CSC320 — Introduction to Visual Computing, Spring 2019

Assignment 4: Generalized PatchMatch & Image Denoising

Posted: Thursday, March 21, 2019
Due: 12:00pm, Thursday, April 4, 2019
Late policy: 1% deduction per 24hrs (not a typo) for the first two days late, 15% deduction per 24hrs after that. Submission not accepted if > 7 days late

---

In this assignment you will extend your PatchMatch implementation from A3 in two ways: (1) computing k nearest-neighbour fields per image pair, corresponding to the k best-matching patches between source and target, and (2) using these fields to denoise a noisy input image. The techniques behind these tasks are described in papers by Barnes *et al.* and Buades *et al.*, respectively. To complete this assignment you need to look at an extremely small piece of each paper—just two paragraphs from the first and three from the second.

Your specific task is to complete the technique's implementation in the starter code. The starter code is essentially identical to the starter code for A3, and your implementation requires just a few changes/extensions to your A3 implementation. As in A3, it is based on OpenCV and is supposed to be executed from the command line, not via a Kivy GUI.

Goals: The goals of the assignment are to (1) get you familiar with the use of more advanced data structures for image processing, namely *heaps*; (2) see how the relatively simple algorithm you implemented in A3 can open the door to very powerful image processing tools; and (3) become even more aware of the efficiency of the code you write.

Important: As in the previous assignments, you are advised to start *immediately* by reading the relevant parts of the papers (see below). The next step is to compare the starter code to the one you used for A3. As in A3, the assignment involves a combination of understanding what it is that you have to implement and then actually doing the coding. Your task, however, is conceptually much simpler than A3 because you are extending an algorithm and an implementation you should already know quite well. *If you were not able to get your A3 implementation working, please contact us on Piazza ASAP. You will be given permission to copy and extend someone else's A3 code so that you are not hampered by your A3-specific issues.*

Testing your implementation on CDF: As in A3, it is possible to develop and run your code remotely from CDF.

Starter code & the reference solution

Use the following sequence of commands on CDF to unpack the starter code and to display its input arguments:

```
> cd ~
> tar xvfz patchmatch_k.tar.gz
> rm patchmatch_k.tar.gz
> cd CS320/A4/code
> python viscomp.py --help
```

Consult the file *320/A4/code/README_1st.txt* for details on how the code differs from A3. Its structure should be thoroughly familiar to you by now.

As I have explained in previous assignments, you should not expect your implementation to produce *exactly* the same output as the reference solution; tiny differences in implementation might lead to slightly different results. This is not a concern, however, and the TAs will be looking at your code as well as its output to make sure what you are doing is reasonable.

320/A4/CHECKLIST.txt: Please read this form carefully. It includes information on the course's Academic Honesty Policy and contains details on the distribution of marks in the assignment. You will need to complete this form prior to submission, and this will be the first file the TAs will look at when marking your assignment.

Part 1. The Generalized PatchMatch Algorithm (70 Marks)

The technique is described in full detail in the following paper (available here):

C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein, "The Generalized PatchMatch Correspondence Algorithm," *Proc. ECCV 2010.*

The only part of the paper you really need to read is the first paragraph of Section 3.2 and the paragraph entitled *Heap algorithm* in the same section. Rather than keeping track of the "best" displacement for each pixel as you did in A3, your implementation must now maintain a *MAX heap* for every pixel. The pixel's heap should contain the k displacements and the patch distance of each displacement.

The MAX heap is a data structure that implements a *priority list.* Its most important feature is that it allows very efficient identification of the maximum element in it. This data structure is (almost) provided by Python: Python's `heapq` package provides a MIN heap, which allows efficient removal of the *smallest* element in the heap, along with operations such as pushing and popping elements from it. You will need to think about what you need to do to make `heapq` behave like a MAX heap for your task.

Beyond this data structure, the random search and propagation are done exactly as in the original algorithm—except that now you have k displacement candidates associated with every pixel rather than just one. I am also providing the skeleton of a couple of helper functions that you should implement in order to streamline your implementation of this function.

More guidelines on how to structure your implementation are provided in the starter code. As in A3, your entire implementation should reside in the file *320/A4/code/algorithm.py.*

You will need to copy some code verbatim from your A3 implementation: the code for image reading and writing and for the `reconstruct_source_from_target()` function. See the file *320/A4/code/README_1st.txt* for details.

Part 1.1 The *propagation_and_random_search_k()* function (55 Marks)

This function generalizes `propagation_and_random_search()` from A3. In fact, the reference implementation differs only by about 15 lines of code from the reference code in A3. The only difference is that it now needs to maintain a heap for each pixel. In addition to the heap, the function must also maintain a dictionary for each pixel. This per-pixel dictionary makes it easy and efficient to check whether or not a particular candidate displacement is already in a pixel's heap so that it is not added again. The `propagation_and_random_search_k()` function should not insert duplicate entries into a heap, as per the algorithm in the paper. See the file *320/A4/code/algorithm.py* for

more information.

Part 1.2. The heap helper functions (15 Marks)

The purpose of these functions is to make it easy for you to convert an array of k nearest-neighbour fields to/from a heap. These functions serve to structure your thinking on how to extend your A3 implementation of `propagation_and_random_search()` without making too many changes to it. It is strongly recommended that you tackle these functions first—and make sure they work correctly—before modifying your `propagation_and_random_search()` function from A3.

Note: The starter code will not run before these functions are implemented since some of the code in file *patchMatch.py* depends on them.

Part 2. The Non-Local Means Denoising Algorithm (20 Marks)

The technique is described in full detail in Section 3 of the following paper (available here):

A. Buades, B. Coll, and J.-M. Morel, "A non-local algorithm for image denoising," *Proc. CVPR 2005.*

Do not get intimidated by the other sections of this paper! They are *not* relevant at all for your implementation; ignore them and you will be just fine.

The non-local means algorithm (NLM for short) takes as input one noisy input image and returns a "denoised" version of it, *i.e.,* an image where noise is reduced compared to the original. The algorithm is extremely simple and is summarized in in Figure 1 and in the first equation of Section 3. According to that equation, the intensity of pixel i of the denoised output image is a weighted average of the intensities of a small set of pixels j in the noisy input image. The NLM algorithm specifies exactly (1) how to choose the "right" set of pixels j for each pixel i and (2) what weight $w(i, j)$ to assign to each pixel j when computing this average.

This is where the Generalized PatchMatch algorithm comes to the rescue: before running NLM, we run Generalized PatchMatch using the same noisy image as both the source and the target. This computes, for every pixel i in the noisy image, the k pixels whose patches are most similar to the patch centered at i. To compute the denoised intensity at pixel i, NLM simply computes a weighted average of the intensity of these pixels.

According to NLM, the weights $w(i, j)$ in this average should depend on the similarity of the patches centered at pixels i and j. The exact expressions to use are given by the third and fourth equation in Section 3 (you can ignore the second equation). They are computed from the sum-of-squared-differences between the patches.

Part 3. Efficiency considerations (0 Marks)

You should pay attention to the efficiency of the code you write. Explicit loops (probably) cannot be completely avoided for most of the functions you have to write, but their use should be kept to an absolute minimum. This is necessary to keep the running time of your code to a reasonable level: expect your code to take longer time to process an image compared to A3, and *much* longer if you use too many loops. That being said, you will not lose marks by writing inefficient code for this assignment.

Part 4. Report and Experimental evaluation (10 Marks)

Your task here is to put the NLM algorithm to the test. Try it on a variety of noisy photos (*e.g.,* taken with your cellphone in poor lighting conditions), and comment on how the following parameters of the Generalized PatchMatch algorithm affect the results of NLM: (a) the maximum search radius w; (b) the number of nearest neighbours k; and (c) the patch size. Include these results in your report, along with the results of applying NLM to the noisy image in the starter code. The more solid evidence (*i.e.,* results) you can include in your report PDF to back up your arguments and explanations, the better.

Finally, your report should also include the results of running Generalized PatchMatch on the `jaguar2` image pair for $k = 3$ and all the other parameters set to their default values.

Place your report in file *320/A4/report/report.pdf*. You may use any word processing tool to create it (Word, LaTeX, Powerpoint, html, *etc.*) but the report you turn in must be in *PDF format*. Please do not submit any actual image files.

What to turn in:

```
CHECKLIST.txt
report.pdf
algorithm.py
patchMatch.py
```

WARNING: Re: Academic Honesty

PatchMatch is a popular algorithm. C++ and OpenCV implementations are just a mouse click (or google search) away. You must resist the temptation to download and/or adapt someone else's code and submit it without appropriate attribution. This is a serious academic offence that can land you in a lot of trouble with the University.

Be aware that if an existing implementation is easy for you to find, it is just as easy for us to find as well—in fact, you can be sure that we already have found and downloaded it.