SEATTLE UNIVERSITY

Project Spot Check:

Developing a Vision-Based Program for Identifying Individual Snow Leopards from Camera Trap Photographs

User Guide to ECE 17.7's updated version of HotSpotter

Introduction

The following User Guide details both the manual automated processes within HotSpotter for use by Panthera biologists in Snow Leopard identification.

DISCLAIMER: Whether the user is working in Linux or Windows, be sure to keep the file structure intact the way it is when installing from the Linux shell file or the Windows executable. The script files expect a specific file structure and therefore the source files must be kept in the state they are in upon installation.

DISCLAIMER: This user guide does not detail installation of HotSpotter. If you have not yet installed HotSpotter, those instructions can be found on our GitHub setup repository: https://github.com/SU-ECE-17-7/ece177setup .

Manual Process

1. Open a database. ECE 17.7 has saved all of their test databases in /hotspotter/hstest/testdb . We recommend that you do this or opt to store your database files in a file directory that you trust.
2. Import Image(s). There are two options for this step: Importing specific image files or importing an entire image directory. After importing images, you can then interact with them by clicking on an image index.
3. "Chip out" your image. After clicking and opening an image, you can then define regions of interest within that image in which HotSpotter can use to compare snow leopards throughout the data set. Begin this process by opening an image and then pressing "A". This will allow you to click on two opposite points of your rectangular region of interest for HotSpotter to use as reference. Repeat this process as many times in one image throughout the whole dataset until you feel that there is a sufficient amount of chips.
4. Query a chip. Once all of the desired chips have been created, you can then continue on to querying. Querying is the process in which HotSpotter will compare an individual chip across the entire chip database and return the best chip matches. Start this by selecting a chip in your chip table and then pressing "Q". This will launch the Querying process and once finished, the results from that process will be visible in the "Query Results Table".

Automated Process

1. Open/Create a database. ECE 17.7 has saved all of their test databases in /hotspotter/hstest/testdb . We recommend that you do this or opt to store your database files in a file directory that you trust.
2. Import Image(s). ECE 17.7 recommends that for the automated process, you import an image directory. Our AutoChipping feature expects a subdirectory of bitmap templates for the image set within your image directory. This is required in order to run

AutoChipping.

3. AutoChip. This process automatically chips out an optimized number of chips from each image in your image table. Simply press the AutoChipping button and HotSpotter will chip each image. This process may take awhile depending on the amount of images in your database. The parameters for AutoChipping can be edited in
4. AutoQuery. This process runs a query for every chip in the database and obtains recognition scores for every chip. Simply press the “AutoQuery” button. Instead of displaying results in the Query Result table, all recognition scores are saved in a .csv file in your _hsdb folder. This .csv file is then used in the next process: Clustering.
5. Clustering. This process will take the recognition scores obtained from AutoQuerying and use a disjoint clustering algorithm to associate very strong matches with each other and group those matches into individual cats. Simply press the Clustering button. The output from this process is another.csv file containing cat identities with corresponding image names.

Detailed Explanation of Improvements made by ECE 17.7

Below you will find detailed explanations of our improvements, as well as some intuitive explanations of how some modules work “under the hood.” For more information about specific modules, please refer to the interface control documents (ICD) regarding that module contained in the technical report.

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.

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.

0	0	0	0	0	0
0	1	1	1	1	0
1	0	1	1	1	1
1	1	1	1	1	1
0	1	1	1	1	0

0	0	0	0	0	0
0	1	3	2	1	0
1	0	3	2	2	1
1	2	2	2	1	1
0	1	1	1	1	0

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.

0	0	0	0	0	0
0	4	12	8	4	0
2	0	9	6	4	2
6	8	6	4	2	1
0	4	3	2	1	0

0	0	0	0	0	0
0	4	3	2	1	0
1	0	4	3	2	1
6	5	4	3	2	1
0	4	3	2	1	0

0	0	0	0	0	0
0	1	4	4	4	0
2	0	3	3	3	2
1	2	2	2	2	1
0	1	1	1	1	0

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 Parameters

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.

AutoQuerying Overview

Autoquerying is the process where we create a recognition/association matrix that represents the relationships between chips. It consists of recognition and population of a score matrix. We begin by generating a matrix by querying every chip against other chips. When each chip is queried we have to compare that one chip to all other chips in the chip table. This was set up so that it would be in parallel, however this was giving us many errors so we decided to move to doing this in series. We simply forced HotSpotter to perform computations in series, thus serving as a sort of band-aid. We have changed the file structure somewhat since this change, so there is a chance that parallel feature computation is possible now. This computation is called querying in HotSpotter, but is a form of instance recognition, so these terms will be used interchangeably throughout this document and elsewhere.

The recognition algorithm operates on rectangular regions of interest known as chips. Within these chips, it compares elliptical areas of similarity known as hotspots, or more generally, patch-based features.

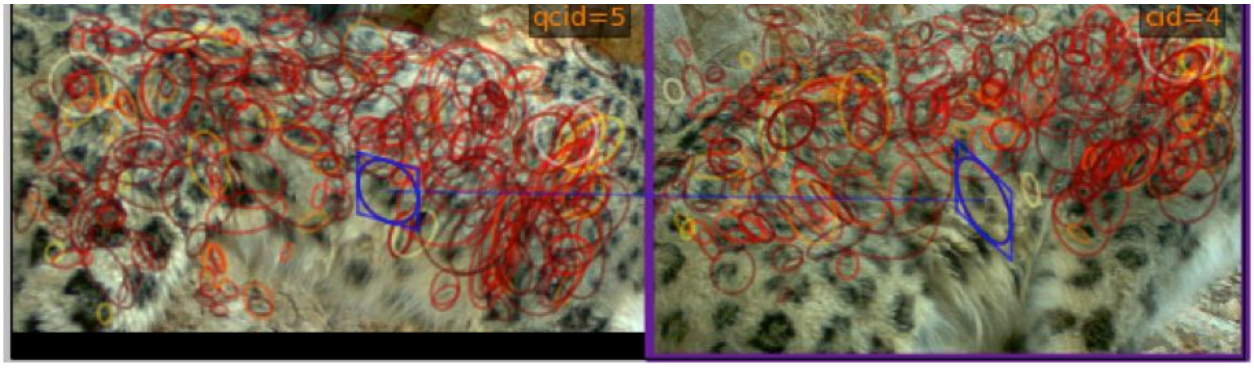

Figure 1: Example of hotspots on the body of a snow leopard

The keypoint associated with each feature contains information about xy-location of the feature within the image, as well as scale, orientation, and shape.[1] This additional information allows the program to account for the size of the feature, as well as different angles of viewing the animal and different poses, effectively allowing recognition between one sighting of an animal and another. However, since it is not desirable that hotspots be allowed to transform freely in reference to nearby hotspots, HotSpotter implements a process known as spatial verification. Spatial verification limits the potential chip-to-chip feature matches by ensuring that hotspots transform similarly to their neighbors. Any feature match that is found to be spatially inconsistent is not considered when computing a similarity score.

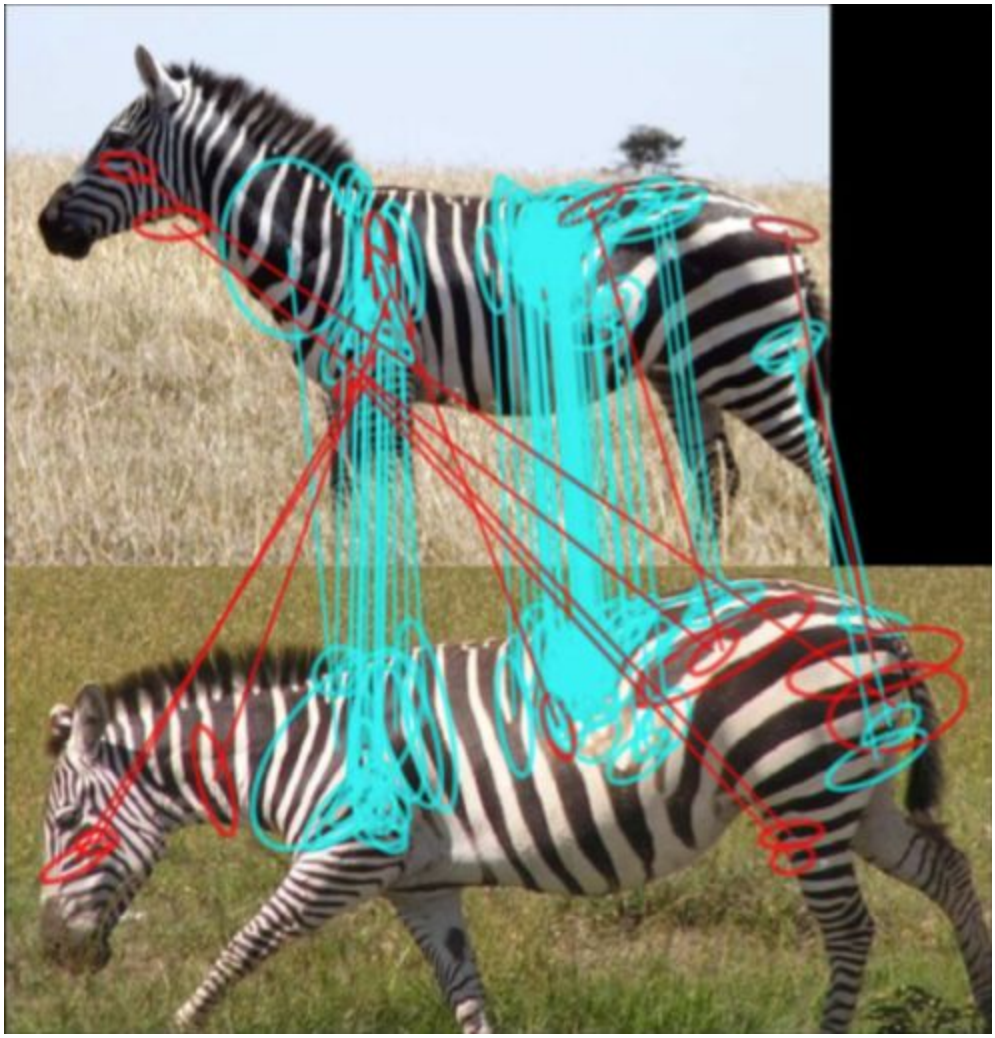

Figure 2: Example of spatial verification on a zebra image. Inconsistent matches are shown in red

Spatial verification works by fitting a rigid projective transformation to the chip. Currently, spatial verification only checks spatial location. The actual recognition is performed using an algorithm similar to Local Naïve Bayes Nearest Neighbor search [1].

The HotSpotter recognition algorithm is well tested on animals with relatively rigid bodies, e.g. zebras. However, snow leopards present a challenge in that their bodies are particularly lithe and flexible, and their fur is long. This fluffiness can sometime occlude identifying features such as spots. The effect these constraints will have on recognition accuracy is unclear, and thus it is necessary to thoroughly test the accuracy of the existing algorithm and determine if modifications are necessary to improve performance. This will involve testing on many cases of snow leopard images (e.g. cats at varying distances from the camera, levels of lighting, body position, etc.) and recording the results. Problem cases will be noted, and from this information, as well as knowledge of the recognition algorithm, the team will determine if modifications to improve the accuracy of the recognition algorithm are possible. Some possible candidates that

are under consideration are modifications to the spatial verification, e.g. expanding it to check additional dimensions (possibly scale and orientation), or modifying the transformation it uses to map hotspots, possibly implementing a non-rigid projective transformation.

To automate HotSpotter we created created functionality in autoquery.py that walks through the whole chip table and queries a chip against every other chip. This is done using functions listed in the MCL/Parsing ICD. However if there are no keypoints in a chip then it will return a string for the match between the two chips. We correct for this by assigning all strings to zero. HotSpotter however does not give the same scores from chip to chip. For example when querying chip 1 we might find chip 2 matches with a score of 500, but when chip 2 is queried we may find that chip 1 gets a matching score of 50. To cluster these chips we need an undirected graph. We create an undirected graph by averaging the scores from the queries and putting them into the score matrix. We must also take into account that a high score in one query may not correlate to a better match than a lower score in another query. To adjust as best that we could we decided that we would normalize all scores from an individual query based on the highest score from that query. That way we would not have such large changes in values as we would get if we did not do this. After talking to Dr. Miguel and our faculty advisor Rana Bayrakçismith, we were told that we can know with a fairly high certainty that if images were taken within 90 seconds from each other that the snow leopard in that image are the same snow leopard. WIth this knowledge we decided that we would add to the recognition scores if the chips were taken from images within 90 seconds or if chips came from the same image.The image name we are provided with from Panthera has metadata that is explained in MCL/parsing ICD. We make sure however that by adding this we do not let the score go above 1 because we want the entire matrix to be normalized. The parameters for same set and same image can be changed under options -> edit preferences. Extensive testing still needs to be done to try to make HotSpotter as accurate as possible and go through with large databases which ECE 17.7 did not have time to do. Also it may be worth looking into not averaging scores and taking the higher or lower of two scores. Querying was not thoroughly tested enough so we cannot say if the scores entered are completely resembling what their actual matching strength relative to all other queries.

---

[1] HotSpotter uses a Hessian-based keypoint detector, which detects blobs

Clustering Overview

Once we have a score matrix created from querying every chip we must then analyze all of the information we have created. We do this by using our score matrix as an undirected weighted graph and run a clustering algorithm on it. This means that we strengthen strong bonds and weaken weak bonds until we have disjoint clusters which represent individual snow leopards. In the MCL/parsing ICD we explain the process of MCL and how we create clusters. The main parameters that should be adjusted are the inflation parameter and the max loops parameter. The inflation parameter makes clusters tighter or looser. The lower the inflation number the fewer clusters the larger the number the more clusters. Max loops is the parameter that says the max amount of times that MCL is allowed to run. Running for more loops will ensure that the clusters will eventually become disjoint. These parameters need to be thoroughly tested and determined for the larger data sets that Panthera will be using. There are other disjoint clustering algorithms that may work better but we did not have time to develop and test other algorithms however. One of the requirements that Panthera had asked ECE 17.7 to accomplish that was unfeasible with time is creating folders of images that belong to individual snow leopards. This can be accomplished in the clusters to output function in mcl_clustering.py where we already determine clusters.