Oxford Internet Institute, University of Oxford

Assignment Cover Sheet

Candidate Number *Please note, your OSS number is NOT your candidate number*	1051177
Assignment *e.g. Online Social Networks*	Data Analytics at Scale
Term *Term assignment issued, e.g. MT or HT*	MT
Title/Question *Provide the full title, or If applicable, note the question number and the FULL question from the assigned list of questions*	FINd Algorithm Report
Word Count	3433

By placing a tick in this box ☒ I hereby certify as follows:

(a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;

(b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1.

(c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml and that I agree to my work being screened and used as explained in that Notice;

(d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.

(e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].

(f) I have not sought assistance from a professional agency;

(g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

FINd Algorithm Report

1.1 FINd analysis

1.1.1 FINd overview

Find Images Now (FIN) provided an image hashing algorithm, widely used for image authentication and matching. Important applications include copyright protection through reverse image search tools and identification of potentially malicious images that match already identified content.

Image hashing algorithms contain two main stages: feature extraction and hash generation. During feature extraction, the FINd algorithm extracts grayscale values of each pixel within an image and subsequently utilizes a spatial domain filter to create a list of averaged grayscale pixel values. This list is turned into a 16x16 matrix from which an image hash is generated. Similarity between images is identified by calculating the Hamming distance of the generated hashes.[1] The lower the distance, the more likely the pictures are to be similar.

Feature extraction is crucial, as it determines the accuracy of the image matching. It is also likely to be most computationally involved, as it reduces an image with, say, 60.000+ pixels times three color values (i.e., RGB-values) into a 16x16 matrix. The hash generation merely utilizes the 16x16 matrix to generate an image hash. As such, this report will focus on reducing computational costs of the feature extraction.

Further, the FINd algorithm is a single-process algorithm. As such, memory usage will be limited to the memory taken up by one image. Although images can be large, they do not exceed the 8GB RAM or more of a modern-day computational machine or server. Additionally, the FINd algorithm applies an upper boundary on the input image size by reducing larger-sized images to thumbnails thereof with a longest side of 512 pixel. Thus, memory optimization will not be a priority of this report. In contrast, optimization will focus on using more of the available working memory by implementing multi-processing. With a typical computational machine having 4-8 CPUs, using all available CPUs simultaneously will not run into memory problems, too.

1.1.2 Line profiling

Appendix 1 includes the line profiler of the original `FINd.py` script.[2] Calling the `fromFile` function runs the `FINd.py` script. 99.1% of the runtime of this function is taken up by the `fromImage` function. `fillFloatLumaFromBufferImage` and `findHash256FromFloatLuma` are the two functions taking up 9.6% and 90.3% of the run-time of `fromImage` respectively. Thus, together `fillFloatLumaFromBufferImage` and `findHash256FromFloatLuma` take up around 99% of the overall run-time of the FINd algorithm.[3] This report will focus on either to showcase improvements in efficiency due to reductions in the number of for-loop iterations and replacing for-loops entirely through scientific Python libraries.

---

[1] Calculating the Hamming distance involves counting the number of bit positions that are different between two hashes. For further information on calculating the Hamming distance, see the Wikipedia article on Hamming distance.

[2] For simplicity, only functions significantly increasing the runtime are included.

[3] This has been calculated in the following way: (proportion of run-time from `findHash256FromFloatLuma` in `fromImage` + proportion of run-time from `fillFloatLumaFromBufferImage` in `fromImage`) times the proportion of run-time from `fromImage` in `fromFile`, which equals $(0.903 + 0.096) \times 0.991 \approx 0.99$.

Starting with `findHash256FromFloatLuma`, 98.7% of the time spent in this function comes from the `boxFilter` function, which is self-contained.[4] In `boxFilter`, most of the runtime is spent in two for-loops, without actually involving computationally heavy calculations. Instead, the sheer number of iterations within the given for-loops increases runtime. The `boxFilter` function contains four for-loops in total: two outer and two inner for-loops. The outer for-loops loop through the grayscale values of each pixel in the image, which totals to 62500 iterations for an image with a simple 250x250 pixel dimension. Thus, for this example, all inner calculations are done at least 62500 times. The inner for-loops perform a sliding-window pixel value averaging operation. Further down the workflow of the algorithm, these averaged values are used to create a 64x64 matrix within the `decimateFloat` function. When inspecting `decimateFloat`, however, note that although the output of `boxFilter` corresponds to the total number of the averaged grayscale pixel from the input image, only a fraction ($N = 4096$, i.e., 64×64) of those values is used to create the 64x64 matrix in `decimateFloat`. Thus, the `boxFilter` function calculates extraneous data by looping through grayscale values that will not be included in subsequent calculations. Consequently, reducing the number of iterations within the `boxFilter` to match the input of `decimateFloat` directly will be an opportunity to reduce the runtime of the FINd algorithm.

Next, almost the entire remainder of time spent in `fromImage` is taken up by the function `fillFloatLumaFromBufferImage`.[5] This function extracts the grayscale values of each pixel. Again, there is a pair of for-loops iterating through all pixel values for each of the RGB-colors, meaning that the number of calculations involved equals three times the number of pixels per image. Instead of looping through each pixel to extract the RGB values, there exist scientific Python libraries that can sum the RGB values of each pixel to a grayscale value through vectorization. This will be a further opportunity to optimize `FINd.py` as a single-process algorithm.

1.1.3 Memory profiling and multi-processing

Although the FINd algorithm is a single-process algorithm, its calculations are embarrassingly parallel, meaning that all outputs are entirely independent from each other. Further, as the bottleneck encountered by FIN is a computational instead of an I/O one (which would suggest a multi-threading approach), multi-processing is a potential solution for further reducing the runtime of the FINd algorithm without changing the final output. The extension to a multi-process algorithm makes use of the capacities of modern-day computational machines with several CPUs. Given that the FINd algorithm is likely to run on a computational machine solely devoted to image hashing, utilizing all CPUs will be an optimal resource-usage.

A further reason for extending the single-process operation into a multi-process one is that the memory usage involved in the FINd algorithm is limited to the size of maximum a 512x512 image thumbnail. Appendix 3 shows that this memory usage lies just above 100 megabytes, while modern machines usually have a 8GB RAM and beyond.[6] Thus, even when parallelizing processes 8 times (given that a machine has 8 CPUs), effectively using up 8 times as much memory, this would not come close to using up all working memory. Given, again, that the FINd algorithm is likely to run on a machine solely devoted to image hashing, it is likely that the RAM of this machine will not be used alternatively either. Thus, multi-processing will be an optimization implemented.

---

[4] The location of this operation is highlighted in green in Appendix 1.
[5] The location of this operation is highlighted in yellow in Appendix 1.
[6] These results may be replicated in `FINd_example.ipynb` in section 'Original FINd algorithm'.

1.2 FINd optimizations

1.2.1 Single-process optimization[7]

Following 1.1.2, this report will implement a reduction in the for-loop iterations within the `boxFilter` function as well as an entire replacement of the for-loops in the `fillFloatLumaFromBufferImage` function.[8]

The outer for-loops of the `boxFilter` function can be replaced by the for-loops of the `decimateFloat` function without changing the results, because the for-loops of `decimateFloat` correspond to the pixel values subsequently used for hash generation. Consequently, the `boxFilter` function only implements a sliding-window algorithm for a subset of 64x64 pixel values. The output of the modified `boxFilter` function corresponds directly to the output of the `decimateFloat` function, i.e., a 64x64 matrix instead of a list of length equal to the number of pixels in the input image. This implementation effectively reduces the number of for-loop iterations to less than 10% of the original level, all the while including the calculations performed originally by the `decimateFloat` function.

The for-loops of the `fillFloatLumaFromBufferImage` function have been replaced entirely by a vectorization process through `numpy` arrays and simple arithmetic. The input image is now converted to a `ndarray` featuring the RGB values for each pixel. From this `ndarray`, three arrays, one for each RGB color value, are extracted and multiplied by the respective grayscale coefficients. The three arrays are then added together and transformed into a flattened list so as to match the original function output.

1.2.2 Multi-process optimization[9]

As argued above, the FINd algorithm is embarrassingly parallel, meaning that this report will implement multi-processing of a slightly refined optimized single-process FINd algorithm from `myFINd1P.py` without running the risk of encountering race conditions. The slight alteration regards the output of the `myFINd1P.py` script, which additionally outputs the input file name instead of merely the output hash. This report will implement an optimization that avoids shared memory between the respective processes entirely, because it is not important to share information across processes, while sharing this information creates additional synchronization overhead. Specifically, this report implements a `ProcessPoolExecutor` with `map` attribute. This choice has been made due to an easy implementation for future use by FIN itself.

The implemented multi-processing algorithm automatically sets a chunksize equal to the number of images in the database divided by the number of parallel processes running. Setting the chunksize to the maximum possible optimizes performance as it minimizes overhead. A

---

[7] The code for this optimization can be run in the `FINd_example.ipynb` file under section 'Single-process Optimization' and corresponds to the `myFINd1.py` script. To be able to run this script, the script `matrix.py` will need to be in the same directory.

[8] The locations of each optimized function are highlighted in green and yellow respectively in Appendix 2.

[9] The multi-processing implementation is run from a separate Python script called `multiprocess.py` and refers to the modified single-process optimization `myFINd1P.py`. The output of `multiprocess.py` is stored in a dictionary with the keys being the input image name and the value being the output hash. The algorithm can be run in the `FINd_example.ipynb` file under section 'Multi-process optimization'. Again, the script `matrix.py` will need to be in the same directory.

potential drawback, however, is that although the chunks contain an equal or very similar number of input images, the images are of different size, and some Python interpreters will take longer to run than others, which are lying idle in the meantime. Since the FINd algorithm caps the input image sizes, however, this should not be a major worry.

1.3 Optimization comparison

1.3.1 Single-process results

The single-process optimization yields the same results as the original FINd algorithm while reducing and replacing a large chunk of the for-loop iterations within it by omitting extraneous calculations and implementing scientific Python libraries.[10] This reduction in for-loop iterations reduces the total runtime of the single-process optimization to less than 10% of the original FINd algorithm.

In specific, the time spent in `boxFilter` has been reduced almost identically in line with the reduction in for-loop iterations. The number of for-loop iterations has been reduced to just under 7% of the original level, and the time spent in the `boxFilter` function to 8%. This, again, stresses the importance of keeping the number of for-loop iterations as low as possible. The runtime of the `fillFloatLumaFromBufferImage` function has been reduced to just above 1% of the original value. This is largely due to optimized calculations using numpy vectorization.[11]

Additionally, the single-process optimization does not affect memory usage of the image hashing algorithm, as the input and output stay unchanged, and no further copies of the input have been added (see Appendix 3).

1.3.2 Multi-process results

The implemented multi-process optimization makes use of all the CPUs available on a computational machine. The performance improvements of multi-processing grow as the number of input files increases, as the proportion of time spent on dividing the input files into chunks and finally adding all the output hashes into a dictionary decreases. This is evident when comparing the time spent on processing 10, 100, or 3444 input images between the single-process and multi-process optimizations. Whereas for 10 input images, the single-process optimization outperforms the multi-process optimization, this result is inverted for 100 input images. For 3444 input images, this difference increases to a roughly fourfold speed-up (see Appendix 4).[12]

The main disadvantage of a multi-processing optimization is that each process requires its own memory, which means (i) that there is overhead in starting and stopping the processes and (ii) that the overall memory usage of the process scales O(N) with the number of processes run simultaneously. The significance of (i) dwindles as the chunksize increases. But (ii) may cause significant problems if too many processes are running simultaneously. As modern-day

[10] Unit tests in the `FINd_example.ipynb` file show that the output hashes from the original FINd algorithm and the optimized single-process algorithm are identical. These results may be replicated in `FINd_example.ipynb` in section ‘Single-process optimization’.

[11] These results may be replicated in `FINd_example.ipynb` in section ‘Single-process optimization’.

[12] These results may be replicated in `FINd_example.ipynb` in section ‘Single-process optimization’ and ‘Multi-process optimization’.

machines have several gigabytes of RAM, however, and only about 8 CPUs, this disadvantage should not cause severe issues.

2.1 FINd vs imagehash

This report continues to compare the multi-process optimized FINd algorithm to the average hashing and perceptual hashing approach of the `imagehash` library.

Average hashing works as follows. To extract image features, the algorithm reduces the size of any input image to an 8x8 square, reduces the colors to grayscale and averages the grayscale values. To generate the image hash, it assigns Boolean values to each pixel depending on whether it is larger or smaller than the average pixel grayscale value. To compare two images, it calculates the Hamming distance.

Perceptual hashing is similar. To extract image features, it first reduces the size of any input image to a 32x32 square and reduces the colors to grayscale again. Next, the algorithm transforms the grayscale pixel values using a discrete cosine transform (DCT), to achieve a more robust image classification later on.[13] A subset of 8x8 transforms is taken and averaged. During hash generation, Boolean values are given to each of the 8x8 transforms depending on whether they are larger or smaller than the average transform. Again, to compare two images, it uses the Hamming distance.

2.1.1 Computational comparison

The performance comparisons include runtime and memory usage. First, the memory usage for each single-process algorithm is identical, as can be seen in Appendix 3.[14] However, since the multi-process optimization makes use of several Python interpreters, it uses as many times as much memory as the `imagehash` algorithms, as the number of Python interpreters utilized. Thus, the `imagehash` algorithms outperform the multi-process optimization regarding memory usage.

Regarding runtime, Appendix 4 shows that, although the multi-process optimization makes use of parallelization, while the `imagehash` algorithms constitute single-process operations, they are around 10x faster. When comparing the speed of each `imagehash` algorithm, the average hashing algorithm is about 15% faster than the perceptual hashing algorithm. [15]

2.1.2 Classification comparison

To compute the classification performance of each algorithm, this report picks out a representative image from each image family from the “meme generator” dataset and computes the Hamming distance for every other image. If the Hamming distance for a certain image hash is below a certain threshold, then the related input image will be classified as belonging to the respective image family.[16] The results of this computation are in Appendix 5 and include the first ten image families of the “meme generator” dataset, i.e., 3444 input images.

---

[13] For further information on DCT, see the Wikipedia article on DCT.
[14] These results may be replicated in `FINd_example.ipynb` in section ‘Imagehash library’.
[15] These results may be replicated in `FINd_example.ipynb` in section ‘Imagehash library’.
[16] The code for this implementation can be found in `FINd_example.ipynb` in section ‘Algorithm classification’

The most general metric of classification performance is accuracy, i.e., how often the classifier is correct, meaning that it predicts images to belong to the same image family when they actually do belong to the same family and vice versa for images not belonging to the same family. More detailed metrics are the True Positive Rate (TPR),[17] True Negative Rate (TNR),[18] and precision.[19]

Starting with general classification performance, the accuracy values indicate that the multi-process optimization of the FINd algorithm performs the best, correctly predicting over 99% of the input images. The perceptual hashing algorithm performs slightly worse (below 99%), and the average hashing algorithm performs worst (98%). Nevertheless, all algorithms are performing well on accuracy, given that not all image labels in the "meme generator" dataset are correct.

A slightly different picture results from the more nuanced classification performance measures. While the multi-process optimization correctly detects over 97% of the similarity, perceptual hashing misses out on more than 5% of similarity and average hashing on 7%. Additionally, while the multi-process optimization practically correctly classifies all dissimilarity, perceptual hashing misses out on 0.3% of the dissimilarity and average hashing on over 1% of the dissimilarity. Together, these measures lead to precision values of close to 1 for the multi-process optimization, about 98% for perceptual hashing, and only about 95% for average hashing. As such, the average hashing and perceptual hashing algorithms include a significant amount of noise in their output.

2.2 Discussion

The task at hand is to match and cluster images at scale. This requires speed, low memory usage and correct classification simultaneously. The algorithms above represent a selection of options that can strike a balance among all metrics or score extremely high on a subset.

Starting with the multi-process optimization of the original FINd algorithm. It scores highest in terms of accuracy among all three options. It correctly classifies most of the similarity between images, while it effectively entirely excludes all dissimilarity. As such, the results of the optimization are likely to be very robust, i.e., include images that are practically similar, but have a slightly different color scheme, have been resized, or added text, all the while excluding dissimilar images. The major disadvantages of that algorithm are, however, that it is by far the slowest option and uses up significantly more memory. While the memory usage should not present a major issue, as it is far below modern-day RAM, the slow speed of the algorithm makes effective search across large image samples difficult.

The average hashing algorithm scores the lowest in terms of accuracy. It misses out on a significant portion of similarity and includes a considerable amount of noise into the predicted similarity. On the flipside, however, it is the fastest option, making for an efficient but not entirely robust image matching tool.

The perceptual hashing algorithm strikes a balance between the above two options. First, it is more robust than average hashing, but not as robust as the FINd optimization. It still does not detect 5% of the similarity, about 2.5% less than the FINd optimization, and adds a significant

[17] TPR represents the number of correct similarity predictions divided by the total amount of similarity.
[18] TNR represents the number of correct dissimilarity predictions divided by the total amount of dissimilarity.
[19] Precision calculates the number of correct similarity predictions divided by the total number of similarity predictions.

amount of noise into its prediction outcomes. Nevertheless, it is merely 15% slower than the average hashing algorithm, which still makes it about 9x as fast as the FINd optimization while running on an eighth of the memory.

Consequently, both the average hashing and perceptual hashing algorithm present a significant speed-up of the FINd algorithm at the cost of robustness. Both algorithms use up less memory than the optimized FINd algorithm as well. As such, they are likely more beneficial for speedy image matching that does not require the most robust results. Of the two `imagehash` algorithms, perceptual hashing is the preferred option, because it increases robustness significantly compared to average hashing while only decreasing speed by 15%. Nevertheless, as the FINd algorithm has only been partly optimized, with room left for runtime improvement, it should not be dismissed and further research into optimization should be devoted to it, as it presents the most robust results.

To see why both robustness and speed matter, consider reverse image search tools such as Google Images. Tools with very bad results in either speed or robustness or both are likely to be unpopular with users. The perceptual hashing algorithm will impress on speed while it is likely to include noise in its output and miss out on some relevant content. The FINd algorithm, in contrast, will only include relevant content while it presents the most variation in the search image to the user but make them wait for too long.

Depending on the required speed and robustness of the output, perceptual hashing and the multi-process optimization offer viable solutions. All in all, however, this report recommends the use of perceptual hashing as it returns relatively robust results while using only a fraction of the runtime compared to the optimized FINd algorithm. Nevertheless, further optimization of the FINd algorithm should be considered, as it returns the most robust results.

Appendix 1

Line profiler results for initial `FINd.py` script. Highlighted lines correspond to function calls that have been optimized. Timer unit: $1e-6s$.

```
Function: fromFile at line 39
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    39                                           @profile
    40                                           def fromFile(self, filepath):
    41         1          4.0      4.0      0.0                  img = None
    42         1          0.0      0.0      0.0                  try:
    43         1      41543.0  41543.0      0.9                          img = Image.open(filepath)
    44                                           except IOError as e:
    45                                                   raise e
    46         1    4334211.0 4334211.0     99.1                  return self.fromImage(img)


Function: fromImage at line 48
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48                                           @profile
    49                                           def fromImage(self,img):
    50         1          1.0      1.0      0.0                  try:
    51         1       1759.0   1759.0      0.0                          img=img.copy()
    52         1         23.0     23.0      0.0                          img.thumbnail((512, 512))
    53                                           except IOError as e:
    54                                                   raise e
    55         1          1.0      1.0      0.0                  numCols, numRows = img.size
    56         1        285.0    285.0      0.0                  buffer1 = MatrixUtil.allocateMatrixAsRowMajorArray(numRows, numCols)
    57         1        280.0    280.0      0.0                  buffer2 = MatrixUtil.allocateMatrixAsRowMajorArray(numRows, numCols)
    58         1         46.0     46.0      0.0                  buffer64x64 = MatrixUtil.allocateMatrix(64, 64)
    59         1         10.0     10.0      0.0                  buffer16x64 = MatrixUtil.allocateMatrix(16, 64)
    60         1          7.0      7.0      0.0                  buffer16x16 = MatrixUtil.allocateMatrix(16, 16)
    61         1         22.0     22.0      0.0                  numCols, numRows = img.size
    62         1     417411.0 417411.0      9.6                  self.fillFloatLumaFromBufferImage(img, buffer1)
    63         2    3913615.0 1956807.5     90.3                  return self.findHash256FromFloatLuma(
    64         1          1.0      1.0      0.0                          buffer1, buffer2, numRows,
                                                                     numCols, buffer64x64, buffer16x64, buffer16x16
    66                                                               )


Function: fillFloatLumaFromBufferImage at line 67
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    67                                           @profile
    68                                           def fillFloatLumaFromBufferImage(self, img, luma):
    69         1          1.0      1.0      0.0                  numCols, numRows = img.size
    70         1        122.0    122.0      0.0                  rgb_image = img.convert("RGB")
    71         1          0.0      0.0      0.0                  numCols, numRows = img.size
    72       251         76.0      0.3      0.0                  for i in range(numRows):
    73     62750      18679.0      0.3      6.4                          for j in range(numCols):
    74     62500     136155.0      2.2     46.7                                  r, g, b = rgb_image.getpixel((j, i))
    75     62500      25732.0      0.4      8.8                                  luma[i * numCols + j] = (
    76    187500      64051.0      0.3     22.0                                          self.LUMA_FROM_R_COEFF * r
    77     62500      22688.0      0.4      7.8                                          + self.LUMA_FROM_G_COEFF * g
    78     62500      24025.0      0.4      8.2                                          + self.LUMA_FROM_B_COEFF * b
    79                                                                           )


Function: findHash256FromFloatLuma at line 81
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    81                                           @profile
    82                                           def findHash256FromFloatLuma(
    83                                                   self,
    84                                                   fullBuffer1,
    85                                                   fullBuffer2,
    86                                                   numRows,
    87                                                   numCols,
    88                                                   buffer64x64,
    89                                                   buffer16x64,
    90                                                   buffer16x16,
    91                                           ):
    92         1          6.0      6.0      0.0                  windowSizeAlongRows = self.computeBoxFilterWindowSize(numCols)
    93         1          1.0      1.0      0.0                  windowSizeAlongCols = self.computeBoxFilterWindowSize(numRows)
    94
    95         1    3864339.0 3864339.0     98.7                  self.boxFilter(fullBuffer1,fullBuffer2,numRows,numCols,
                                                                     windowSizeAlongRows,windowSizeAlongCols)
    96         1          0.0      0.0      0.0                  fullBuffer1=fullBuffer2
    97
    98         1       8377.0   8377.0      0.2                  self.decimateFloat(fullBuffer1, numRows, numCols, buffer64x64)
    99         1      38150.0  38150.0      1.0                  self.dct64To16(buffer64x64, buffer16x64, buffer16x16)
   100         1       2722.0   2722.0      0.1                  hash = self.dctOutput2hash(buffer16x16)
   101         1          1.0      1.0      0.0                  return hash



Function: decimateFloat at line 103
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   103                                           @classmethod
   104                                           @profile
   105                                           def decimateFloat(
   106                                                   cls, in_, inNumRows, inNumCols, out  # numRows x numCols in row-major order
   107                                           ):
   108        65         12.0      0.2      0.2                  for i in range(64):
   109        64         37.0      0.6      0.7                          ini = int(((i + 0.5) * inNumRows) / 64)
   110      4160       1247.0      0.3     23.5                          for j in range(64):
   111      4096       1993.0      0.5     37.5                                  inj = int(((j + 0.5) * inNumCols) / 64)
   112      4096       2019.0      0.5     38.0                                  out[i][j] = in_[ini * inNumCols + inj]
Function: boxFilter at line 171
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   171                                           @classmethod
   172                                           @profile
```

173					def boxFilter(cls,input,output,rows,cols,rowWin,colWin):
174	1	0.0	0.0	0.0	halfColWin = int((colWin + 2) / 2) # 7->4, 8->5
175	1	1.0	1.0	0.0	halfRowWin = int((rowWin + 2) / 2)
176	251	202.0	0.8	0.0	for i in range(0,rows):
177	62750	19368.0	0.3	0.9	for j in range(0,cols):
178	62500	19584.0	0.3	0.9	s=0
179	62500	35250.0	0.6	1.6	xmin=max(0,i-halfRowWin)
180	62500	31066.0	0.5	1.4	xmax=min(rows,i+halfRowWin)
181	62500	31197.0	0.5	1.4	ymin=max(0,j-halfColWin)
182	62500	31088.0	0.5	1.4	ymax=min(cols,j+halfColWin)
183	435250	149797.0	0.3	6.6	for k in range(xmin,xmax):
184	2595831	909375.0	0.4	40.0	for l in range(ymin,ymax):
185	2223081	1006649.0	0.5	44.3	s+=input[k*rows+l]
186	62500	37409.0	0.6	1.6	output[i*rows+j]=s/((xmax-xmin)*(ymax-ymin))

Appendix 2

Line profiler results for initial `myFINd1.py` script. Highlighted lines correspond to function calls that have been optimized. Timer unit: $1e-6s$.

```
Function: fromFile at line 39
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    39                                           @profile
    40                                           def fromFile(self, filepath):
    41         1         15.0     15.0      0.0                  img = None
    42         1          1.0      1.0      0.0                  try:
    43         1      21026.0  21026.0      5.6                          img = Image.open(filepath)
    44                                           except IOError as e:
    45                                                   raise e
    46         1     357257.0 357257.0     94.4                  return self.fromImage(img)


Function: fromImage at line 48
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    48                                           @profile
    49                                           def fromImage(self,img):
    50         1          0.0      0.0      0.0                  try:
    52         1       1138.0   1138.0      0.3                          img=img.copy()
    53         1         34.0     34.0      0.0                          img.thumbnail((512, 512))
    54                                           except IOError as e:
    55                                                   raise e
    56         1          2.0      2.0      0.0                  numCols, numRows = img.size
    57         1       4660.0   4660.0      1.3                  buffer1 = self.fillFloatLumaFromBufferImage(img)
    58         1         52.0     52.0      0.0                  buffer64x64 = MatrixUtil.allocateMatrix(64, 64)
    59         1         10.0     10.0      0.0                  buffer16x64 = MatrixUtil.allocateMatrix(16, 64)
    60         1          7.0      7.0      0.0                  buffer16x16 = MatrixUtil.allocateMatrix(16, 16)
    61         1          3.0      3.0      0.0                  numCols, numRows = img.size
    62         2     350962.0 175481.0     98.3                  return self.findHash256FromFloatLuma(
    63         1          1.0      1.0      0.0                          buffer1, numRows, numCols, buffer64x64, buffer16x64, buffer16x16
    64                                                   )


Function: fillFloatLumaFromBufferImage at line 66
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    66                                           @profile
    67                                           def fillFloatLumaFromBufferImage(self, img):
    68         1        771.0    771.0     16.6                  rgb_image = np.array(img.convert("RGB"))
    69         1        372.0    372.0      8.0                  r = np.array(rgb_image[:,:,0])*self.LUMA_FROM_R_COEFF
    70         1        332.0    332.0      7.2                  g = np.array(rgb_image[:,:,1])*self.LUMA_FROM_G_COEFF
    71         1        372.0    372.0      8.0                  b = np.array(rgb_image[:,:,2])*self.LUMA_FROM_B_COEFF
    72         1       2793.0   2793.0     60.2                  return (r + g + b).flatten().tolist()


Function: findHash256FromFloatLuma at line 74
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    74                                           @profile
    75                                           def findHash256FromFloatLuma(
    76                                                   self,
    77                                                   fullBuffer1,
    78                                                   numRows,
    79                                                   numCols,
    80                                                   buffer64x64,
    81                                                   buffer16x64,
    82                                                   buffer16x16,
    83                                           ):
    84         1          5.0      5.0      0.0                  windowSizeAlongRows = self.computeBoxFilterWindowSize(numCols)
    85         1          1.0      1.0      0.0                  windowSizeAlongCols = self.computeBoxFilterWindowSize(numRows)
    86
    87         1     309276.0 309276.0     88.1                  self.boxFilter(fullBuffer1,buffer64x64,numRows,
                                                                  numCols,windowSizeAlongRows,windowSizeAlongCols)
    88
    89         1      38986.0  38986.0     11.1                  self.dct64To16(buffer64x64, buffer16x64, buffer16x16)
    90         1       2676.0   2676.0      0.8                  hash = self.dctOutput2hash(buffer16x16)
    91         1          0.0      0.0      0.0                  return hash


Function: boxFilter at line 150
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   150                                           @classmethod
   151                                           @profile
   152                                           def boxFilter(cls,input,output,rows,cols,rowWin,colWin):
   153         1          1.0      1.0      0.0                  halfColWin = int((colWin + 2) / 2)  # 7->4, 8->5
   154         1          1.0      1.0      0.0                  halfRowWin = int((rowWin + 2) / 2)
   155        65         24.0      0.4      0.0                  for i in range(64):
   156        64         41.0      0.6      0.0                          ini = int(((i + 0.5) * rows) / 64)
   157      4160       1517.0      0.4      0.8                          for j in range(64):
   158      4096       3129.0      0.8      1.7                                  inj = int(((j + 0.5) * cols) / 64)
   159      4096       1706.0      0.4      0.9                                  s=0
   160      4096       2808.0      0.7      1.6                                  xmin=max(0,ini-halfRowWin)
   161      4096       2415.0      0.6      1.3                                  xmax=min(rows,ini+halfRowWin)
   162      4096       2285.0      0.6      1.3                                  ymin=max(0,inj-halfColWin)
   163      4096       2336.0      0.6      1.3                                  ymax=min(cols,inj+halfColWin)
   164     28480      11774.0      0.4      6.6                                  for k in range(xmin,xmax):
   165    169545      70498.0      0.4     39.2                                          for l in range(ymin,ymax):
   166    145161      78373.0      0.5     43.6                                                  s+=input[k*rows+l]
   167      4096       2829.0      0.7      1.6                                  output[i][j]=s/((xmax-xmin)*(ymax-ymin))
```

Appendix 3

Memory profiler results for the original FINd algorithm, its single-process and multi-process optimizations as well as `average_hash` and `phash` from the `imagehash` library. Note that the reported peak memory for the multi-process optimization refers to each parallel process, so the actual peak memory lies around the reported peak memory times the processors used.

```
%load_ext memory_profiler
```

```
# Importing images
imgs = glob.glob('das_images/0000*.jpg')
```

```
# Original FINd algorithm
findHasher = FINDHasher()
```

```
%memit [findHasher.fromFile(i) for i in imgs]
```

```
peak memory: 107.57 MiB, increment: 0.01 MiB
```

```
# Single-process optimization
findHasher_new = FINDHasherr()
```

```
%memit [findHasher_new.fromFile(i) for i in imgs]
```

```
peak memory: 107.53 MiB, increment: 11.33 MiB
```

```
# Multi-process optimization
%memit multiprocess(imgs)
```

```
peak memory: 107.60 MiB, increment: 0.07 MiB
```

```
# average hashing
%memit [imagehash.average_hash(Image.open(i)) for i in imgs]
```

```
peak memory: 107.56 MiB, increment: 0.00 MiB
```

```
# phashing
%memit [imagehash.phash(Image.open(i)) for i in imgs]
```

```
peak memory: 107.56 MiB, increment: 0.00 MiB
```

Appendix 4

Time profiling results for the original FINd algorithm, its single-process and multi-process optimizations as well as `average_hash` and `phash` from the `imagehash` library for a total input of 3444 images. Note that the multi-process optimization is running on 8 local CPUs.

```
# Importing images
imgs = glob.glob('das_images/000*.jpg')
print(f"Number of images: {len(imgs)}")
```

```
Number of images: 3444
```

```
# Original FINd algorithm
findHasher = FINDHasher()
```

```
%timeit [findHasher.fromFile(i) for i in imgs]
```

```
20min 45s ± 51.2 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Single-process optimization
findHasher_new = FINDHasherr()
```

```
%timeit [findHasher_new.fromFile(i) for i in imgs]
```

```
1min 50s ± 5.39 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Multi-process optimization
%timeit multiprocess(filepath=imgs)
```

```
35.5 s ± 2.02 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# average hashing
%timeit [imagehash.average_hash(Image.open(i)) for i in imgs]
```

```
3.75 s ± 62.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# phashing
%timeit [imagehash.phash(Image.open(i)) for i in imgs]
```

```
4.29 s ± 113 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Appendix 5

Accuracy results for the multi-process optimization of the original FINd algorithm as well as `average_hash` and `phash` from the `imagehash` library for the first ten image families (i.e., `'das_images/000*.jpg'` = 3444 Images).

	multi-process	ahash	phash
Accuracy	0.9956	0.9803	0.9887
True Positive Rate	0.9735	0.9297	0.9486
True Negative Rate	0.9999	0.9880	0.9969
Precision	0.9994	0.9494	0.9806