Blobs Learner: Improving in detecting omogeneously colored POIs in images

Fabrizio Nenci

April 10, 2013

Contents

1 What's this?

Two different algorithms for selecting colored points of interest in images' streams have been developed, then for one of them a genetic algorithm has been produced in order to find proper parameters.

This project is part of a program for detecting points of interest (*POIs*) in a stream of images. These POIs are supposed to be omogeneously colored areas.

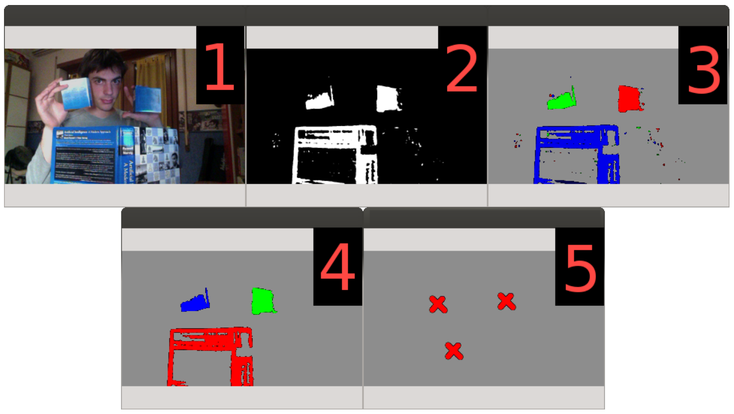


Figure 1: Detected POIs. Steps: 1)original image. 2)select pixels. 3)detect connected areas. 4)purge non important areas. 5)compute centroids.

As shown in figure 1, the interesting pixels are selected in the image, then connected areas (*blobs*) are extracted. Blobs that are not considered important are then pruned. The last step is to compute the centroids of the blobs that have not been discarded.

This project will examine the pixel selecting phase (step 2 in figure 1). Both algorithms select each pixel with a certain probability, while they differ in the way this probability is assigned.

2 Usage

When the program is launched, the user sees in a window the streaming from the webcam. Each time the user clicks on a point in the image, it means that he/she is telling the system that that point's color is interesting. The program then keeps learning from these clicks which are the points to be selected, and these are shown in another window.

3 From RGB to HSV

The image from the camera is usually loaded in a RGB format. Changing the value of a channel in this format can strongly affect the color we perceive. This is not very nice in this application, where the user wants the sistem to select pixels "similar"(from a human point of view) to the clicked ones.
s

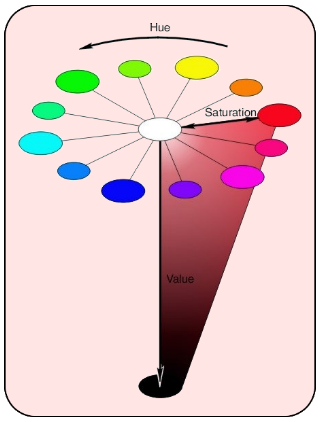


Figure 2: HSV color model dissected.

In order to have a more intuitive representation of the colors, the image is converted in HSV format. This is a quite comfortable color model for this application, since the "color", as we interpret it, is simply given by the *Hue* channel. In a certain way H indicates the dominant wavelength. The other two channels, *Value* and *Saturation*, respectively refer to the lightness and to the ratio of the dominant wavelength to other wavelengths (A full saturation means that you see the color, no saturation means you see a shadow of gray).

Both the developed algorithms will work on the HSV representation of the image.

4 Pixel selection: First Algorithm

In this section the first algorithm developed for this project is analyzed.

The three channels composing the color are considered separately. On each of them, each possible value is associated to a probability. When determining whether a pixel should be chosen or not, the probabilities associated to its three values are multiplied.

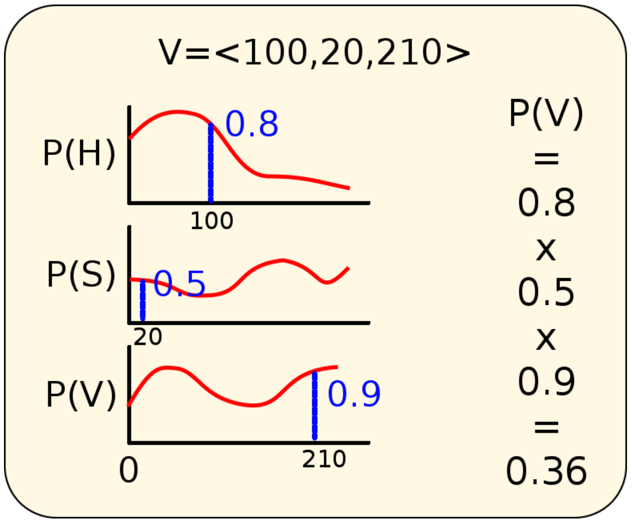


Figure 3: Algorithm 1: This example shows the computation of the probability that a point V is chosen. (The probability profiles are purely illustrative.)

Each time a new sample is provided (I.E. the user clicks on a point) the probability profiles are modified. The update procedure is this:

- Decrease all the probability values of each channel. Each channel has an assigned decay rate.
- Increase the probabilities corresponding to the clicked point values by a certain amount.
- On each channel, increase the probabilities associated to the values contained in a certain range from the clicked one. The probability is incremented of a quantity that depends on the "distance" from the clicked value. Close values' probabilities are incremented more than far values' ones.

This procedure for assigning probabilities has its good points. It behaves well when the number of samples is high, because the system does not need to check distances from previous samples.

On the other hand, this approach can suffer of "cross effects". An example will explain it. Let's consider two samples s1 and s2.

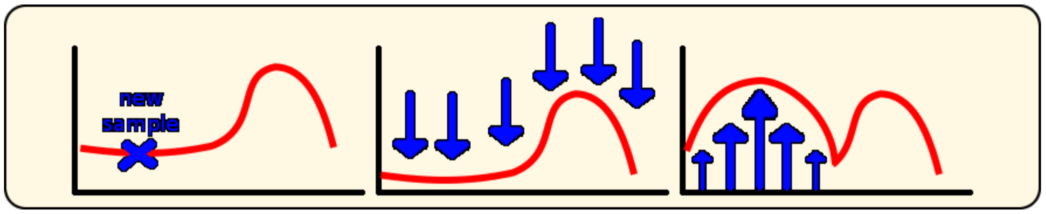


Figure 4: Algorithm 1: This illustrates the probability profile update of a channel when a new sample is added.

$s1 =< 200, 200, 200 >$, $s2 =< 50, 50, 50 >$.
These two samples are very different. They represent different colors and different lighting conditions. When evaluating the pixel $p =< 203, 46, 48 >$, we notice that its Hue is similar to that in s1, while its Saturation and Value are similar to those in s2. The probability profiles at those values will hence be high, so the composed probability will be high, even if the pixel p is not similar to s1 nor to s2. This may be an undesired effect.

5 Pixel selection: Second Algorithm

The second developed algorithm works in a different way.

In this algorithm, every sample provided is stored in a set. When evaluating a pixel, the set is scanned for the "closest" sample. The considered distance is a weighted sum of the three distances on the individual channels (I.E. a weighted variant of the Manhattan Distance). This way, a point that has the same values of a provided sample is considered at distance 0.

The probability that a pixel is selected depends on the computed distance. At distance 0 the probability is 1, at a certain threshold distance MAX_DIST (and over it) it is 0. The probabilities for distance values between 0 and MAX_DIST decrease with a quadratic shape.

This approach does not suffer of the "cross effects" that affect the other algorithm, but for every pixel a set of samples is scanned and distances computed. If the samples provided are many, the computation may become heavy.

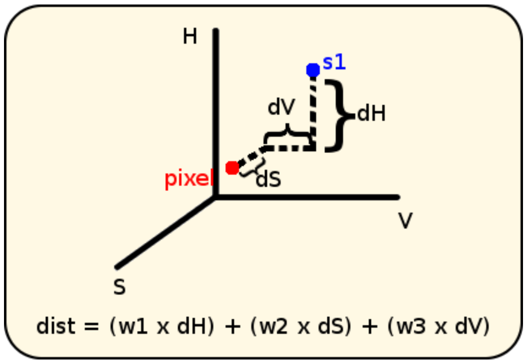


Figure 5: Algorithm 2: Computing the distance between a pixel and a sample.

6 Finding good parameters for the second algorithm

The second algorithm developed performs well on some homemade tests. As explained in section 5, this algorithm has some parameters that strongly influence its behaviour. These parameters are:

- The weights assigned to each channel when computing distances.
- The distance of influence of the samples.

It may be the case that, changing the place where the program is run, the different lighting conditions affect the importance of the H, S, and V components in the distance computation. For tuning these parameters, another genetic algorithm has been developed.

The training procedure is the following:
There is a set of images taken in the environment where the actual program will be run. This set is divided into two subsets, Ss and Ts, not necessarily disjoint. In the first gathering phase, Ss (Samples Set) is used for taking samples, that is:

- The user is shown each of the images in Ss
- The user **can** click on some points

Each clicked point is stored and will be used as a sample.
In the second gathering phase, Ts (Test Set) is populated. The images in

this set will be used for validating choices during the training. For preparing the test set:

- The user is shown each of the images in Ts
- The user **should** click once on each POI in the image, as close as possible to the centroid of the POI.

In this gathering phase, the H,S and V values of the clicked point are ignored, while their position in the image is stored and associated to the image. The system will interpret the overall gathering phase as "the user expects that, clicking on the points specified in the first gathering phase, I dectect all (and no more than) the POIs specified in the second gathering phase". Note how the training algorithm will take advantage of the negative informations that arise from the user non-clicking on an area.

It is now the case to make a clarification. As seen in the introduction, the outputted blobs are less than the detected ones. The simple rule for eliminating fake blobs is pruning away blobs that are smaller than a certain ratio to the bigger one. The threshold ratio has been left as another parameter of the Blobs Detecting algorithm. The training algorithm will then train also this parameter. Fixing the blobs size threshold could have led to human-computer misunderstanding, having the user clicking on two POIs that are in a ratio under the fixed threshold.

Here is a sketch of the genetic algorithm:

```
// definitions
BlobsLearner // the instance of the algorithm (charachterized
    by its parameters)
LIST_SIZE     // the number of BlobsLearners considered at
    each generation
GENERATIONS   // the number of generations to be produced

// global variables
BlobsLearner list[LIST_SIZE]  // The list containing the
    BlobsLearners
double fitness[LIST_SIZE]     // i-th element will contain the
     fitness value of the i-th BlobsLearner
       // the fitness representation is: the smallest the value
            is, the better the fitness. 0 is te best fitness.

// start the algorithm
read samples set;
read test set and associated images;
```

```
fill list with randomly generated BlobsLearners;

for GENERATIONS times
      set all the fitness values to 0;

      for each image 'img' in the test set
              for each BlobsLearner 'bl' in the fitness
                      found_centroids <-- ask bl to extract
                         the POIs from img (using the samples
                         set);
                      distance += compute the 'set_distance'
                         between found_centroids and the set
                         of centroids associated to img in the
                          test set;
              // end of 'for each BlobsLearner...'
      // end of 'for each image...'

      // at this point, fitness has been populated

      order list and fitness according to the values in
         fitness;

      // it is now time to produce the next generation

      BlobsLearner next_generation[LIST_SIZE];          // will
         put here the next BlobsLearners list

      // elitism: propagate the first (I.E. the best)
         BlobsLearner
      // two copies because one will mutate and one will
         remain equal
      next_generation[0] <-- list[0];
      next_generation[1] <-- list[1];

      // mating: produce new individuals combining the current
          ones
      for (i=2; i<LIST_SIZE-2; i++)
              randomly choose two BlobsLearners from list,
                 with a probability that is proportional to
                 their reversed fitness  // small values of
                 fitness are good, so they are associated to
                 higher probabilities of mating;
              next_generation[i] <-- a combination of the two
                 chosen BlobsLearners;
```

```
        // end of 'for (i=2...'

        // generate two new random elements
        next_generation[LIST_SIZE-2] =
            generateRandomBlobslearner();
        next_generation[LIST_SIZE-1] =
            generateRandomBlobslearner();

        // mutation: slightly change the elements of the next
            generation
        // NOTE: not applied to the first element
        for (i=1; i<LIST_SIZE; i++)
                next_generation[i].mutate(probabilities);
                    // mutate takes as input the probabilities
                    that each of the parameters mutates

        list = next_generation;

// end of 'for GENERATIONS times...'

// at this point, the first element in the list is the best
    fitting BlobsLearner
winner = list[0];
```

7 Further implementation details

The algorithm discussed in section 6 relies on some typical genetic operators, that have been adapted for the structures used in this project. In this section more details are given on each of these operators, and further implementation details are also analyzed.

7.1 Distance between two sets of points

In the genetic algorithm, the fitness is evaluated by accumulating the distances obtained comparing the set of centroids that the BlobsLearner detects in each image with the set of points indicated by the user for that image.

$$total\_distance(i) = \sum_k distance(compute\_set(i,k), test\_set[k])$$

It was then necessary to invent a way to compare two sets of points.

The method that compares the two sets takes as input:

- The two sets of points.
- The maximum distance*.
- A value 'extra' for extra points.
- A value 'lack' for lacking points.

**it is supposed that the points are extracted from the same image, so the diagonal of the image is the maximum distance that can separate two points.*

The method associates each point in the first set to its closest point in the second set (that is not associated to another one). The associations' combination found is the one that minimizes the overall distance between the associated points. Each point couple participates in the total distance with a contribute that is the two points distance divided by the maximum distance, hence the maximum contribute is 1. Each point in the first image that is not associated to a point in the second image gives a contribute equal to 'extra'. On the contrary, if the second set has more elements than the first, each of the overnumbering points gives a contribute equal to 'lack'.[1]

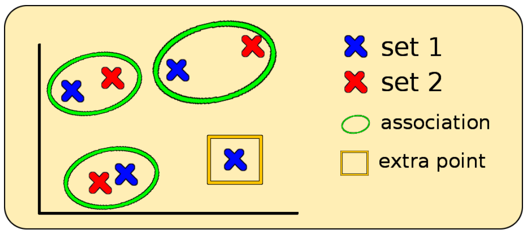


Figure 6: Example of associations when computing the distance between two sets of points.

7.2 Generating individuals

The parameters characterizing each individual are:

- The *Filter size*: it's the distance of influence of each sample.
- The *weights*: used to compute the distance.

[1]For performing experiments, both *extra* and *lack* have been set to 1.

- The *blobs size threshold*: used to prune small blobs.

In the various phases in which individuals are generated, these parameters are properly set/modified.

First of all, when **randomly** generating individuals, each of the parameter is set, with homogeneous probability, inside a certain interval. Random generation is used for filling the list at the very beginning, and for generating two random children at each generation. The values' limits used for this phase in the experiments are:

- Filter size: [1, 100].
- weights[i]: [0, 10].
- BS-Threshold: [0, 100].

Then, we have the **elitism** operator, that simply copies the best performing individual in the new list. This is to assure that the best result reached up to now is not lost.

The operator for generating children from two individuals is called **crossing over**(*mating*). The features of the child are a mix of those of the parents. In this specific case, the crossing over is performed by copying each of the parameters randomly from one of the two parents.

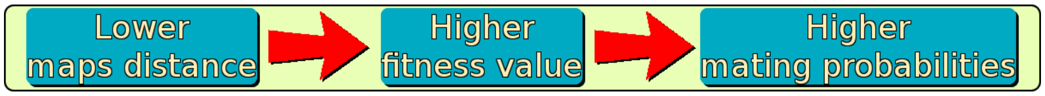
The very last operator used is **mutate**. This operator perturbates the current values of the parameters. For each of the parameters, a mutation's probability is given[2]. The perturbations' values used in the experiments are:

- Filter_size += f (f $\in$ [-20, 20])
- weights[i] += w (w $\in$ [-1,1])
- BS-Threshold = m*BS-Threshold (m $\in$ [0.5, 1.5])

7.3 Fitness function and choosing probabilities

As seen in section 7.1, the *quality* of a BlobsLearner is evaluated comparing the generated set of points and the one provided by the user. Being this measure a "distance", it will be smaller for better performances (overlapping maps will have distance 0).

[2]In the experiments, all the mutation probabilities have been set to 0.5



When choosing which individuals should have higher chances to generate children, anyway, it is common to refer to their **fitness**, assuming that the higher the fitness is, the better that element performs.

It was then necessary to reelaborate the distance measures in order to assign a fitness value to each individual. The procedure for computing fitnesses is the following:

```
input: distances[LIST_SIZE]
output: fitness[LIST_SIZE]

bigger_distance = max(distances[i]);
for each value 'd' in distances
        fitness[d] = bigger_distance - distances[d]
```

Note that the element corresponding to the bigger distance will be assigned a fitness equal to zero.

At this point, the fact that a greater fitness corresponds to higher reproduction chances is modeled in a roulette way: $prob(i) = \frac{fitness(i)}{\sum_k fitness(k)}$

8 Experiments with the genetic algorithm

Here is an example of the parameters tuning phase.

A small samples/test set has been created. It consists of the three images you see in figure 7.

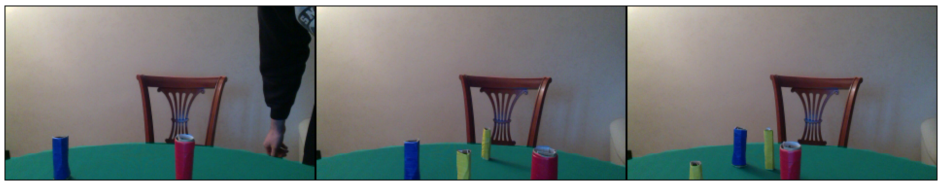


Figure 7: Images used for samples gathering and as test set.

For the samples gathering part, only three clicks were totally made: one on the red tube in the first image, one on the blue tube in the same image, and one on one of the yellow tubes in the second image.

For the test centroids gathering part, a click has been done on each of the colored tubes in each of the images, as close as possible to the *center* of the colored area.

Running the genetic algorithm on these data led to good results, in fact the found parameters performed very well on the same environment where the test set had been taken. In figure 8 you can see that the Blobs Learner correctly detects all the tubes, and nothing more than them. As samples it only got, again, one pixel of each of the desired colors, so this is the result after only three clicks. Note that the lighting conditions have slightly changed with respect to the images used to tune the parameters, but it works well anyway.

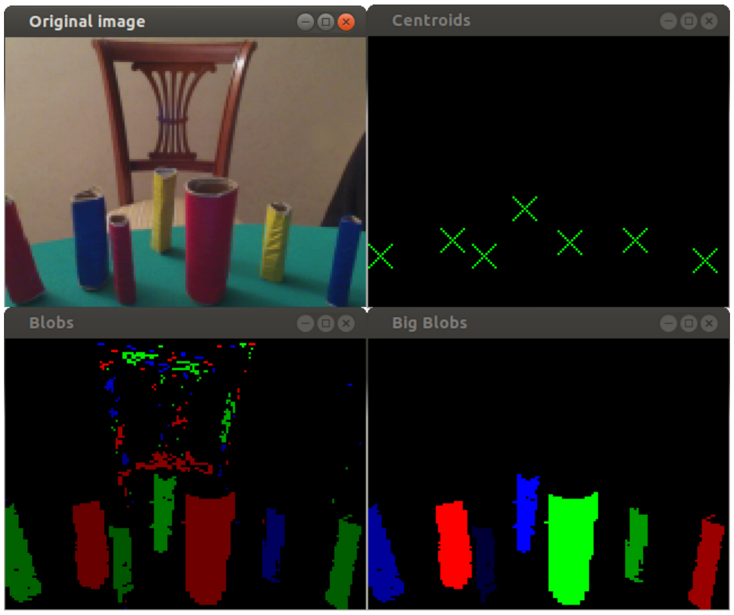


Figure 8: Running the second selection algorithm using the parameters found using the genetic algorithm.

Repetition of similar experiments in various environments shows that the weight assigned to the Hue channel is always much higher than the weights assigned to the Saturation and Value channels. This may be linked to the fact that, as seen in section 3, the Hue channel is somehow the *color* as we intend it.

9 Getting the software

The programs are downloadable from a repository on github. Just run:

```
git clone git://github.com/ieatmosquitos/blobs_learner.git
```

The compilation step is powered by cmake:

```
mkdir build
cd build
cmake ..
make
```

Note that OpenCV libraries are required.
The compilation will produce four executables:

- BlobsLearner - implementation of the BlobsDetector that uses the first algorithm for selecting pixels.
- BlobsLearner2 - implementation of the BlobsDetector that uses the second algorithm for selecting pixels.
- SamplesGathering - gather positions and HSV values from images for pixels specified by the user.
- WeightsFinder - use gathered informations for training the second algorithm.

There actually is another executable, namely "gather.sh". This is a script for running SamplesGathering on each of the images contained in the "images" folder.

Running the programs

BlobsLearner and **BlobsLearner2** take as input a file specifying the learner characteristics. For the first, the file can specify the filter size (I.E. the range of influence of the click action), the starting probabilities, and the decay rates. For the second, the file can specify the filter size (I.E. the range of influence of each sample), the weights to be assigned to the three channels when computing the distances, and the threshold size for pruning blobs. Two files have been prepared for the purpose: blob.cnf (for BlobsLearner) and blob2.cnf (for BlobsLearner2). Just edit them and launch:

```
BlobsLearner blob.cnf
```

or

```
BlobsLearner2 blob2.cnf
```

SamplesGathering wants as input the name of the image to open, and the text file where the new samples should be written. User will usually want to use the **gather.sh** script, that calls SamplesGathering on each of the files inside the directory “images”, and takes an optional parameter, that is the name of the file where the samples should be written (this file, if already existing, will be truncated). Hence, put images in the “build/images” folder and launch:

```
gather.sh samples.txt
```

WeightsFinder takes as input two files (generated by two separated runs of the gather.sh script). The first is taken as samples file, the other as test. Example:

```
gather.sh samples.txt
    // gather samples : click on the colors you want to
       provide as input
gather.sh test.txt
    // gather test: click once on each of the landmarks you
       assume that should be detected using the samples in
       ``samples.txt''
WeightsFinder samples.txt test.txt
```