**Autochipping Overview**

Before instance recognition (querying) can be performed, HotSpotter requires rectangular regions of interest (chips) to be defined. In the legacy product, this step was done manually and was very time-consuming. A primary improvement made by ECE 17.7 is autochipping. This uses pre-generated binary bitmaps showing motion (templates) in order to find chips in each image. Below is an example of an image and its corresponding template.



In order for autochipping to work, every image needs a corresponding template with the same name and a .bmp file type. The templates must exist in a folder titled ‘templates’ inside the folder containing the images. If these criteria are not met, the autochipping function does not know where to look for the templates, and HotSpotter currently does not have a method for handling missing images or templates.

Autochipping operates by systematically finding the largest (according to some maximization criterion) rectangle in a template, removing some part of that rectangle for future searches, and repeating until some stopping criterion has been met.

ECE 18.7 implemented multithreading/parallel processing in the AutoChipping module to decrease the runtime. Upgrading a single threaded process to take advantage of multithreading is traditionally a very complex computer science task. However, recent additions to Python provided time savings in the design and coding phase as well as in the final runtime of the application. After Beta testing and customer feedback, we tuned down the process multi-threading to use N-2 cores which still provides a dramatic speed increase while allowing the computer to be responsive to other tasks like web browsing or using other programs.

Python 2.7 provides the *multiprocessing* library which has a map function that allows the creation of a pool of “workers” to iterate over the list of images in parallel. This enables multiprocessor systems to perform much more efficiently when running the AutoChipping module

**Autochipping Modules**

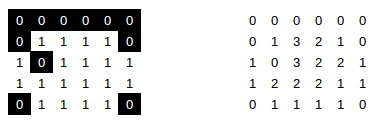
Autochipping consists of several modules:

* doAutochipping
* autochip
* findLargestRects
* findLargestSquares
* getTemplate
* getNumTemplates

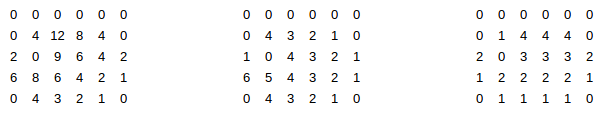
**doAutochipping** is essentially a driver for the package, and is what the HotSpotterAPI module uses to call autochipping. It handles top-level directory navigation, basic error handling, and packaging all the data for the HotSpotterAPI module to process and store.

**autochip** handles the chipping process: running searches for largest rectangles, removing part of each, and re-running until the stopping criterion has been met

**findLargestSquares** is called by findLargestRects, and must be discussed before it. findLargestSquares takes as an input a binary bitmap, with 0s corresponding with background and 1s corresponding with motion (a snow leopard). This module scans the template in reverse-raster order and records the side length of the largest square with the upper-left corner at each point. An example template and side-length map are shown below.



**findLargestRects** is the component that searches for and returns the largest rectangle in the template. It first calls findLargestSquares to create a side-length map, then creates two more maps with the largest width and height rectangles found. It then uses the maximization criterion to calculate the ‘size’ of the rectangle at each pixel by using these two new maps, and searches the maximization map for the largest of these. When area is to be maximized, the maximization map corresponds with the area of the largest rectangle with the upper-left corner located at that index. Below is the maximized value for each pixel, the maximum width, and maximum height of the above template, left to right.



**getTemplate** is simply a file reader that loads a bitmap into a matrix to be used in getLargestSquares.

**getNumTemplates** simply returns the number of templates at a given directory, for the purpose of error-checking.

**Autochipping Paramters**

The user-adjustable parameters of autochipping are:

* The amount of each chip removed for subsequent searches, called the exclusion factor
* The stopping criterion

The **exclusion factor** determines how much each chip is excluded in future searches, and may be any value from [0,1]. A value of 1 removes all of each chip for future searches, a value of .5 removes a rectangle with 50% of the width and 50% of the height (thus removing a rectangle with 25% of the original size from future searches, and a value of 0 removes only the center point of the rectangle. Chips with no overlap yield zero redundancy in data, but produces smaller chips, and chips with excess overlap produce many similar chips. Empircally, we found an exclusion factor of .75 to produce acceptable chips for our use.

The **stopping criterion** determines when the autochipping process stops for one image. It can be any value from (0,1) or any integer, and this decision changes the behavior of autochipping. If the stopping criterion is less than one, autochipping will continue until that ratio of pixels in the template have been captured in chips. Intuitively, this can be considered the percentage of the animal to be represented by chips, so a stopping criterion of .75 will result in at least 75% of the cat being captured. Conversely, if the stopping criterion is an integer, autochipping will remove that number of chips. We do not recommend setting the stopping criterion to an integer value for any case other than testing, as it will not produce optimal coverage for each cat. We have found through empirical testing that a value of about .6 tends to yield a few large chips that usually capture important identifying features of cats, such as flanks, necks, and hindquarters. We have not done extensive testing with tweaking this parameter.

Other parameters that can be changed in the source code include:

* Maximization criterion
* Minimum size
* Number of lines skipped per search for the largest rectangle

The **maximization criterion** is a length-3 list. The values of these numbers correspond to the weight applied to the rectangle’s height, width, and area respectively. We choose to calculate the largest rectangle based on area (with maximization criterion set to [0, 0, 1]), as this will produce larger chips than any other maximization criterion. We do not recommend using other values, as they tend to yield very narrow rectangles which are relatively useless for recognition.

The **minimum size** of a chip is defined as a length-2 list, with the minimum height and width respectively. The minimum size of chips defaults to a 1x1 square, but we find that this does not typically affect the autochipping process, since every template we have encountered is large enough to produce at least one meaningful chip.

In searching for the largest rectangle, the autochipping module scans every pixel in the maximization map to find the largest. The **skip length** parameter allows the module to skip a number of lines in its search, thus speeding up the process. By increasing the skip parameter, the process can be made faster at the expense of not guaranteeing the largest rectangle. Through early-stage testing, we found that a skip length of 8 allows for a significant increase in speed (16% of the original time) at the expense of an insignificant loss of chip size, so we recommend maintaining this value.