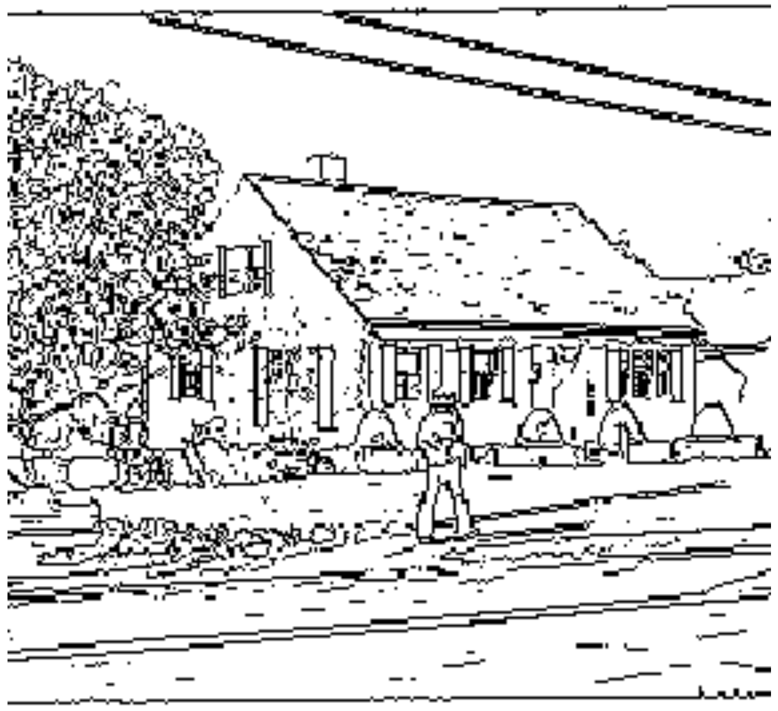


Figure 6: Result from final network

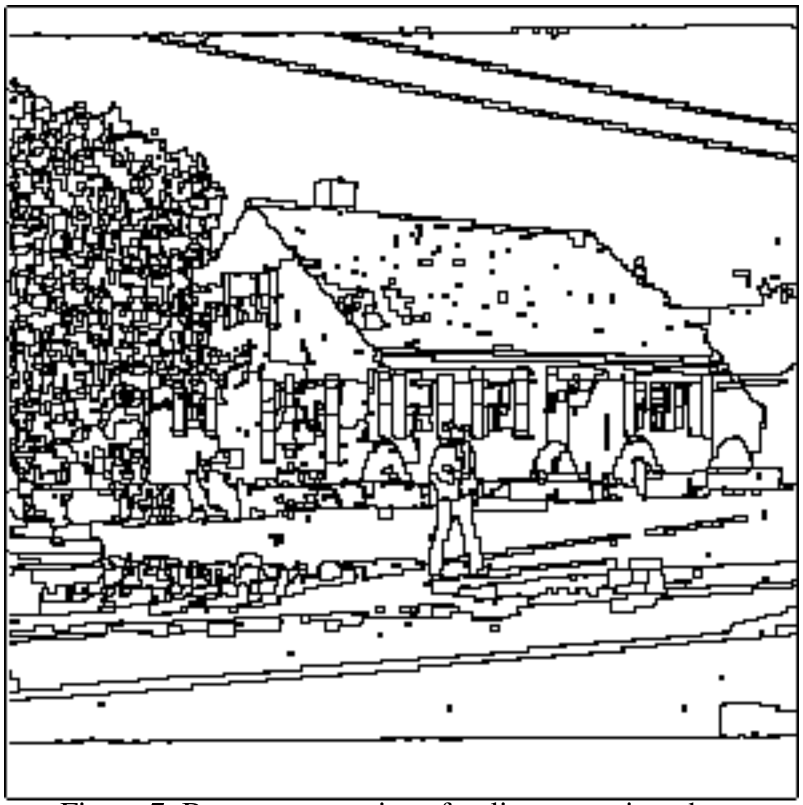


Figure 7: Base segmentation after line extension phase

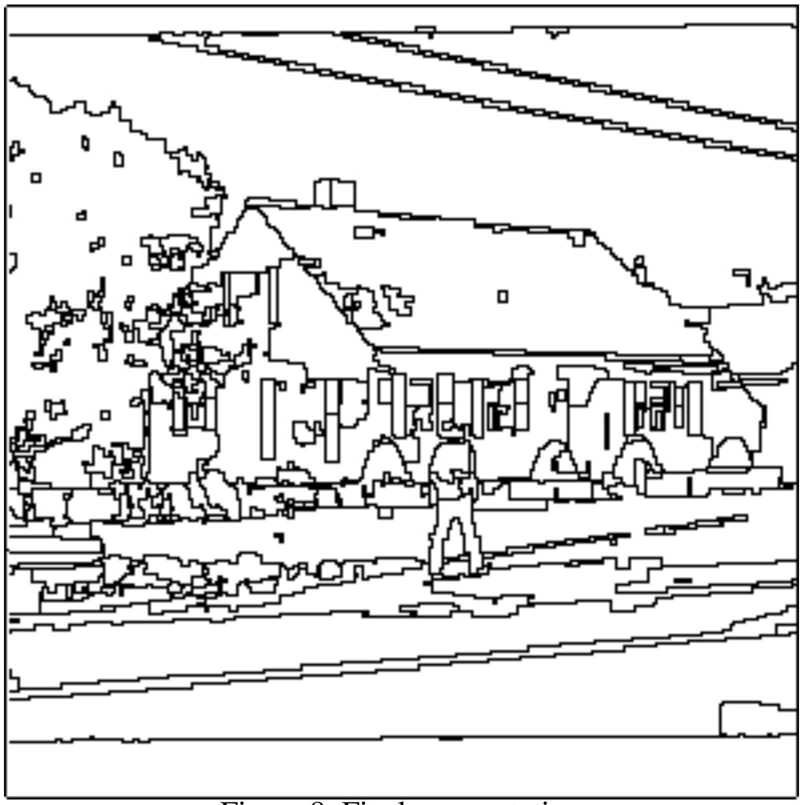


Figure 8: Final segmentation

Edge Extending

As mentioned before, I was not able to reproduce the results of Cortes et al. on the synthetic images. I found that there is no guarantee the final network will completely set all edge markers around the boundary of a region. This problem is especially noticeable when the algorithm is applied to outdoor scene images. To help alleviate this shortcoming a simple edge extension phase has been added after the final network to complete the boundaries around likely regions.

The method is very simple and quite effective. Edge units with an end not connected to another edge unit are called dangling ends. Edge units adjoining dangling ends are set to 'extend' boundaries until another edge unit is reached that is set. A dangling end can be extended in one of three directions. See figure 9.

To encourage the completion of regions, a direction that will result in a connection to another edge is always preferred over a direction that will not. The gradient magnitude is used to resolve multiple choices. The range of search for the nearest existing edge can be extended beyond a distance of one. To prevent the explosion of possible directions in the search the options are restricted to the three initial directions. A restricted search of distance 2 is used in this implementation.

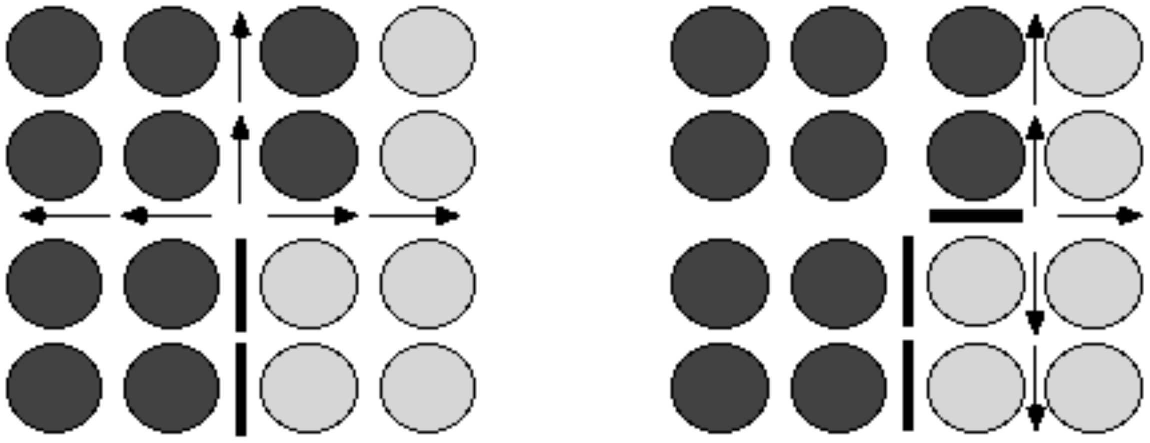


Figure 9: Extending the boundaries

The extension process is iterative with each dangling edge being extended one step per iteration. Figure 10 is a plot of dangling edges per iteration step. The graph shows that most dangling edges are resolved in the first few iterations. In the test images using 10 iteration steps has produced good results. The initial segmentation that results is shown in figure 7. There are 1672 regions. This is the segmentation to which region merging is applied.

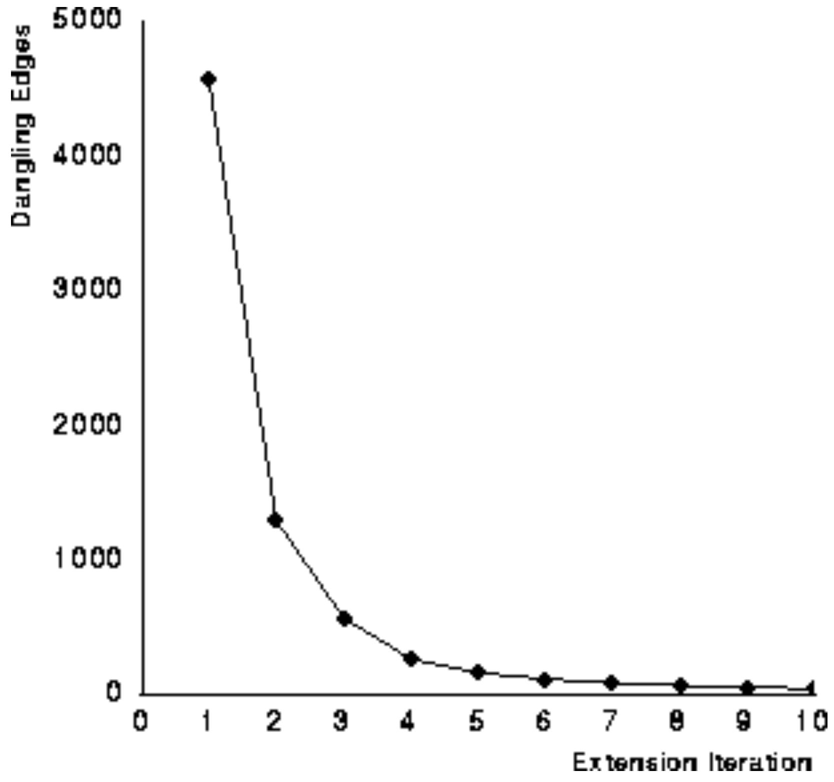


Figure 10: Dangling edges per extension iteration

Region Merging

A region merging phase is common in segmentation algorithms [Kohler, 1981 #5] to reduce over-segmentation. One is used in this algorithm to simplify the initial segmented image and is based on methods for texture classification. The texture analysis method is a modified version of the method by Modestino et al. found in [Modestino, 1981 #6] which classifies regions by degrees of contrast. The method detailed by Modestino uses a grey-level co-occurrence matrix as defined by Haralick [Haralick, 1973 #4] and applies an optimum maximum likelihood classifier to assign pixels to one of the predefined stochastic texture models. In the paper the authors concentrate on textures that are invariant under rotation so the the grey-level co-occurrence matrix is a probability distribution represented by a $Q \times Q$ matrix with element (m, n) defined as the probability that the grey levels m and n occur separated by distance d. Q is the number of grey-levels.

To classify a pixel (i, j), a measure of texture similarity is made against each defined texture. The pixel is assigned the texture with the strongest match value. This algorithm classifies regions by assigning to each region in the initial segmentation the texture occurring most frequently within its boundaries.

To calculate a match with a given texture T, a window around the selected pixel (i, j) is scanned. For each pixel (u, v) in the window many indexes into the co-occurrence matrix C_T (representing texture T) are formed by pairing the intensity m of pixel (u, v) with the intensity n of each pixel in a ring at distance d from (u, v). A running sum is kept of the *log* of $C_T(m, n)$. Special bookkeeping is necessary to avoid counting pixel pairs twice. The pixel *(i, j)* is set to the texture with the highest match value. The resulting texture is the maximum log likelihood estimate for the texture of the pixel.

There are only three textures allowed in this implementation. The textures are for low, medium, and high contrast regions. The histograms for each texture are from samples of other images that are not shown in this paper. The samples for the low, medium, and high contrast histograms are from a piece of cloudless sky, a field of grass, and a military tank, respectively.

Once the regions in an image have been classified it is possible to have a separate merge policy for each class and between any two classes. This capability is exploited by having a separate merge threshold for each type of region. When two regions are selected for merging, a threshold determines if the intensities of the regions are close enough to allow the two to merge. There are three thresholds for merging; one each for the low, medium, and high contrast regions. From empirical experimentation the threshold values expressed as a fraction of the intensity range have been set to 0.00 for low contrast regions and 0.10 for both medium and high contrast regions. A value of 0.00 means that such regions will not merge with any others. A value of 1.00 means that such regions will always merge with adjoining regions of the same class. Adjoining regions of different classes that are selected for merging use the lower of the two threshold values.

Figure 11 shows the effects on total region count of adjusting the thresholds *independently* from 0.00 to 1.00 while the other thresholds are at the aforementioned values. The plot for the low contrast threshold is not shown as even small values erroneously allowed significant regions to merge. As a result, this class of region is not allowed to merge. This is reasonable as low contrast regions tend to be large. Notice that the threshold value for medium contrast regions is at the low end of the knee of curve while the threshold for high contrast regions (coincidentally the same value) is higher on the knee of its curve. This corresponds to the intuition that high contrast regions are areas of interest and minimizing their number by very--aggressive merging is likely to result in important information loss. On the other hand, regions of medium contrast very often denote regions that are not interesting and should have aggressive merging techniques applied. The threshold values were chosen by experimentation and the subjective appraisal of the results.

The graph shows the effect of threshold values on the total number of regions in the segmentation. The total number of regions is not an objective indicator of segmentation quality; it is just a metric of the desired goal of minimizing the number of regions. A subjective evaluation of the segmentation quality was also made for each run. The number of significant regions that were misclassified steadily increased with the increase in either threshold. Simply put, the thresholds provide a tradeoff. High thresholds reduce the number of regions but increase the number of errors and low thresholds provide accuracy but may produce severe over-segmentation. The goal is to find an appropriate balance point. It is clear that there are diminishing returns with high thresholds. Values above 0.15 showed obvious errors. The value of 0.10 for both thresholds provides much of benefits of merging while still maintaining an accurate segmentation. Referring back to the initial segmentation in figure 7 gives an indication of the minimum errors possible in the final segmentation.

Figure 12 shows the classification of the regions of the house image. Here, black is for low contrast, white for medium contrast, and grey represents high contrast. Figure 13 shows the original classification of each pixel. Notice that the texture classification process seems to highlight the border between contrasting regions. If we are only concerned with edge detection we could use simpler techniques like a Sobel operator. The point of using the co-occurrence matrix is that the contrast of a *region* around the pixel of interest is measured rather than the contrast at a single point. I have found that the performance degrades if too small a window is used in the classification process. Future work should include analyzing the structure of the regions in the mathematical framework of [Modestino, 1981 #6] and solving for the optimal filter.

Observations and Contributions

Data Reduction. The implementation in this segmentation algorithm is a modification of the above method. The *Q x Q* matrix is collapsed into a single dimension array of size *Q* by recording only the difference between intensity values of pixels *m* and *n*. This removes the dependence of absolute intensity value as an indication of texture and it significantly decreases the amount of storage required for each texture definition. In addition, it significantly reduces the time of classification in a SIMD implementation. This will be addressed in a later section. With 256 intensity levels, 64K values are required in the original co-occurrence matrix but only 256 values are needed for the modified representation. In practice, only about the first 50 values seem necessary, decreasing the amount of data storage even further.

Region Classification. The original classification method is to classify individual pixels. In this algorithm merging is performed on regions. So, all pixels within a region must share the same classification. Selecting the simple majority texture for the region was an obvious design choice.

Non-optimal filter. Modestino et al. propose a method of calculating the optimum filter for the window around *(i, j)*. It is posed and solved as a Weiner filtering problem. The optimum filter was not calculated for the test images. A simple mask of ones is used that has proven very successful and reduces the amount of calculation by removing a multiplication step. The mask has width 3 and distance 1 from the center pixel. These dimensions were shown to perform well in the study by Dropsho in [Dropsho, 1994 #2].

Multiple Merging Policies. Using texture classification to group regions allows multiple merging policies to be used. This has proven a very useful technique. It should not be unexpected that regions with very different characteristics are best managed with different policies. In particular, classifying regions by contrast is very helpful in reducing the number of regions of natural objects. The medium contrast texture is often indicative of regions with a lot of edges due to randomly distributed pixels with slight variations in intensity. In general, such regions are not of interest in the image and should be simplified as much as possible. Classification allows aggressive merging tactics to be selectively taken.

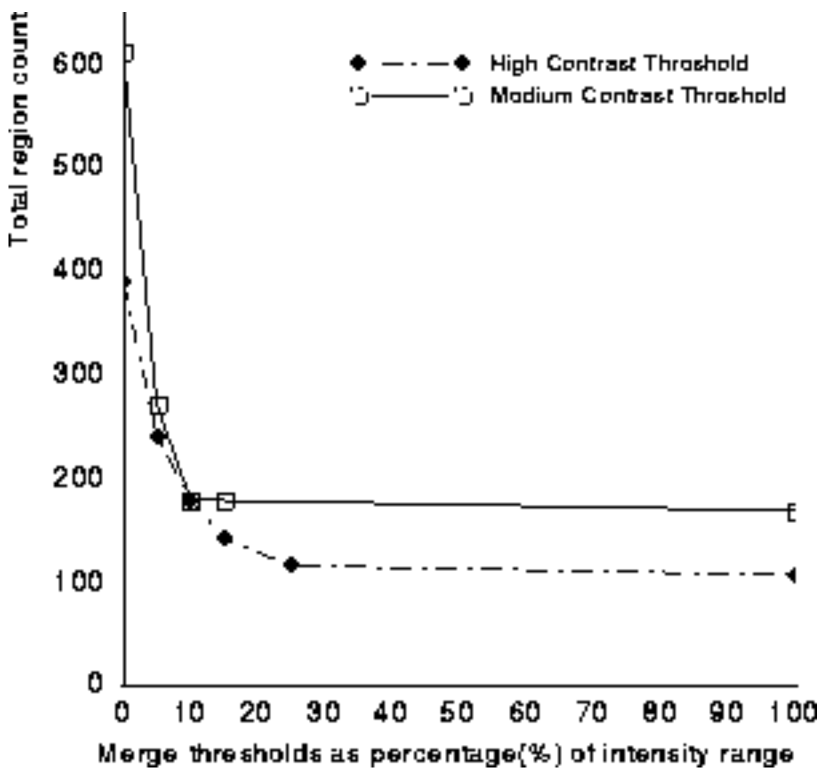


Figure 11: Effects of the merge factors on total region count of final segmentation

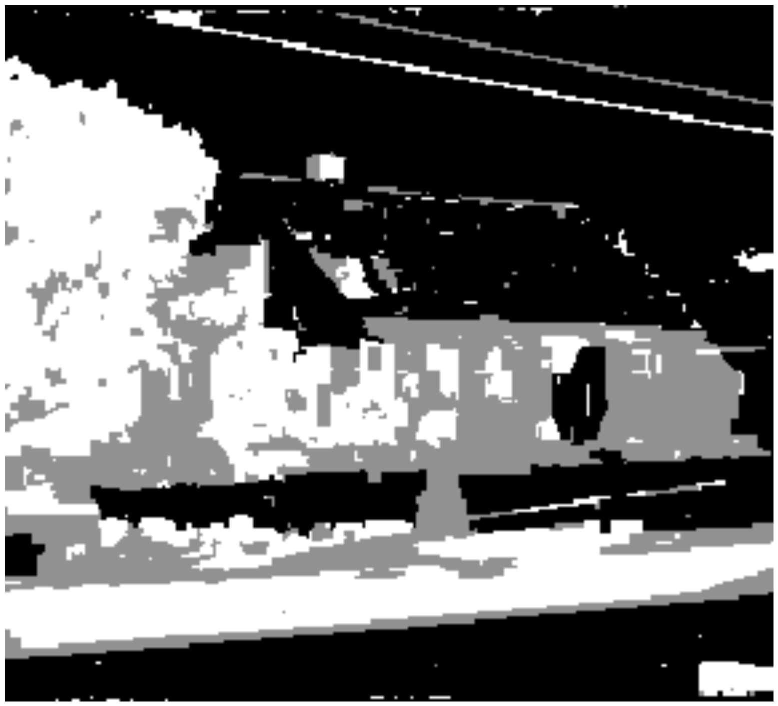


Figure 12: Texture classification of regions

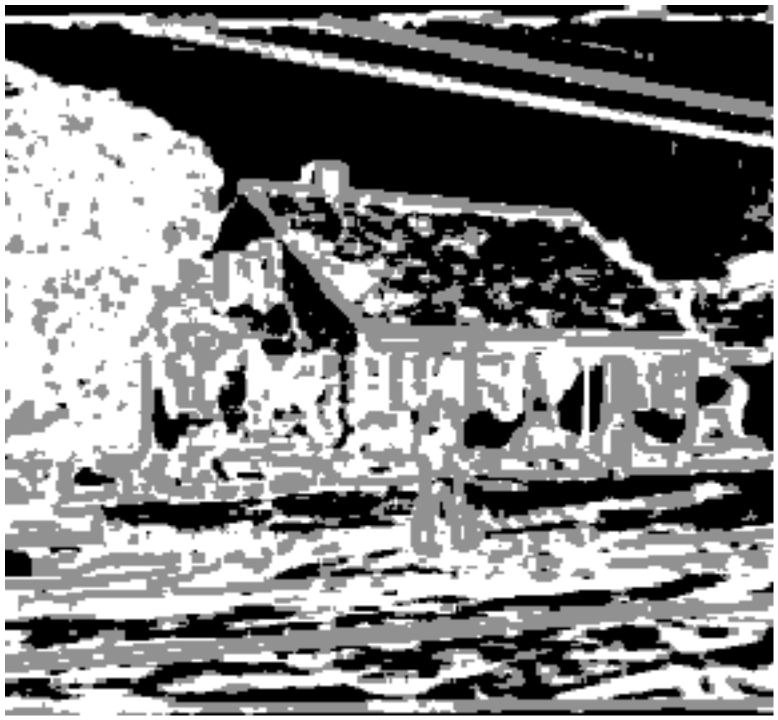


Figure 13: Texture classification of pixels

Color Images

In some cases it is very useful to use color to help in the segmentation process. Some images are very difficult to segment using only intensity information. The road scene in figure 14 is one such image. The result of applying the segmentation algorithm on the intensity image only is shown in figure 15. Notice that the upper part of the road is missing as is the lower right shoulder. The road sign is also poorly segmented.

Using images from the three video color bands can add information to improve the segmentation. The road scene is shown in the red, green, and blue bands, respectively, in figure 16. Notice that the road is highlighted better in blue. Also notice the difference in the road sign between the images. The segmentation of the individual bands is shown in figure 17. Individually, none of the segmentations are satisfactory, however, if we combine the edges of the individual segmentations a much better segmentation results as shown in figure 19. But, the image is now over-segmented. A merging phase is applied to reduce the number of regions.

The merging phase requires that each region is assigned a texture class. Which image should be used to classify the pixels? We could use the intensity image but there is more information in the color images. A conservative approach is to merge the classifications of each of the color images. On a pixel by pixel basis, the classification with the greatest **interest rating** is chosen. The order of interest of the textures has been set so that low contrast regions have the highest interest rating, followed by high contrast regions, and lastly medium contrast regions. From experience low contrast regions are usually segmented initially into large regions so do not need to be simplified by merging. High contrast regions generally denote objects of interest. On the other hand, medium contrast regions generally mark objects like foliage where merging should be aggressively applied. In general, if the segmentation of one color image classifies a region as low or high contrast it is usually an indication of a region of interest and as such should be preserved. Figure 18 shows the texture classification of the individual color images. Black is for low contrast regions, grey indicates high contrast regions, and white is for medium contrast regions. Figure 20 is the combined classification.

The final segmentation relies on an intensity image formed by combining the color images. The pixels of the created intensity image are grouped according to the boundaries defined in the combined segmentations of the color images. A merging phase is applied and the result is the final segmentation. Figure 21 is the final segmentation of the road scene. There are 278 regions. Comparing this to the segmentation of the intensity image alone (reprinted in figure 22 for convenience), notice that the road sign has been successfully separated from the surrounding foliage as has the upper portion of the road. The lower right portion of the shoulder is also better but there are still breaks in it due to the general low contrast of the region.

Additional detail can sometimes be gained by focusing attention in a particular region and redoing the segmentation in the restricted region of interest. Taking the segmentation in figure 21 as an example, a higher level vision process might generate a hypothesis about the existence of a road in the image but the poor segmentation along the right shoulder could be used to veto the correct hypothesis. By focusing attention in that particular region and performing a segmentation on the restricted region additional detail can be gained that, in this case, would support the hypothesis. The shoulder region in our example is shown in figure 23. The result of resegmenting on this region is presented in figure 24. By eliminating the sky and the darker regions elsewhere in the original image the intensities in the area of interest are now scaled to the full range of 0.0 to 4.0 which adds contrast between the regions. The additional contrast helps the algorithm reveal more detail. To achieve the same effect, parameters could have been adjusted to increase the sensitivity of the segmentation process. To date, settings of the parameters to define low, medium, and high sensitivity have not been explored. It has been shown in [Kohler, 1981 #5] that such settings are useful.

Another example is shown in figure 25. Only the green image is shown as the red and blue images are quite similar. The grassy field is very easy to over--segment as can be seen in initial segmetation in figure 26. The final segmentation is shown in figure 27 and the combined texture classification is shown in figure 28. Using texture classification allows the highly over--segmented field region to be very aggressively merged.

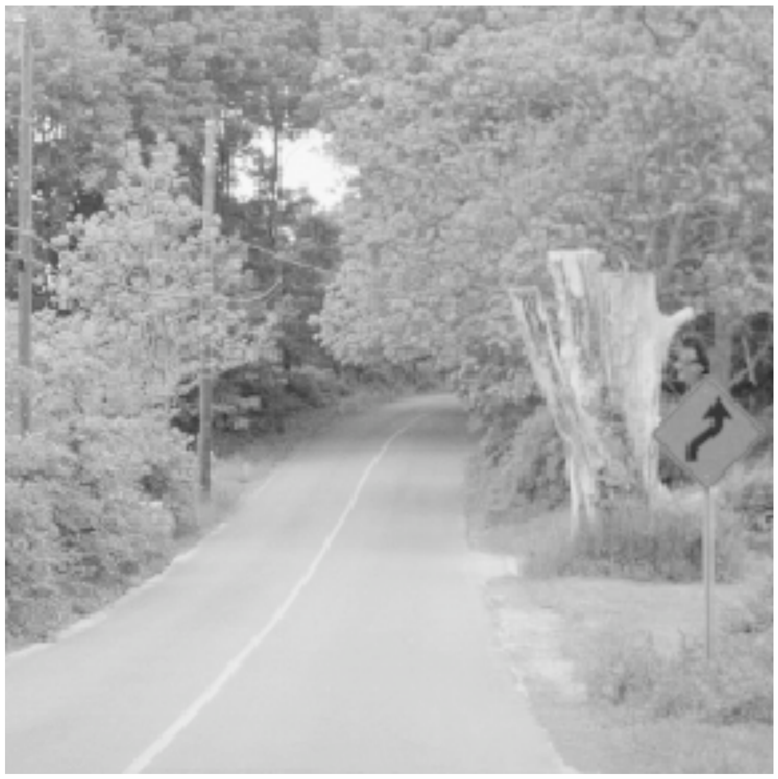


Figure 14: Road scene intensity image

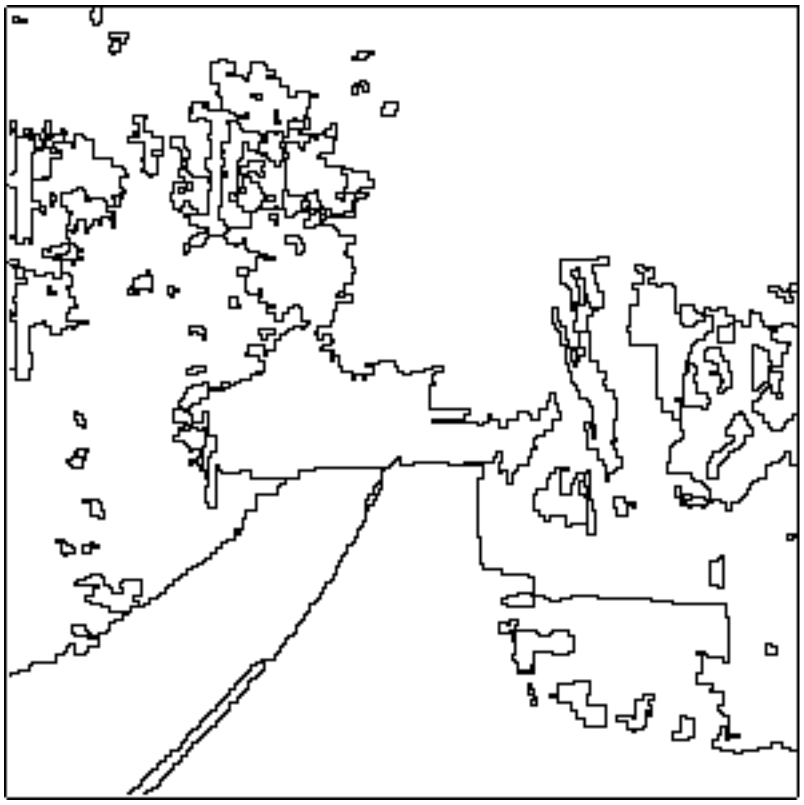


Figure 15: Segmentation of road scene intensity image

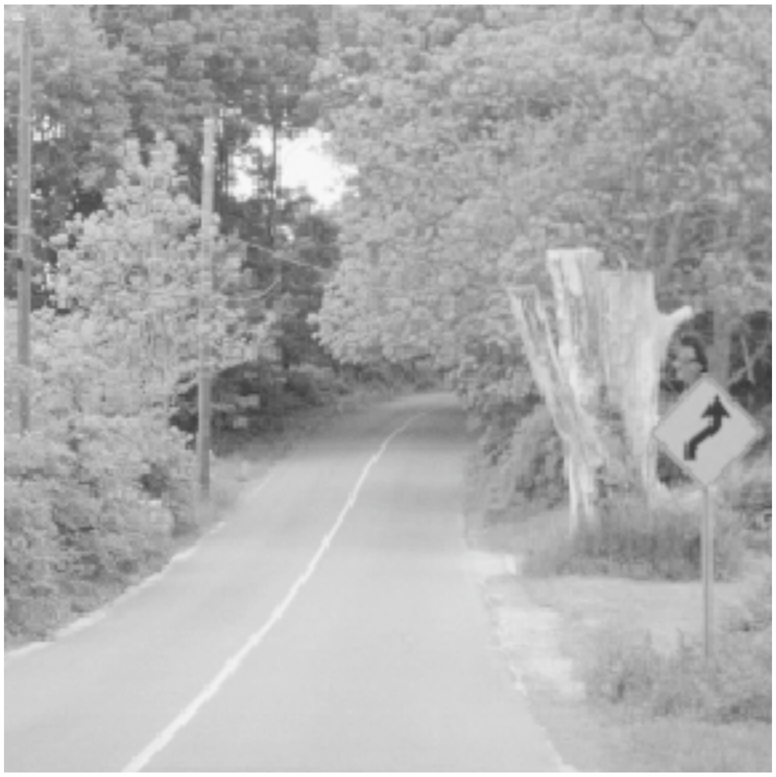


Figure 16a: Red image of road scene

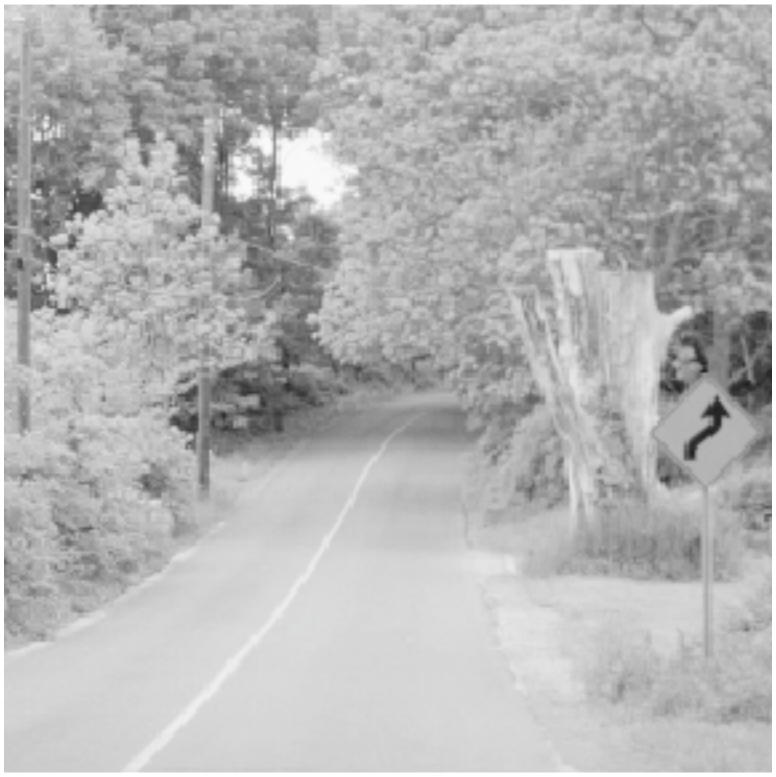


Figure 16b: Green image of road scene

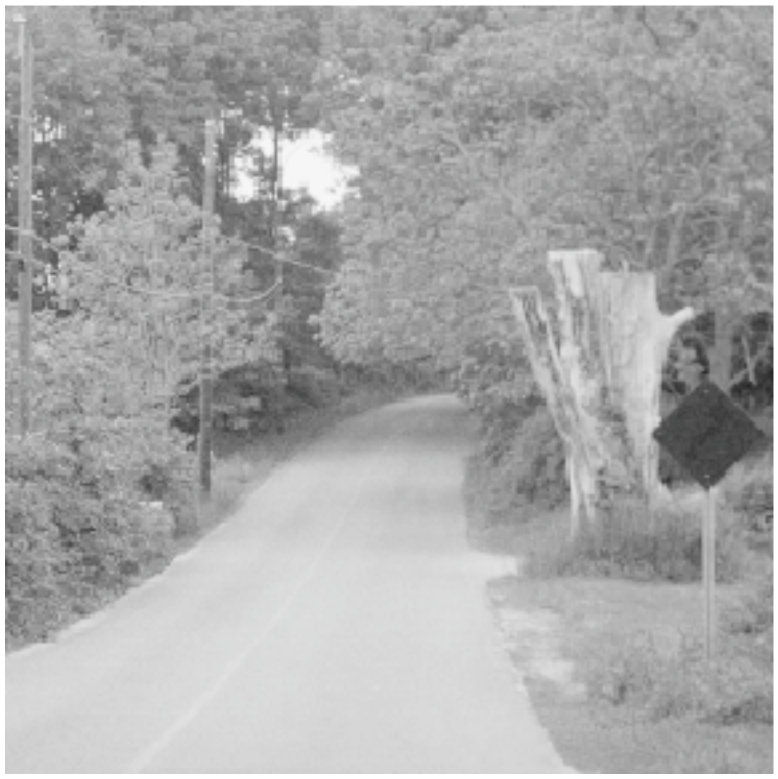


Figure 16c: Blue image of road scene

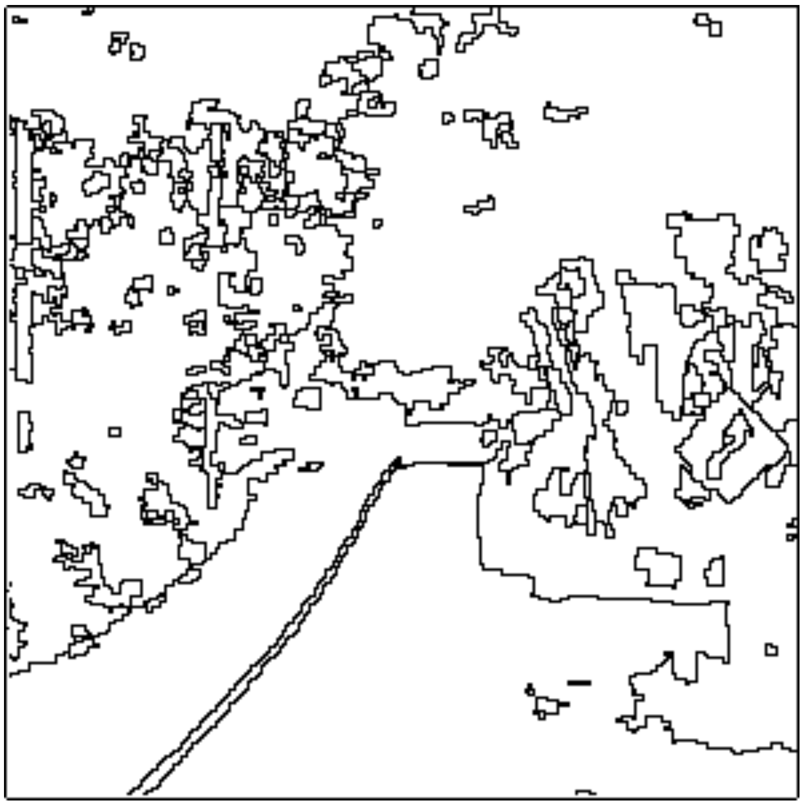


Figure 17a: Segmentation of red image of road scene

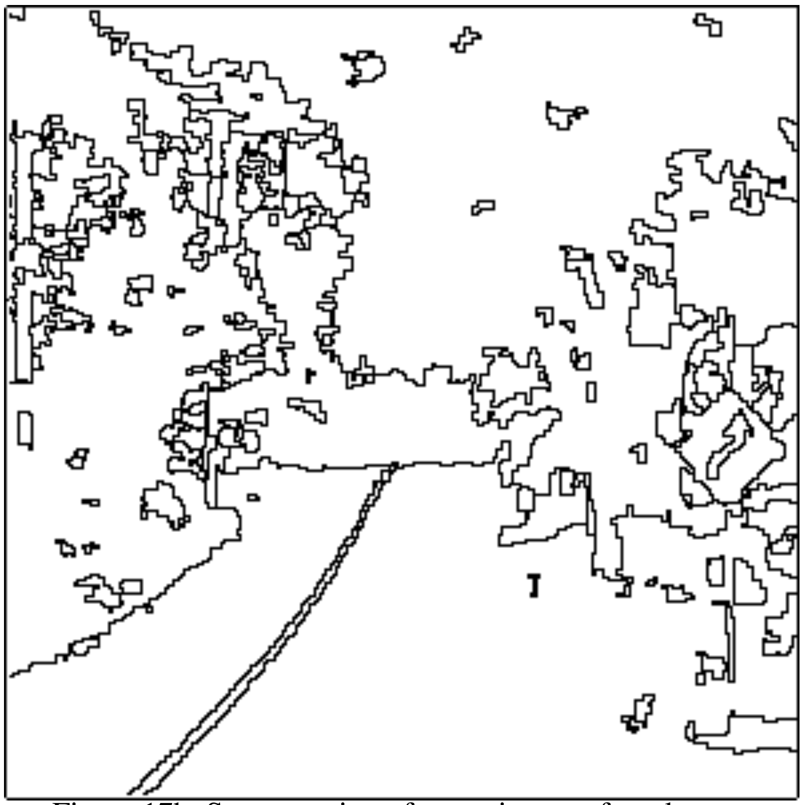


Figure 17b: Segmentation of green image of road scene

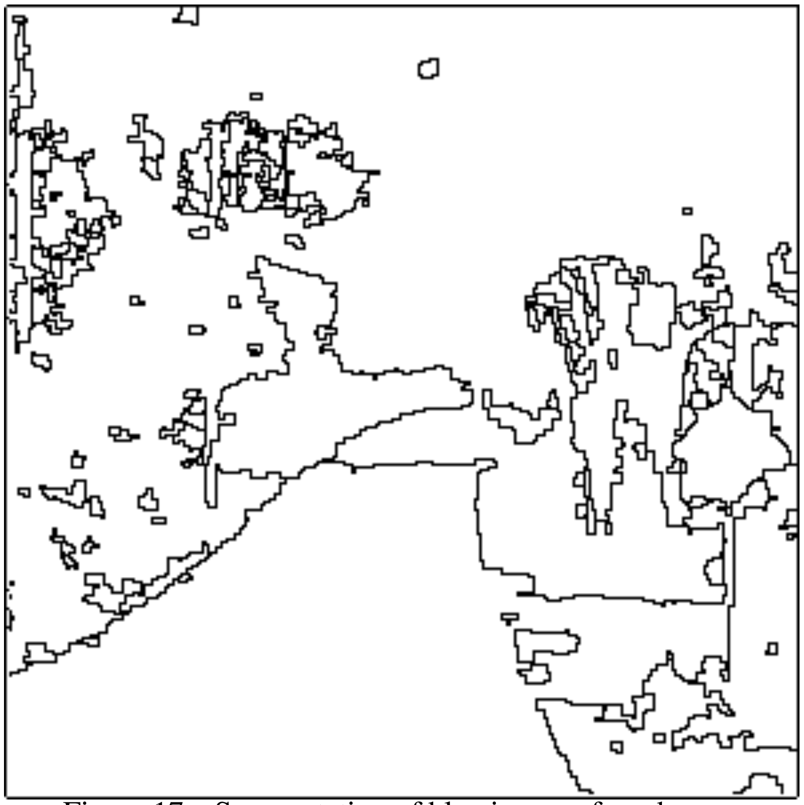


Figure 17c: Segmentation of blue image of road scene

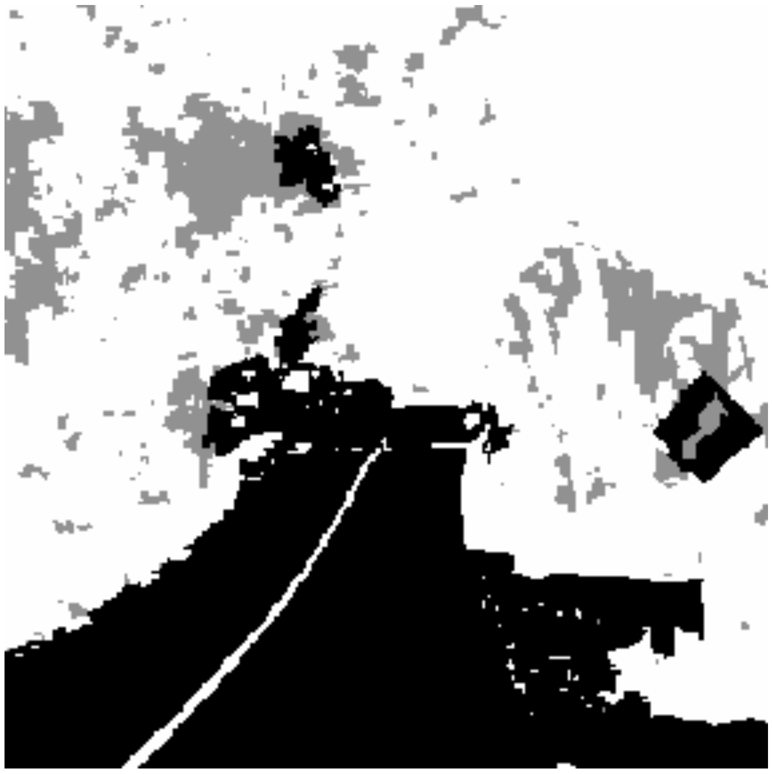


Figure 18a: Texture classification of the red image of road scene

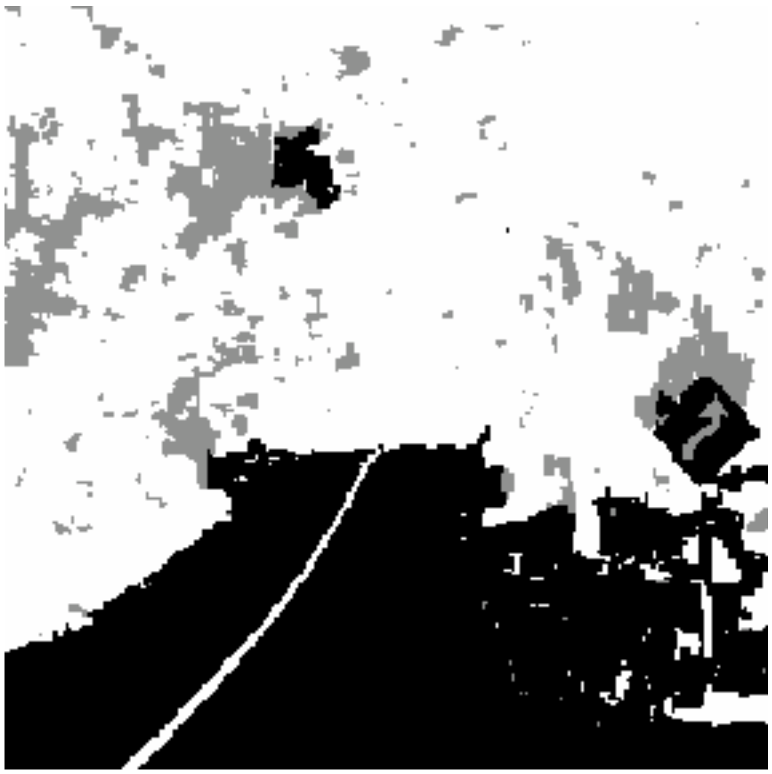


Figure 18b: Texture classification of the green image of road scene

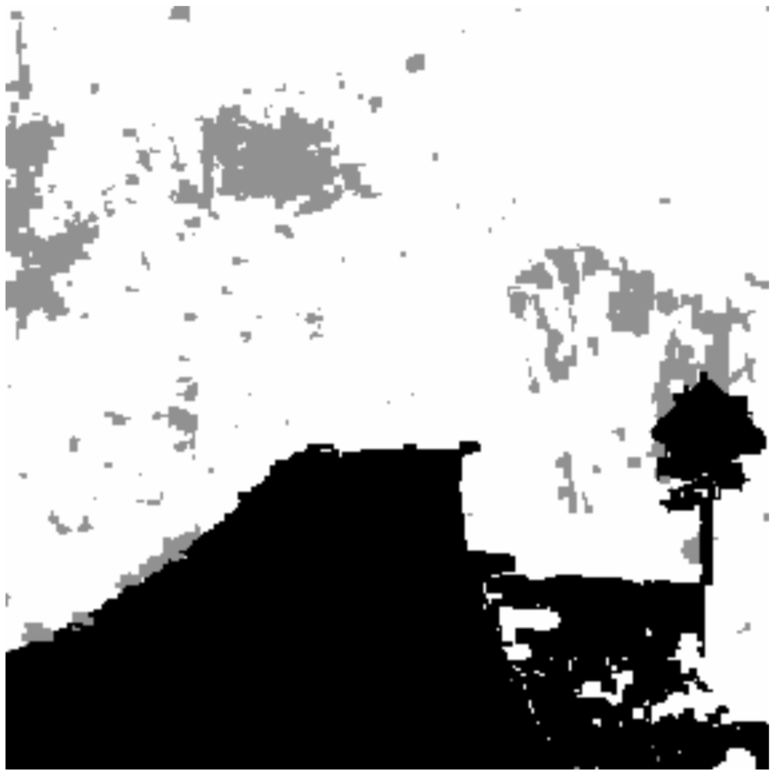


Figure 18c: Texture classification of the blue image of road scene

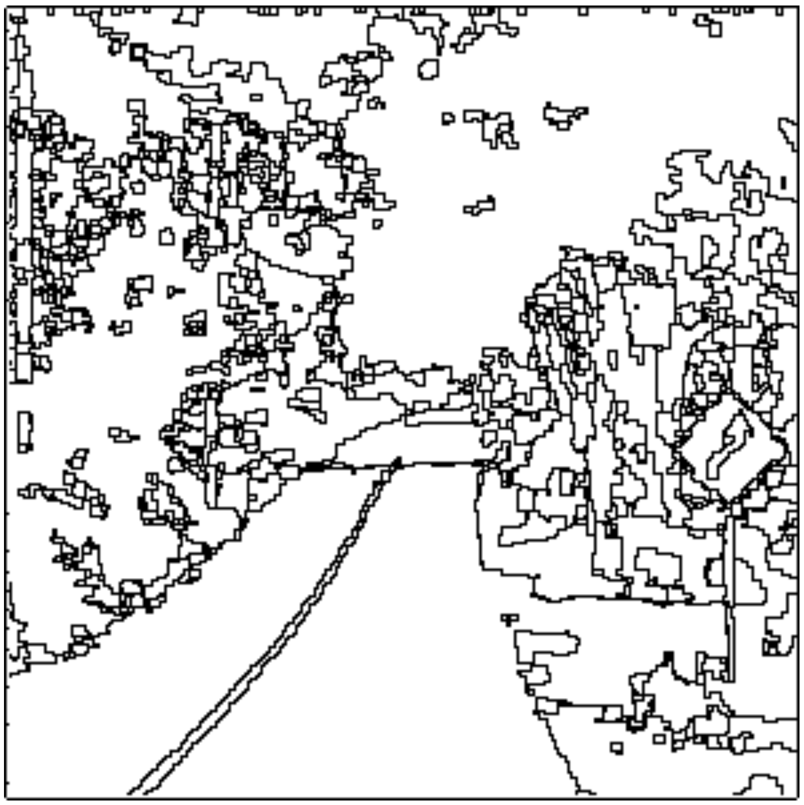


Figure 19: Combined edges from the three color segmentations

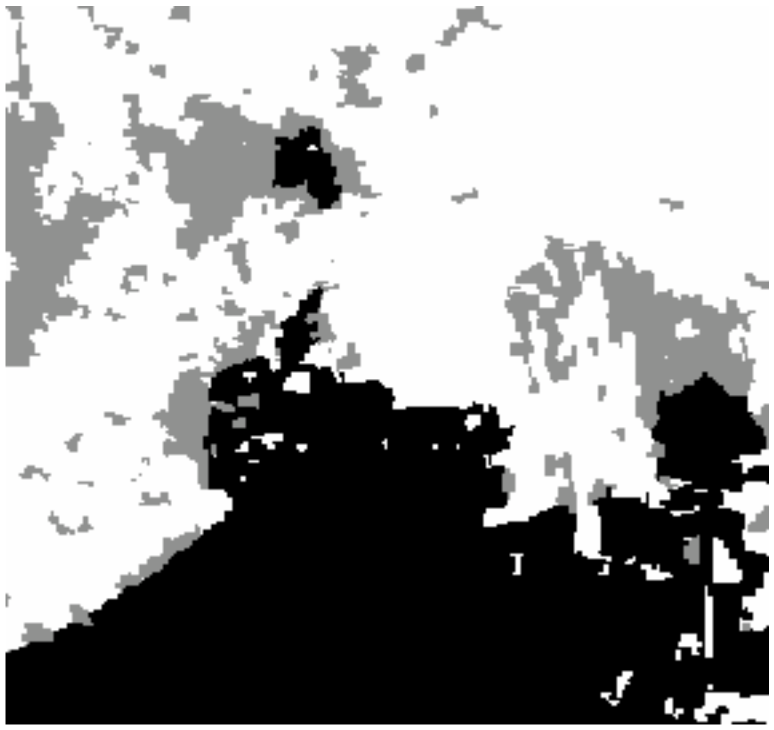


Figure 20: Combined texture classification of the color images

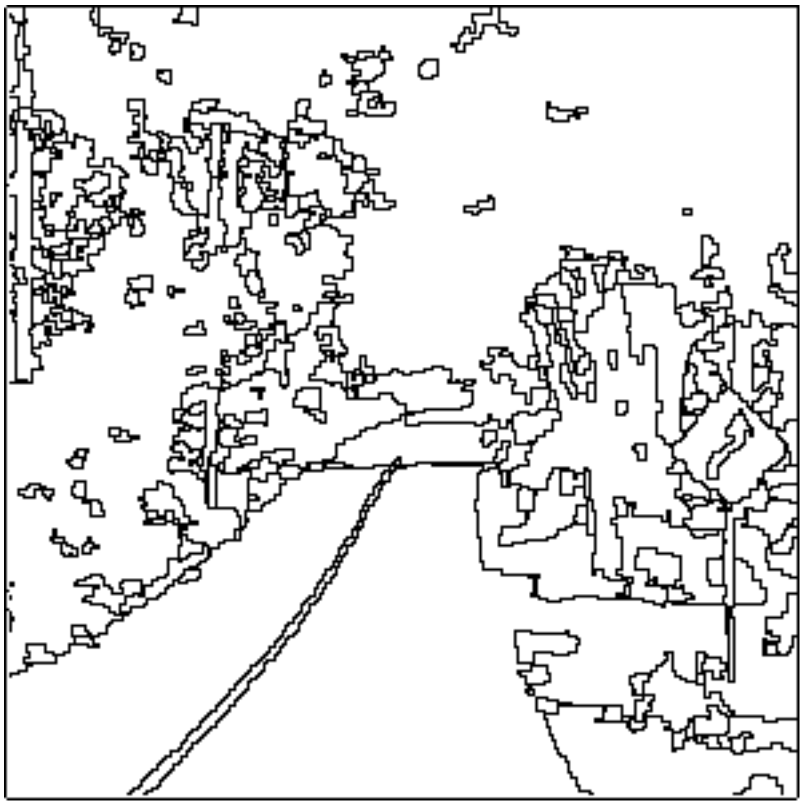


Figure 21: Final color segmentation of road scene, 278 regions

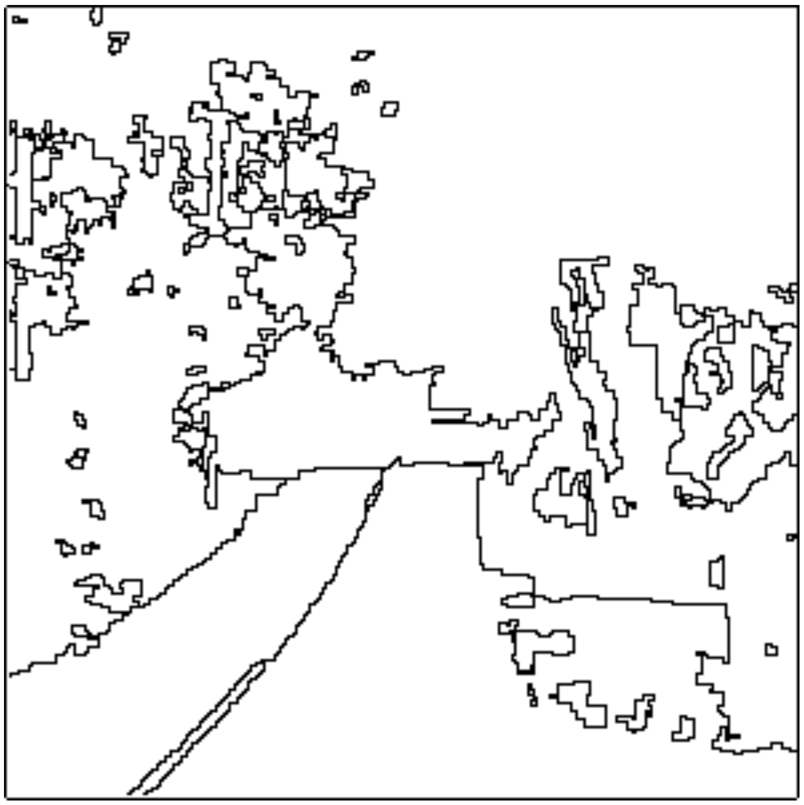


Figure 22: Rrprint of the segmentation of road scene intensity image

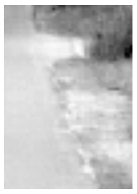


Figure 23: Shoulder section of road scene

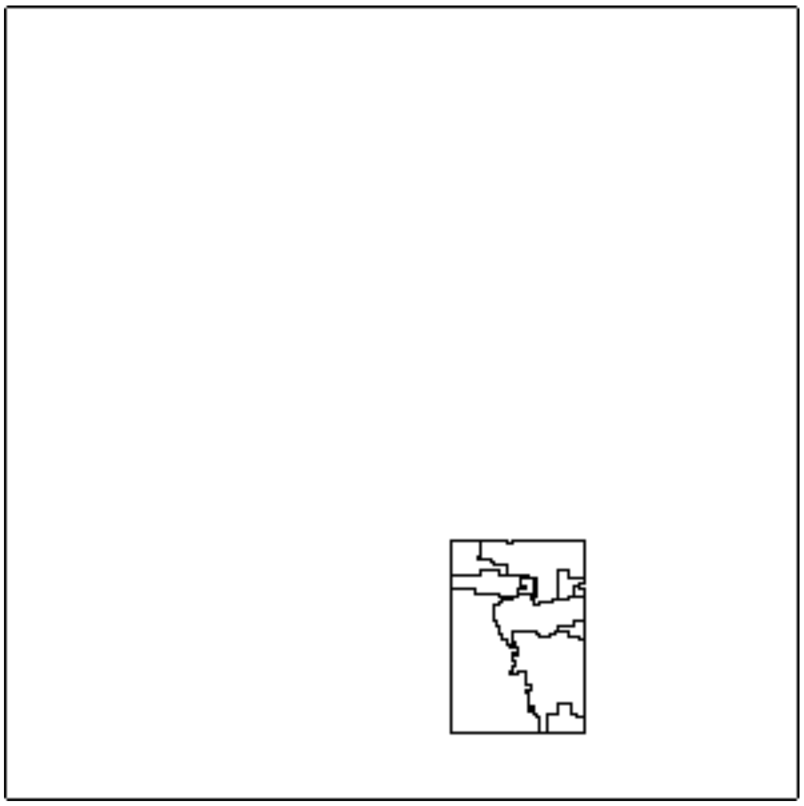


Figure 24; Segmentation of shoulder section of road scene

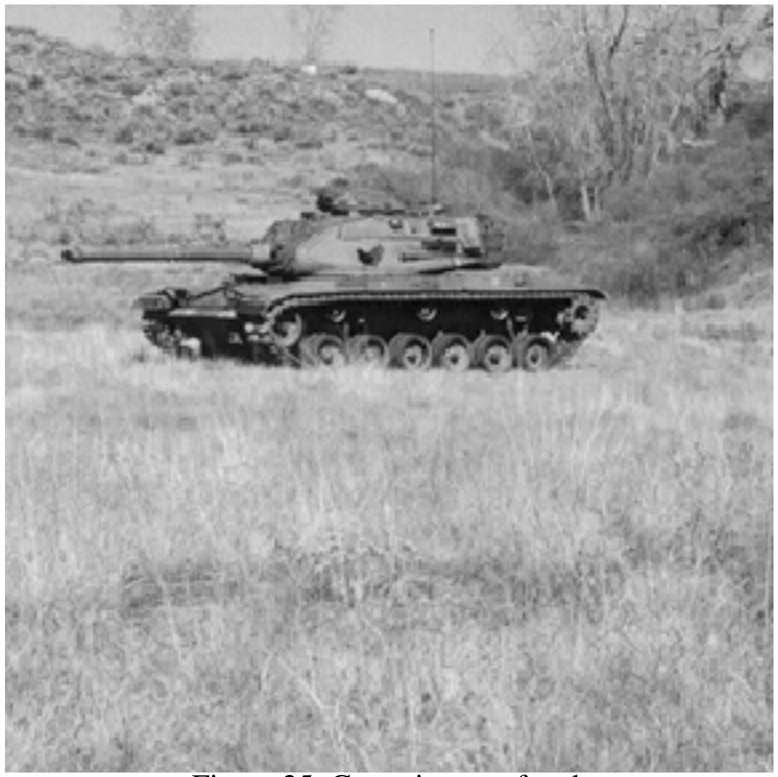


Figure 25: Green image of tank

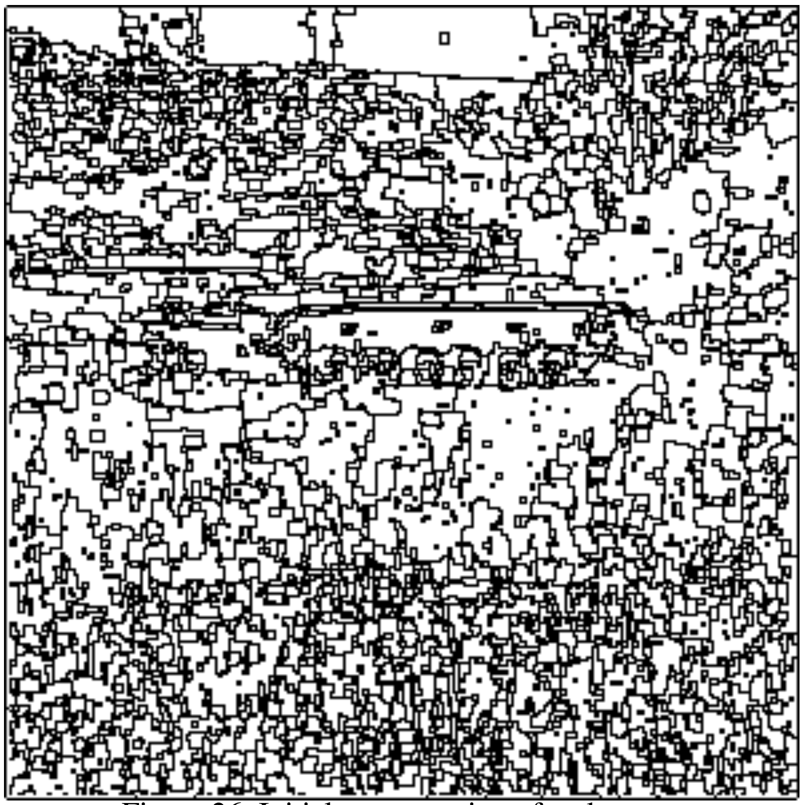


Figure 26: Initial segmentation of tank scene

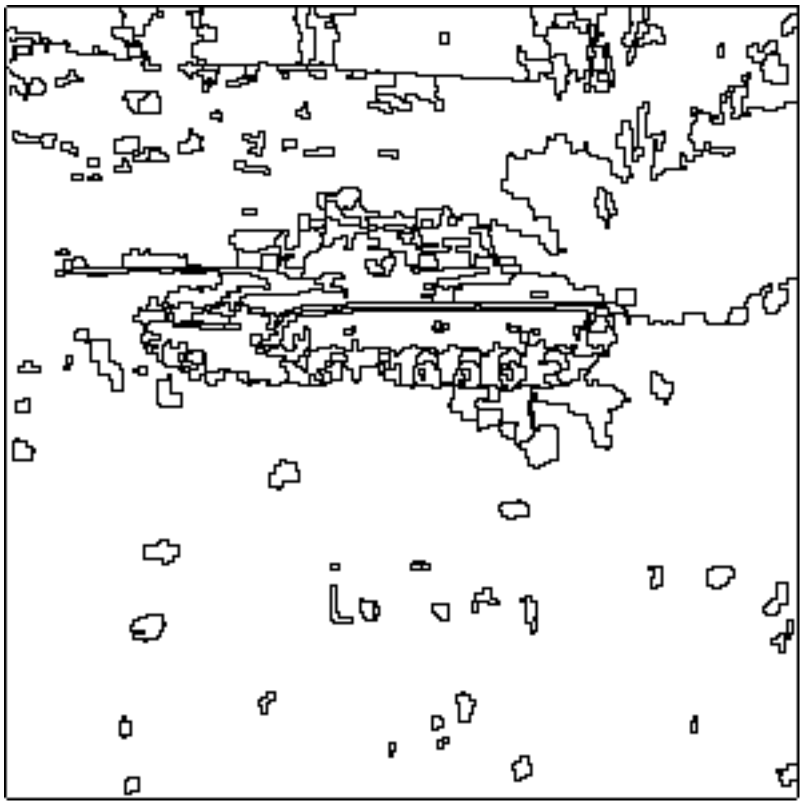


Figure 27: Final color segmentation of tank

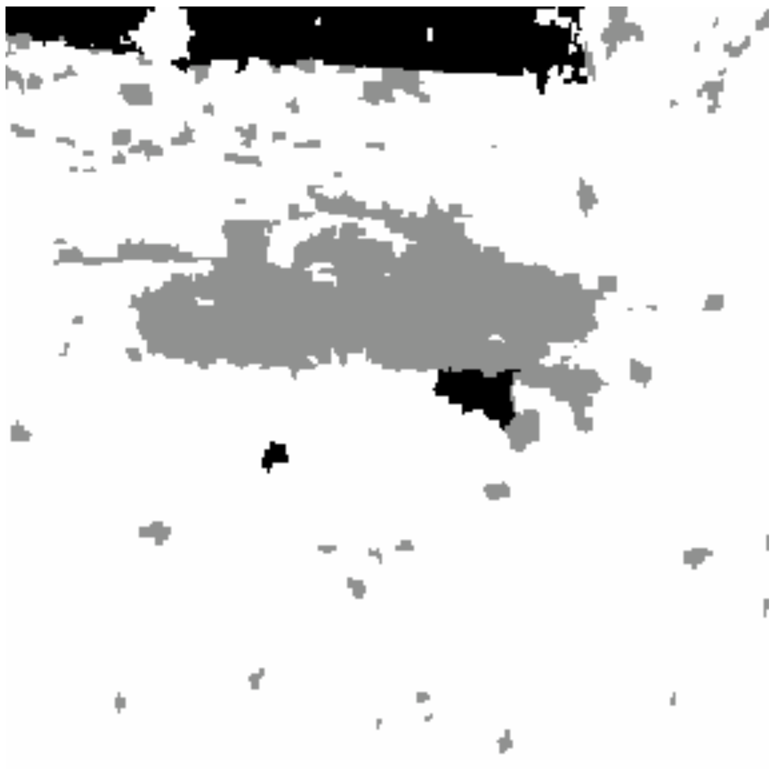


Figure 28: Texture classification of tank

Performance

Serial Performance

The segmentation algorithm is very modular and has five main components. Table 1 lists the components and the percentage of total run--time attributable to each. The first three components are from the edge detection stage.

Code Execution Time Breakdown	
Horizontal Network	11%
Vertical Network	11%
Combining Network	26%
Edge Extension	7%
Merging Phase	33%
Miscellaneous	12%
TOTAL	100%

Table 1: Main Algorithm Components

The runtime of the algorithm is essentially independent of the image. On ten test images the run--times were very close. On a 100MHz MIPS R4400 processor the quickest ran in 90 seconds and the slowest in 97 seconds to segment a single image. Segmenting a scene using the color images is essentially three times as long since three separate images are being segmented. The additional work done to combine the three images is almost totally offset by leveraging the initialization of data structures over the three images.

The algorithm is parallelizable at many levels. Of course, if color segmentation is being done then each image can be processed independently and then quickly combined at the end. Within the segmentation of an image the horizontal network and the vertical network code can be run simultaneously, however, this can only reduce the run--time by about 11\%. The most dramatic speedup can be attained only if each component is able to run in a SIMD fashion. This is explored in section~\ref{parallel_section}.

Data Structures

A proper data structure minimizes the overhead of managing regions during the merging process. The merging process requires information about regions such as the average color (intensity), size, and neighboring regions. Since regions are being merged this information must be able to be recalculated and updated very quickly. The Union/Find--Set structure as described in [Rivest, 1992 #7] allows very efficient access to this information. For each region a pixel is selected to be the **parent**, or representative of the region. All information about the region is accessed using the parent's indices in the image as the relative address into the other data structures. Information about a region can be quickly retrieved by querying any pixel in the region for the information. Using the Union/Find--Set structure, the parent is quickly found and its indices are then used to find the requested information in a table lookup. In fact, using the **union by rank** and **path compression** heuristics in the Union/Find--Set structure reduces the upper bound on the *average* time per query (e.g., number of pointers followed) into the data structure to $O(lg^* n)$ where n is the number of potential regions. For a 256 x 256 image the number of potential regions is equal to the number pixels, 2^{16} and $\lg^* 2^{16}$ is 4. Thus, regions are queried in essentially constant time. In addition, each parent has a pointer to the head and tail of the list of pixels representing neighboring regions so the neighbors can be quickly queried and combined during the merging phase. The structures for recording size, color, and neighbors are two dimensional arrays with an entry for each pixel and require approximately 3.3 megabytes if segmenting a 256 x 256 image.

Parallel Performance

This section explores the performance of the algorithm on a SIMD architecture. The algorithm has been ported to the Image Understanding Architecture (IUA) low--level processing layer called the CAAPP. The IUA is an architecture with multiple types of parallelism and is designed for vision applications. There are three levels of computing elements in the IUA. The top two have 32-bit processors connected in a MIMD fashion. The lowest level, the CAAPP, is a 256 x 256 array of simple 1-bit processing elements (PEs). Although the multiple layers are very tightly coupled, the algorithm is coded to use only the SIMD CAAPP layer. The cycle time of the CAAPP is 100 nanoseconds.

The PEs of the CAAPP are arranged in a mesh with direct communication channels to the four nearest neighbors. The mesh is dynamically partitionable through the Coterie network, a set of switches that make or break connections between the nearest neighbors. Thus, sets of PEs can be grouped and isolated from others in arbitrary patterns. This is a useful feature that supports segmentation.

The algorithm divides cleanly into three distinct stages: edge detection, edge extension, and region merging. The three stages have distinctive computational needs and are discussed individually.

Edge Detection

Effects of Precision. The two networks that perform the edge detection have arithmetic operations as a large percentage of their total operations. Table 7 in the appendix lists the types of operations and their frequency in the implementations of the two networks. Arithmetic operations make up 58\% and 64\% of the operations in the procedures, respectively. Since the processors are bit-serial there is a direct correlation between the time in cycles of the arithmetic operation and its complexity. The complexity of addition and subtraction is $O(n)$ where n is the number of bits in the operands. Multiplication and division have complexity $O(n^2)$. As a result, the procedures for the networks are very sensitive to the precision of the operands. The precisions considered are byte (8 bits), short (16 bits), integer (32 bits), and floating point (32 bits).

From the tables in the appendix it is obvious that floating point operations are in general much more time--consuming than integer arithmetic, however, floating point datatypes have the advantage that they are very convenient for the programmer to use when non-integer values are required. Bytes, shorts, and integers can be used in such computations by using scaled values to represent the actual range of values. For example, 8 bits can be used to represent values between 0 and 1 with increments of 0.0039. While addition and subtraction are straight forward, multiplication and division require that the placement of the implicit decimal point be maintained during operations. This requires some additional work at the end of the operation to shift the result. While this is more cumbersome to program than floating point, it has the advantage of being much faster on additions and subtractions and potentially faster on multiplication and division. If the precision is 16 bits or fewer then scaled integer is indeed faster. This last qualification is an artifact from multiplication/division being faster on floating point values than on full 32 bit integer values. The floating point datatypes simply add/subtract their exponents and then only do a 23 bit multiply (the width of the mantissa). The integer datatypes, on the other hand, must do a full 32 bit multiply.

Also considered is a datatype float16 that is not currently available on the CAAPP. The datatype float16 is a 16 bit floating point number that has an 8 bit mantissa and an 8 bit exponent. Estimates of its instruction times are given in the appendix. This datatype combines the ease of programming that floating point datatypes provide while enhancing performance by decreasing the precision. The following sections will discuss its use. A quick remark should be made about the float16 datatype. Its implementation does not require any hardware support not already in the CAAPP. It can be incorporated by microprogramming the controller and adding compiler support to make it a new first-class datatype.

The First Network. Edge detection is performed by two different networks that perform a relaxation process to minimize an evaluation function (the energy function). Most of the operations by these networks is arithmetic so their run--times are heavily dependent on the precision of the operands. The first network detects potential vertical and horizontal edges. This network is used for both directions with only slight modifications to reflect the direction. The effects of the data precision on run--time are graphically shown in figure 29. The raw data is shown in table 2. Two values are given for each precision. The first value excludes cycles where the array is waiting or idle; a portion of these cycles are due to the inefficiencies of the particular array controller. The second value includes these cycles. The entry for float16 is an estimate using an approximate value for the operations assuming a 16-bit floating point operand.

The Second Network. The second network combines the outputs from the two instantiations of the first network and generates the final set of edges. Again, due to the emphasis on computation in this module the run--time is very sensitive to the precision. Figure 30 shows the run--times relative to precision.

Procedure Execution Times (ms)

Procedure	Byte	Short	Int	Float32	Float16
Network 1	1.5	3.6	10.6	18.8	5.8
w/waits and idles	2.2	4.7	12.3	22.0	NA
Network 2	3.6	8.0	21.9	62.8	19.1
w/waits and idles	5.5	10.5	25.1	73.9	NA

Table 2: Execution Times for Procedures

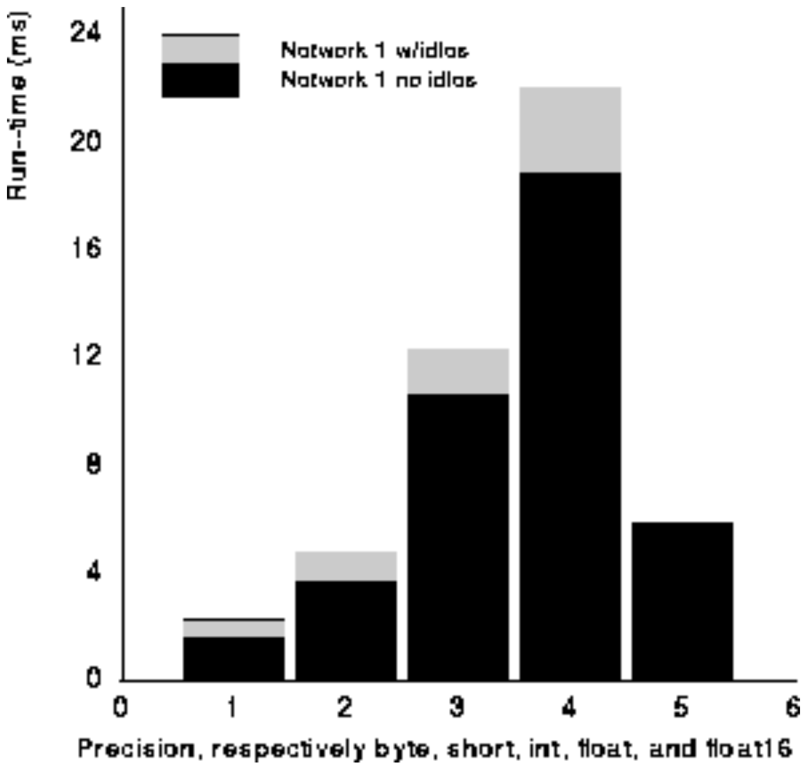


Figure 29: Execution time of first network

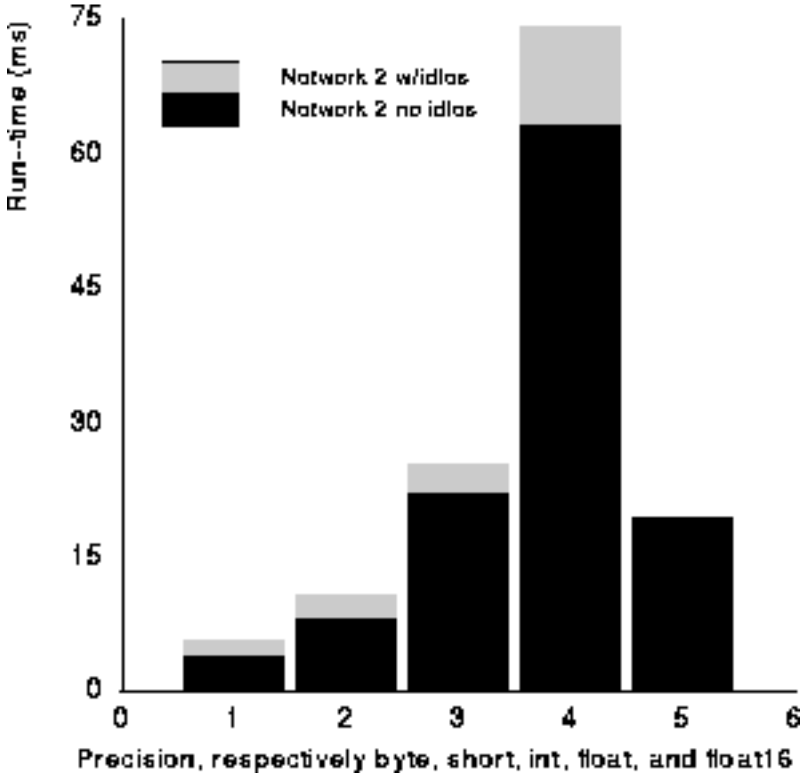


Figure 30: Execution time of second network

It is clear from the charts that the run--time is very sensitive to the precision. Ideally, one would prefer to use the floating point representation so there is full precision and minimal information lost due to approximation. But there is clearly a significant time penalty for using more precision than is required. While floating point is adequate during the development stage every effort should be taken to minimize the precision of the operands to the bare minimum.

Required Precision. To determine the minimum precision that provides adequate results, two factors must be known. The first factor is the largest magnitude that any intermediate calculation can generate in either network. This is easily calculated and is bounded by the value 64. The second factor required is the sensitivity of the segmentation results to the discretization of the real number space. Since the intensity is scaled to the range 0.0 to 4.0 from the range 0 to 255, a total of 8 bits are required with 6 bits to the right of the decimal point. Thus, we expect the required precision to accomodate the maximum magnitude of any intermediate results with an additional 6 bits for the fraction. The maximum magnitude for both networks is bounded by the value 64. Hence, 12 bits of precision may suffice. This is only an estimate and does not tell us anything about the sensitivity of the algorithm as a whole to the precision in these modules. Such sensitivity is difficult to measure.

One method to do so is suggested in [Dropsho, 1994 #2] that details a statistical method of comparing techniques across a suite of test images to give a statistical measure of the liklihood that two techniques differ in the quality of their results. This method can be used to compare segmentation results across a range of precisions. This has not yet been done and only a single image example is used here to illustrate the effects of precision.

Figures 32, 33, 34, and 35 are the results of restricting floating point precision to 16, 12, 8, and 4 bits respectively. These results can be compared to the full precision segmentation reprinted in figure 31. Notice that with the precision at 8 bits portions of the telephone line are missing. This precision appears to be at the lower bound of acceptability for the algorithm and suggests that the algorithm is fairly robust in terms of the precision for the edge detection stages.

In this example, 8 bits of precision appear just adequate while 12 bits are certainly sufficient. With a requirement of a maximum magnitude of 64 and 12 bits of precision, a floating point datatype with a 4 bit exponent and a 12 bit mantissa could be implemented in the controller of the CAAPP array. While this would be adequate for this example the datatype lacks the range for general use. The float16 datatype has with an 8 bit exponent and 8 bit mantissa and could be used to speed up the edge detection phase over using full floating point while still providing the ease of programming of floating point datatypes. The loss of precision has to weighed against the ease of use. However, scaled integer arithmetic of 16 bits is adequate for this example and has a performance advantage over float16 arithmetic.

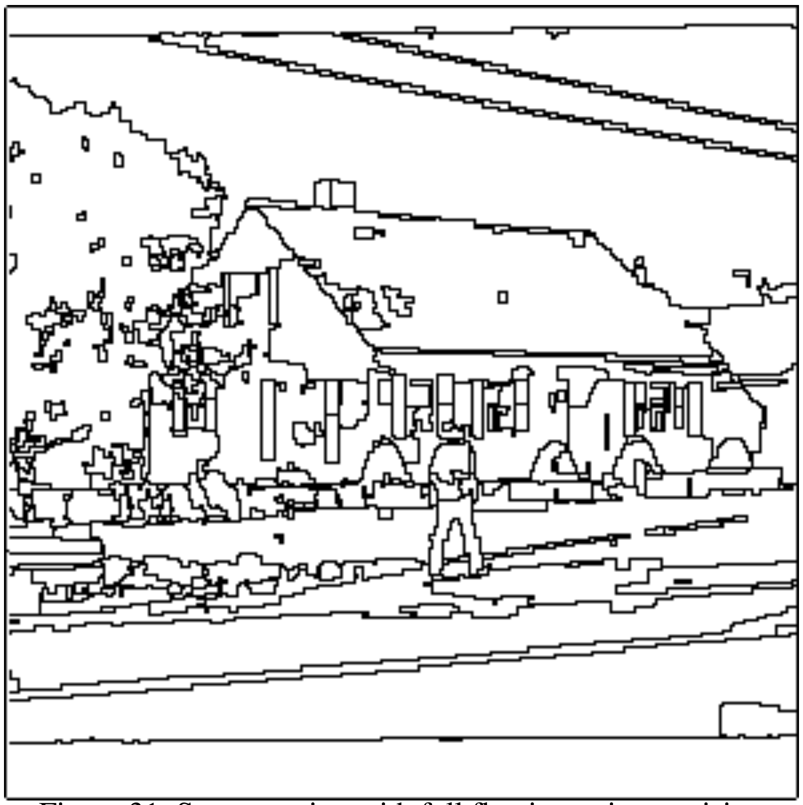


Figure 31: Segmentation with full floating point precision

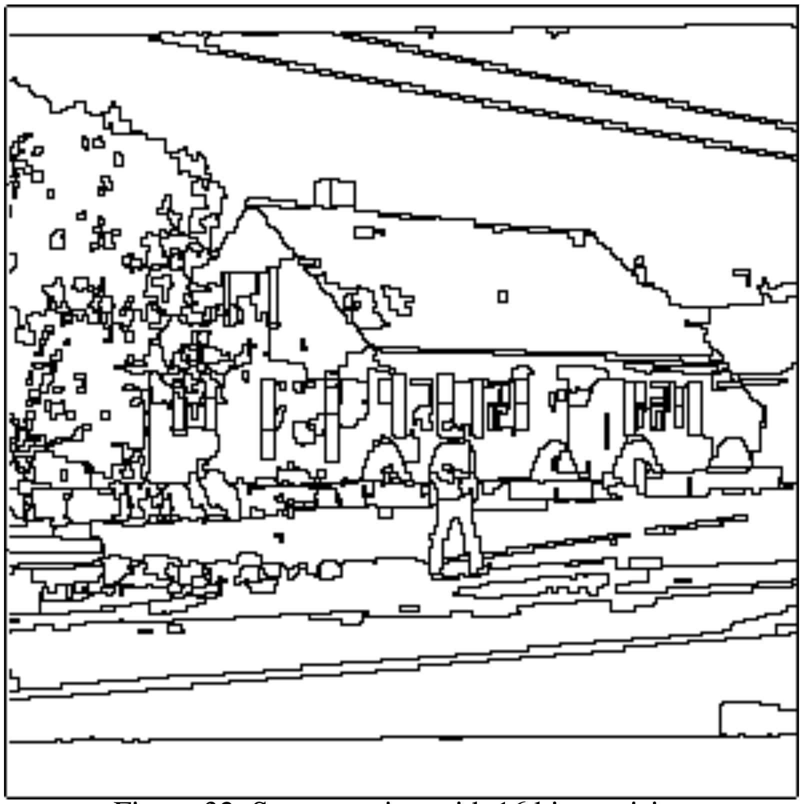


Figure 32: Segmentation with 16 bit precision

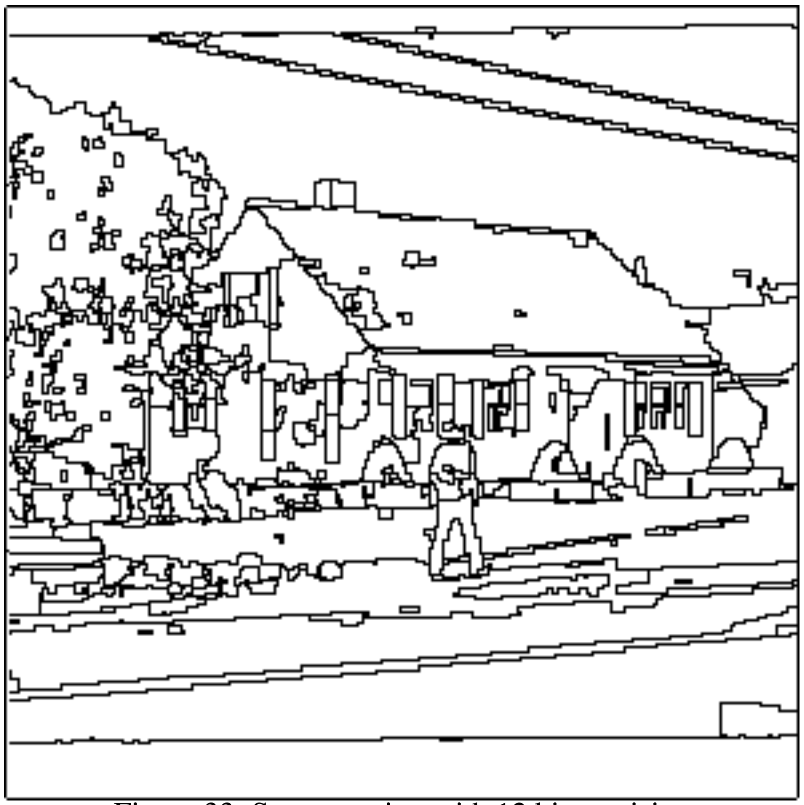


Figure 33: Segmentation with 12 bit precision

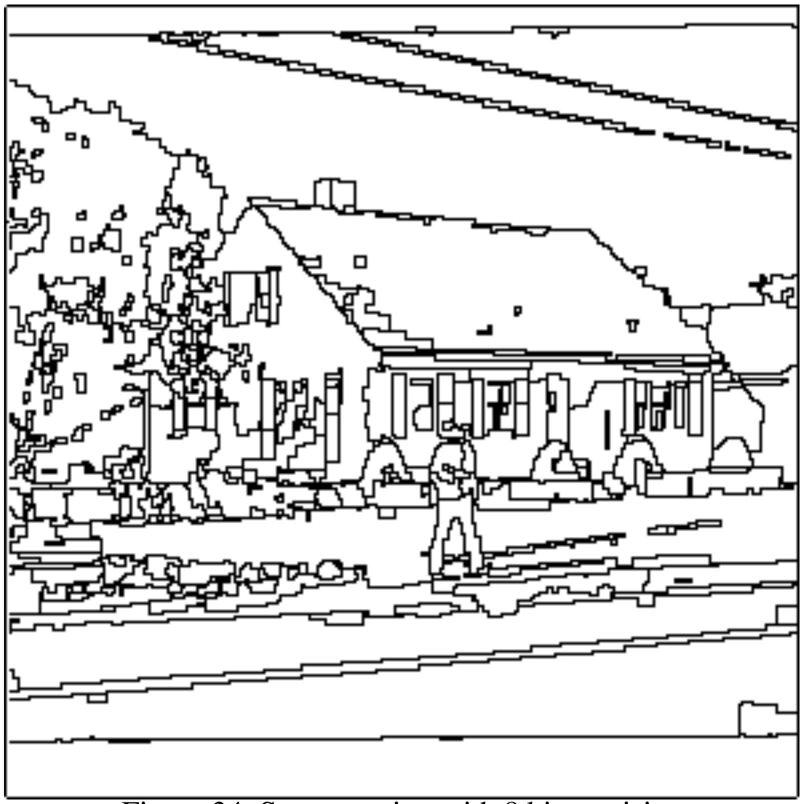


Figure 34: Segmentation with 8 bit precision

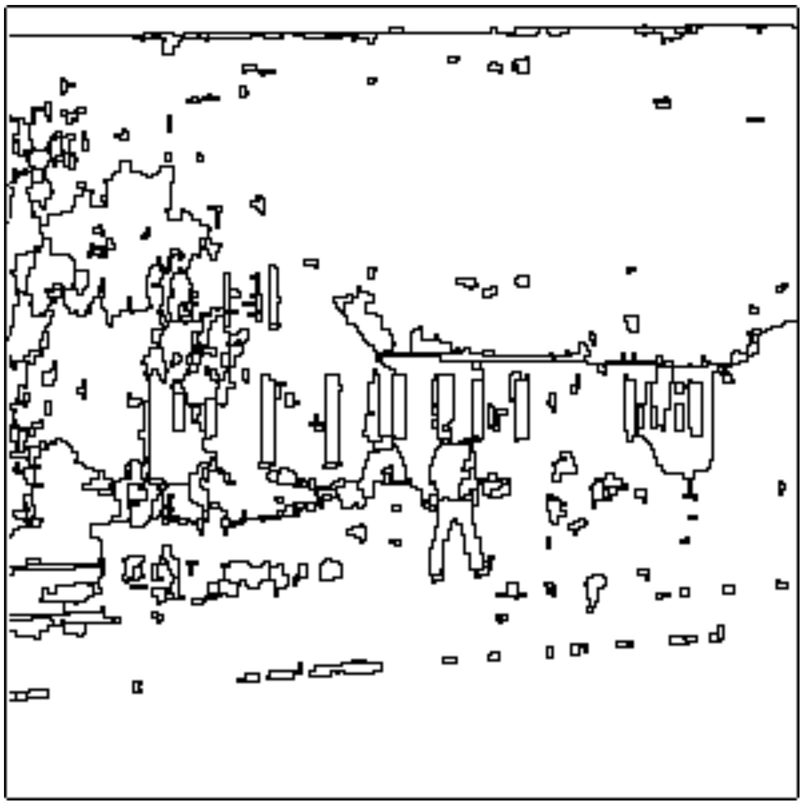


Figure 35: Segmentation with 4 bit precision

Approximation of the *tanh* function. Another method of increasing performance is to approximate expensive calculations with simpler, but less precise methods. Such is done with the *tanh* function. A piecewise-linear approximation replaces $y = tanh(x)$ with

$$y = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$

Figure 36 graphs the two functions together for comparison. The approximation is adequate since the function is primarily to limit range of the output. And, this approximation reduces the time of the *tanh* function from 37 milliseconds to 45 microseconds, a factor of over 800 speed up. From the tables in the appendix you can see that the *tanh* function is called about 20 times in each network for a total of 60 times in the algorithm. The calls to the actual *tanh* function would take over 2 seconds alone on the CAAPP.

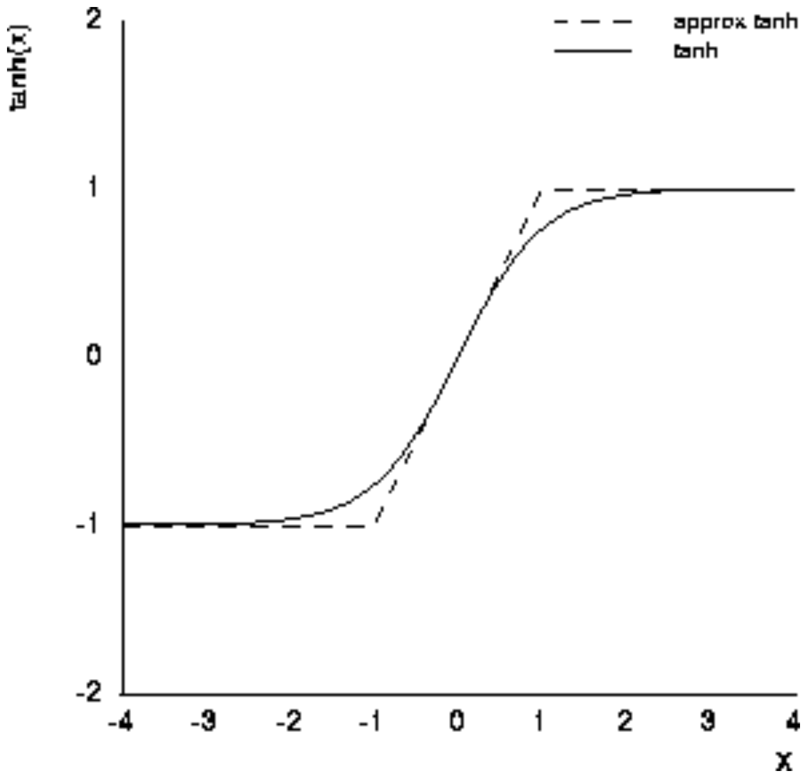


Figure 36: Graph of tanh vs. piece-wise-linear approximation

Edge Extension

The edge extension phase only requires calculating the differences in intensity between pixels so this phase is implemented with the image scaled to the traditional range of 0 to 255 and not the 0.0 to 4.0 range used in the edge detection stage. As no division is done, fractions are not generated so the arithmetic is strictly integer arithmetic. The current implementation requires 9 bits to encode the maximum value generated. As shown in table 3, the run--time is 4.5 ms if wait and idle cycles are excluded and 10.7 ms if all cycles are included. All operations used are simple operations that the CAAPP performs efficiently. There are no recommendations for run--time performance improvement.

Procedure Execution Times (ms)

Procedure	Run-Time Excluding RoutePlus	Runt-Time Including RoutePlus	Percentage in RoutePlus
Edge Extension	4.5	4.5	0%
w/waits and idles	10.7	10.7	0%
Texture Classification	8.8	46.0	80.9%
w/waits and idles	13.7	NA	NA
Region Merging	9.3	158.1	94.1%
w/waits and idles	16.2	NA	NA

Table 3: Execution Times for Procedures

Texture Classification in the Region Merging Stage.

Texture classification using the co--occurrence matrices requires that values are retrieved from a lookup table. This is ideally implemented on a machine with local indexing as each PE may have a different index into the table. On such a machine, all accesses could be satisfied in one query. Unfortunately, on SIMD machines all PEs must access the same relative local memory location at any given time. This means that if the index values may take any one of *m* values then *m* queries into the lookup table must be made while selecting only the appropriate PEs to be active for a given access.

This artifact of SIMD architectures underscores the importance of the decision to use a 256 element one--dimensional co--occurrence matrix based on the difference between intensity values of pixels rather than the 256 x 256 two--dimensional matrix described in {Haralick, 1973 #4}. This reduces the size of the lookup table from 65536 to 256 for each texture. The algorithm defines three textures for a maximum lookup table of 768 elements. The number of accesses can be significantly reduced by detecting the last element in the lookup table for each texture that has useful information and collapsing all accesses beyond that index into one access. Doing so reduces the total number of accesses to the tables to 144 from 768 for the specific textures used in the algorithm. From table 3 it is evident that the code for the table lookup, while awkward, is not the bottleneck for this implementation of the algorithm.

Region Merging

This stage identifies adjoining regions that meet the criterion for merging and then merges them by weighted averaging the intensities on population counts and updating the merged regions to the new intensity value. Like the edge extension stage, the precision of operands is not an issue in this stage because region merging uses only integer values. All sizes of datatypes are required in this phase from bit through full integer values. Operands have been set to their minimum sized datatypes. However, this stage presents a unique requirement missing in the previous stages of the algorithm. Before, operations worked on strictly local information requiring only nearest neighbor communication. Here, global information about regions must be gathered to do the averaging of intensities for the merging process. This necessitates the gathering of intensity values and the number of representatives of each value and then broadcasting the new intensity value to all the PEs of the joined regions. This turns out to be the bottleneck in this section of the algorithm.

First, a word about how region merging is implemented in the parallel CAAPP. The initial segmentation is used to define the boundaries around regions. These boundaries are represented explicitly in the CAAPP through the partitionable Coterie network by breaking communication links between neighboring pixels that lie in different regions. Once the regions have been defined, all regions can simultaneously count the number of pixels in the region and sum the intensity values of the pixels. This is accomplished in a straight forward manner using the RoutePlus function to route and tally the pixel count and sum the intensities at a master PE selected for each region. A simple division is sufficient to calculate the average which is broadcast back to the members of the region.

The actual merging is accomplished by having each PE at a region boundary check all directions for a possible merge. PEs that satisfy the conditions of the algorithm as stated in section~\ref{region-merging} make a link in the Coterie network joining the two regions. The new regions average the intensities via the RoutePlus function and the entire process is iterated again. This is just one simple method, many others are possible. One is to only merge regions that want to merge with each other. This was implemented but suffered from too little merging due to circular dependencies between three or more regions (e.g. A to B to C to A again). This resulted from frequent equal values returned by the simple evaluation function used to determine if two regions should be merged. The segmentation algorithm in [Kohler, 1981 #5] used a much more complex evaluation function that had more possible values in its range which lessens the possibility of creating the circular dependencies stated above.

Performance of RoutePlus. Table 3 shows the run--times of the two phases of the region merging stage: texture classification and the actual region merging. Three columns are shown. The first gives the time for the code excluding the RoutePlus operation. The second column is the time including RoutePlus calls. Finally, the third column is the percentage of time each routine spends doing the RoutePlus function. The times for routing that include waits and idles were not measured and are marked as NA. The RoutePlus performed here is sparse in the number of PEs involved. This is explained in more detail below. RoutePlus run--times are data dependent and take approximately 12.4 ms per call for the house image. There are 3 calls in the texture classification routine and 12 calls during the actual region merging. The RoutePlus functionality is clearly the bottleneck in these routines.

In the algorithm only a sparse set of elements route information to a destination PEs in the merge routine. This is useful for reducing potential congestion. In this implementation, a single PE per pre-merge region is selected as a representative of the region to send the size of the region and the intensity level multiplied by the size. The destination PE adds the weighted intensities and divides by the sum of the sizes to determine the new intensity level of the newly formed region. The new size and new intensity value are broadcast to all PEs in the new region.

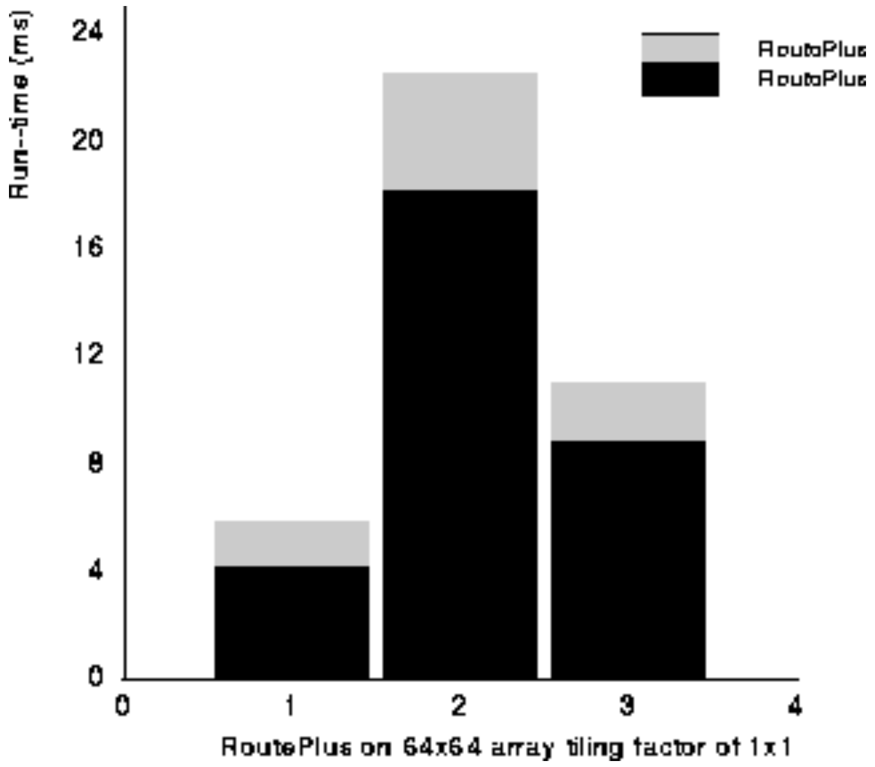


Figure 37: Execution time of RoutePlus on whole array, regions, sparse senders

Since it is the bottleneck, other implementations of the RoutePlus functionality should be looked at to speed up the function. One possible implementation of the RoutePlus functionality on the CAAPP includes doing each region serially using broadcasts and the very fast global feedback circuits in the array. This is unlikely to improve much on the time as there are often many hundreds of regions (over 1600 in the initial segmentation of the house image) and each broadcast takes about 500 cycles for bytes (to represent the intensity level). If there are over 500 regions we can expect this method to take greater than a quarter million cycles (25 ms) per use. This is about twice the current cost.

Another serial method on regions is to select the minimum and maximum intensities in the each region. For each intensity level between and including these two values select the PEs with that value and do a fast count of their number. Multiply the intensity value by the count and keep a running sum. A single divide of the total intensity by the total count gives the new intensity level of the region. There is the same dependency on the number of regions here as above and the select and multiply (assuming a byte value multiplied by a short value) take over 500 cycles. Again, this cost is significantly more than the current implementation.

A more promising approach may be to implement the RoutePlus at the ICAP level of the IUA. At this level there is a single powerful DSP chip for every 64 PEs at the CAAPP level. The ICAP processors can reduce the information from the 64 PEs and route the intermediate results to the ICAP processors responsible for the destination PEs. The time advantage of this scheme depends on the overhead of communication between the CAAPP and the ICAP processors as well as the communication efficiency within the ICAP layer itself.

Other approaches involve changing the design of the IUA. Some of these options are explored in [Herbordt, 1994 #3] that discusses algorithms for routing on reconfigurable meshes such as the CAAPP. Hardware enhancements studied were wider datapaths, ALUs, nearest neighbor communication paths, and a broadcast bus. The study shows a lower bound on speedup due to the required handshaking. The paper also mentions features such as the routing network in the CM-2 and local indexing as expensive enhancements that can be exploited for routing.

Recommendations. The ability to gather region-wide information easily is useful functionality for this algorithm. However, on the CAAPP this is very time-consuming and is the bottleneck in the algorithm's performance. The routing algorithm for the CAAPP is efficient given the hardware resources. Improvements in performance may have to come from exploiting the power of the ICAP processor network by using it as an intelligent routing network for the CAAPP. This is enticing since the ICAP has the wider datapaths, ALU, and communication paths that [Herbordt, 1994 #3] shows are important for accelerating routing. Implementating RoutePlus on the ICAP will be studied in future work.

Results

Table 4 compiles the run--times of all the stages of the segmentation algorithm. For edge detection 16 bit scaled integer times are used. It is clear that the RoutePlus functionality is the dominant factor in the overall run--time. Any algorithm that must gather statistics on a region-by-region basis will be very sensitive to the speed of the routing in the CAAPP. For the segmentation algorithm presented in this paper all efforts at optimization should be directed to speeding up the RoutePlus functionality. Novel methods of emulating the functionality with simpler, faster primitives needs to be explored as well as the possibility of using the ICAP.

Procedure Execution Times (ms)

Procedure	Run-Time Excluding RoutePlus	Runt-Time Including RoutePlus	Percentage in RoutePlus
Edge Extension	11.6	11.6	0%
w/waits and idles	15.2	15.2	0%
Texture Classification	4.5	4.5	0%
w/waits and idles	10.7	10.7	0%
Region Merging	18.1	204.1	91.1%
w/waits and idles	29.9	NA	NA
Total	34.2	220.2	84.5%
w/waits and idles	55.8	NA	NA

Table 4: Execution Times for Procedures

Future Work

Additional work that should be done includes:

• Try a simpler directional edge detector in place of the horizontal and vertical networks. I suspect that a technique such as the Sobel operator filtered through a threshold would work well on the test images. Cortes et al. point out that many simpler networks performed equally as well as theirs in the absence of blur. They considered only synthetic images. How significant is blur in the images that are of interest? There is a potential savings of 20% of computation if a simple edge detector proves sufficient.

• For texture classification the optimal filter should be calculated as specified in [Modestino, 1981 #6]. I suspect that the results may not be affected much. The types of textures Modestino et al. looked at were much more coarse than the test images and, as such, required a larger mask. The much finer textures in the test images will likely require a small mask so that the weighting of the entries may fall very rapidly from the center. The resulting optimal mask may be very similar to the mask currently being used. This hypothesis should be verified.

• A better control scheme to coordinate merging should be incorporated. Currently, the control scheme selects regions as they occur from left to right and top to bottom in the image and are merged according to the differences in intensities. Although quick, it ignores the fact that applying additional criteria as well as structuring the order of merges can improve the segmentation results. In [Kohler, 1981 #5], Kohler uses an evaluation function to calculate the best two regions to merge and iteratively selects pairs of regions until no more pairs match the minimum criteria. His use of a multi--term evaluation function gives more control over the merging process at the cost of additional computational complexity for calculating the parameters used in the evaluation function. However, the use of proper data structures should be able to minimize this overhead.

• RoutePlus is an important function but needs to be accelerated. Thus, an implementation using the ICAP level of the IUA must be explored.

Conclusion

An algorithm is presented that performs well on real outdoor images by using texture classification techniques to guide the use of multiple merging policies. Allowing mulitiple merge policies is a novel mechanism for region merging. In the example in the paper, texture classification allows an aggressive merging stategy to be applied to foliage--like regions to significantly reduce over--segmentation without destroying other structures in the image. Being able to selecively apply merging policies is shown to be a very powerful capability.

The algorithm is highly parallel and has been ported to the IUA CAAPP, a SIMD machine. Analysis of its performance shows that while a portion of the code is very sensitive to the arithmetic precision, the overwhelming majority of the run--time is spent gathering region statistics. On the given machine, the CAAPP, optimizing this functionality or rewriting the algorithm to eliminate its need is paramount. This algorithm requires that the hardware support the very efficient gathering of region-wide statistics across highly irregular regions. In the IUA architecture the most promising method to have this functionality may be at the ICAP level. This is being explored. It does require a low latency network at the ICAP level and very low latency communication with the SIMD CAAPP.

Appendix: Data Tables

This section details the cost of individual operations on the CAAPP SIMD array and the frequency of each instruction type in the segmentation algorithm. Not all operations available on the CAAPP are shown. Only those that are used in the algorithm are shown.

The cost of each operation is reported as a cost relative to the cost of doing an addition. Byte addition is defined as a cost of 1.00. These are approximate values from empirical measurements. Exact values can be gotten through detailed analysis of the microcode. However, the purpose of detailing these costs is only to get first order effects of operations on an application's performance. Using values from empirical measurement of high-level instructions has the advantage of folding in the effects of the many low-level instructions that implement each high-level instruction.

Since the CAAPP uses bit-serial processors, the cost of arithmetic operations depends on the precision being used. Table 5 shows the cost of doing an addition at each level of precision relative to a byte addition. The cost of a byte addition is 24 cycles. As expected, short integers and integers have almost two and four times the cost as a byte. Notice that there is some savings due to amortizing the overhead across more bits. The 32-bit floating point cost for addition is over 50 times the cost of byte addition. This is understandable in that floating point addition requires the mantissas to be aligned by comparing the operands' exponents, shifting the mantissa of the smaller operand, doing the addition, and then normalizing the result. This is significant work on the bit-serial CAAPP processor. The value for the type Float16 is an estimated cost for implementing a 16 bit floating point type on the CAAPP. This data type would give 8 bits of precision with an 8 bit exponent.

Please note that bit addition (or any bit logic operation) is only 30\% less costly than a full 8 bit addition due the internal 8 bit wide data path.

Addition Cost Normalized to Byte Operands

Bit	Byte	Short	Int	Float32	Float16
0.71	1.00	1.91	3.70	53.04	15.87

Table 5: Instruction Counts for Addition (Normalized to Byte Operands)

Table 6 lists the cost of each operation used in the algorithm relative to the cost of doing an addition at the given precision level. From the table we can see that multiplication and division are very expensive operations relative to addition. This is not surprising as the two operations are $O(n^2)$ while addition is only $O(n)$ where n is the number of bits in the word. Floating point is much more balanced across the operations since addition is very slow due to the shifting during the computation. Floating point addition requires over 1200 instructions compared to 24 for byte addition. Entries with NA are operations that are not available or were not measured.

Some operations are constant in time across the different datatypes. The Index operation requires 85 instructions to return the PE index ID. This is clearly independent of datatype. Another such operation is RoutePlus which sums the values of a plane in a region defined by the Coterie network. The cost of this operation should be dependent on the width of the data and the size of the array. It could be slightly improved for bit and byte datatypes. But, due to their constant run--time across datatypes, these instructions show marked decrease in their relative cost as the datatype gets larger reflecting the increase in cost of addition.

Below are the counts of each type of instruction for the procedures in the segmentation algorithm. Table 7 has the instruction count for four procedures. Network 1 refers to the procedures to find potential horizontal and vertical edgels. They are essentially identical. Network 2 refers to the final network of the algorithm that takes the potential edgels from Network 1 to generate actual edgels. The Edge Extension phase extends dangling edgels to complete regions and produce the initial segmentation. The two columns under this entry correspond to strictly bit precision operations and greater precision operations respectively.

Table 8 gives the instruction breakdown for the merging routine that selects the regions to merge. This excludes the call to the subroutine that performs the details of the actual merge, Union_All_Regions. The instruction count for Union_All_Regions is shown in table 9. Table 10 lists the instruction tally for the texture classification routine. The large values for the Select and comparison operations are due to the loop to look up values in the co-occurrence distribution tables for the various textures. Without local indexing support (very expensive to implement in hardware in a SIMD array) a Select must be done for every possible index value to set the activity bit on the appropriate PEs so actions may be selectively taken. There are three table of 256 entries each so a maximum loop of 768 iterations is possible. Using pointers to the last meaningful entry in each table has reduced the iteration count to 144 for the example textures. In contrast, local indexing would only require a PE to make only 8 accesses into each table for a total of 24 accesses. Local indexing, however, is extremely costly in resources to implement in a bit--serial SIMD array.

Operation Cost Normalized to Addition						
Operation	Bit	Byte	Short	Int	Float32	Float16
=	0.12	0.13	0.15	0.13	<0.01	0.01
+/-	1.00	1.00	1.00	1.00	1.00	1.00
multiplication	NA	11.22	19.16	33.93	1.51	1.49
division	NA	16.09	28.47	52.82	2.66	2.62
nearest neighbor	1.88	1.83	1.56	1.54	0.10	0.18
logical shift	0.12	3.35	3.07	2.38	NA	NA
approx. tanh	NA	4.43	3.86	3.51	0.38	0.59
Select (cmp)	2.94	2.35	1.42	0.86	0.18	0.29
Select (bit)	0.41	NA	NA	NA	NA	NA
AND, OR (var)	1.00	1.00	0.75	0.62	NA	NA
InsertBits	0.12	0.10	0.05	0.02	<0.01	<0.01
comparison	1.00	1.25	1.00	0.86	0.06	0.12
Index	5.00	3.54	1.93	1.00	0.07	0.23
RegionSelectMin/Max	7.41	130.43	69.77	35.29	2.46	8.22
RegionBroadcast	5.00	23.35	23.26	23.53	1.85	2.74
RoutePlus	2411.76	1783.61	953.49	482.35	NA	NA
Count	1.82	NA	NA	NA	NA	NA
Coterie pattern	0.88	0.63	0.34	0.18	0.01	0.40

Table 6: Instruction Counts for Operations (Normalized to Addition)

Procedure Operation Counts				
Operation	Network 1	Network 2	Edge Extension	
+/-	52	305		100
multiplication	51	60		
division		11		
nearest neighbor	42	84	320	140
logical shift	10	50		
approx. tanh	21	20		
Select (cmp)	1	50	170	128
Select (bit)				
AND, OR (var)			580	
comparison				120
Index				
RegionSelectMin/Max				
RegionBroadcast				
RoutePlus				
Count				
Coterie Pattern				

Table 7: Occurrences of Instructions in Listed Procedures

Merge Regions Excluding Union_All_Regions				
Operation	Bit	Byte	Short	Int
+/-		20		
multiplication				
division				
nearest neighbor		54		
logical shift		6		
approx. tanh				
Select (cmp)		179		
Select (bit)	10			
AND, OR (var)	313			
comparison	40	192	20	17
Index			5	5
RegionSelectMin/Max	5		5	5
RegionBroadcast			10	
RoutePlus				
Count				
Coterie Pattern				

Table 8: Occurrences of Instructions in Region Merging Excluding Union_All_Regions

Union_All_Region Ops				
Operation	Bit	Byte	Short	Int
+/-				
multiplication				
division				5
nearest neighbor				
logical shift				
approx. tanh				
Select (cmp)				5
Select (bit)	5			
AND, OR (var)				
comparison				5
Index		5		10
Row/Col Index			10	
RegionSelectMin/Max		5		
RegionBroadcast				
RoutePlus				10
Count				
Coterie Pattern	5			

Table 9: Occurrences of Instructions in Union_All_Regions

Texture Classification				
Operation	Bit	Byte	Short	Int
+/-				51
multiplication				
division				
nearest neighbor		8		36
logical shift				
approx. tanh				
Select (cmp)		2091	2	3
Select (bit)				
AND, OR (var)				
comparison		2091		3
Index			1	
Row/Col Index			2	
RegionSelectMin/Max			1	
RegionBroadcast			3	
RoutePlus			3	
Count				
Coterie Pattern				

Table 10: Occurrences of Instructions in Texture Classification