Supplementary Material for The Generalized PatchMatch Correspondence Algorithm

Connelly Barnes[1], Eli Shechtman[2], Dan B Goldman[2], Adam Finkelstein[1]

[1]Princeton University, [2]Adobe Systems

1 Overview

In our submission, we elided less important details from the sections on k-nearest neighbors, enrichment, and the parallel tiled algorithm (Sections 3.2, 3.3, and 3.6, respectively, of the submission), and referred the interested reader to the supplementary material.

Therefore, the supplementary material addresses the details of these three sections. First, we discuss details of the k-nearest neighbors algorithm. We include a description of the ten different alternative algorithms that we compared our heap algorithm against (Section 2), and offer additional benchmarks comparing our algorithm, kd-tree, and FLANN (Section 3). Second, we discuss different variants of the enrichment algorithm beyond the forward and inverse enrichment algorithms mentioned in the submission (Section 4). Finally, we demonstrate the roughly linear speed-up obtained by our parallel tiled algorithm on a multicore machine (Section 5).

2 Variants of k-nearest neighbors

In our submission, we described a heap algorithm for finding k-nearest neighbors. The heap algorithm generally outperforms the ten alternative k-NN algorithms that we compared against, as well as kd-tree and FLANN, so we suggested it is a good algorithm for a wide variety of problems requiring image patch correspondence. Here we describe these ten alternative algorithms. The relative efficiency of these algorithms, as well as further comparisons with kd-tree and FLANN are shown in Figure 1. The k-NN algorithms are as follows:

Heap. As a reminder, the heap algorithm stores a heap of k nearest neighbors at every patch position. During propagation, we improve the k nearest neighbors at the current position, by exhaustively testing each of the k nearest neighbors implied by the adjacent patches to the left or above (or below or right on even iterations). If any candidate is closer than the worst nearest neighbor currently stored at (x, y), the worst nearest neighbor is replaced with the candidate. This can be done efficiently with a max-heap, where the heap stores the patch distance D. The random search phase works similarly: n samples are taken around each of the k nearest neighbors already in the heap, giving nk samples total. The worst element of the heap is evicted if the candidate's distance is better.

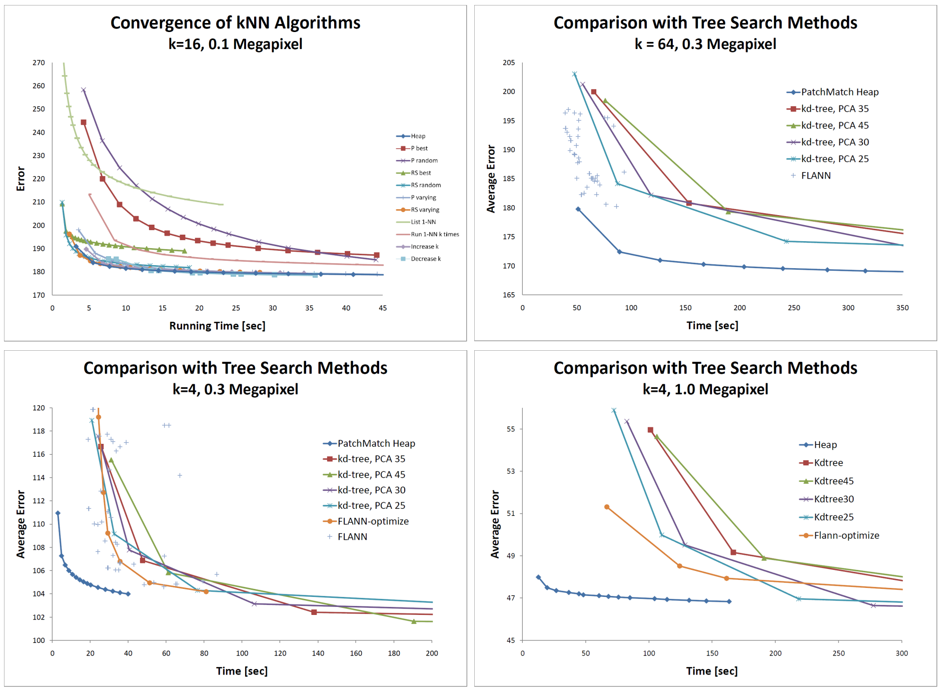

Fig. 1. Upper left: Comparison of variants of our k-NN. This is similar to Figure 2 left of our submission, except the images are sampled at 0.1 MP instead of 0.2 MP. The other three graphs show that our k-NN outperforms kd-tree by factor of roughly 3-10, and FLANN by a factor of roughly 1.4-7, depending on k value and resolution. These plots are averaged over the dataset of 36 images.

Heap algorithm with fewer P or RS operations. The previous heap algorithm can be modified to only propagate or random search on one element, instead of all elements. For propagation, one can examine only the adjacent nearest neighbor with smallest distance (denoted "P best" in Figure 1) or choose an element at random as a candidate for propagation (denoted "P random"). Likewise, in random search, one can randomly sample around the nearest neighbor with smallest distance (denoted "RS best") or sample around a randomly chosen nearest neighbor (denoted "RS random"). Finally, one could run propagation or random search on the top m nearest neighbors, where m is randomly chosen uniformly between 1 and k (denoted "P varying" and "RS varying"). As shown in Figure 1, these strategies are slower than the original heap algorithm.

Use the 1-NN algorithm to find k-NN. One can use the original PatchMatch algorithm to find k-NN. One strategy is to retain all of the candidate nearest neighbors sampled by the algorithm in a list, and take the top k of these, after partial sorting of the list (denoted "List 1-NN"). Another strategy is to run the 1-NN algorithm k times, with each run constrained so that nearest neighbors cannot be equal to any of the previously chosen nearest neighbors (denoted "Run 1-NN k times"). Again these strategies are slower than the original heap algorithm.

Changing k during iterations. One can start on iteration 1 with a number of nearest neighbors k_0, and after half of the iterations are completed, increase or decrease this to the final desired number of nearest neighbors k, either dropping the worst elements of the heap, or adding uniform random elements as needed. We tried increasing from a small number of nearest neighbors $k_0 = k/2$ (denoted "Increase k"), and decreasing from a large number of nearest neighbors $k_0 = 2k$ (denoted "Decrease k"). Again these algorithms are slower than the simple heap algorithm.

3 Comparison of our k-NN against kd-tree and FLANN

In Figure 1, the upper-right and lower plots give additional comparisons of the speed of our algorithm against kd-tree and FLANN. These plots are similar to Figure 2 left in our submission, however they vary the number of neighbors from $k = 4$ to $k = 64$, and the image resolution from 0.3 to 1.0 megapixels. We find that for equal errors, our k-NN outperforms kd-tree by a factor of roughly 3-10, and FLANN by a factor of roughly 1.4-7, depending on k value and resolution. These plots are generous to the competing algorithms, because they tune all parameters of the competing algorithms, whereas we only varied the number of iterations of our algorithm. The "FLANN" data points indicate a dense random sampling of the FLANN parameter space, and the "FLANN-optimize" curve indicates that the simplex method included in FLANN was used to tune the algorithm. These data points exclude the overhead time of the tuning optimization. Note also that the plots in Figure 1 do not include enrichment, so the performance gap becomes greater when enrichment is employed.

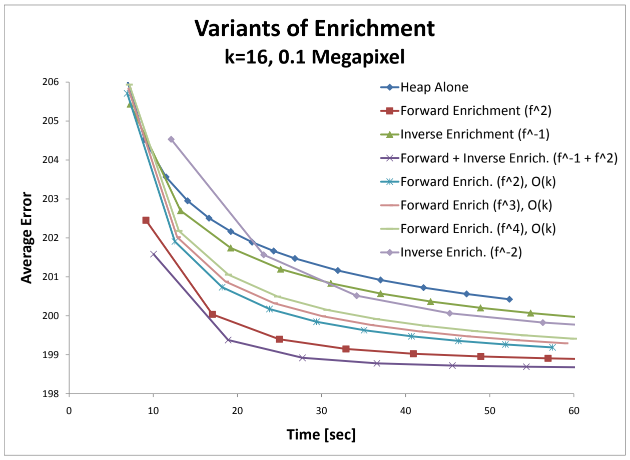

Fig. 2. Effect of different variants of enrichment on convergence. This is the same as Figure 3 left of the submission, but it includes more variants. Here $O(k)$ indicates that the $O(k)$ time algorithm was used. The fastest algorithm overall is the $\mathbf{f}^{-1}$ inverse enrichment followed by the $\mathbf{f}^2$ forward enrichment. This plot was averaged over a subset of the 36 images in the dataset.

4 Variants of enrichment

In Section 3.3 of our submission, we discussed an optimization technique for further improving PatchMatch performance: *enrichment*. As a review, enrichment is defined as the propagation of good matches from a patch to its k-NN, or vice versa. We call this operation enrichment because it takes a nearest neighbor field $\mathbf{f}$ and improves it by considering a "richer" set of potentially good candidate matches than propagation or random search alone.

In our submission we introduced two types of enrichment, in the special case of matching patches in image A to other patches in the same image. Here we elaborate on the different possible variants of enrichment. We show in Figure 2 that the simplest forward enrichment algorithm ($\mathbf{f}^2$) and inverse enrichment algorithm ($\mathbf{f}^{-1}$) generally perform best.

Forward enrichment uses compositions of the function $\mathbf{f}$ with itself to produce candidates for improving the nearest neighbor field. The canonical case of forward enrichment is $\mathbf{f}^2$. That is, if $\mathbf{f}$ is a NNF with k neighbors, we construct the NNF $\mathbf{f}^2$ by looking at all of our nearest neighbor's nearest neighbors: there are k^2 of these. The candidates in $\mathbf{f}$ and $\mathbf{f}^2$ are compared and the best overall are used as an improved NNF $\mathbf{f}'$. If min() denotes taking the top k matches, then we have: $\mathbf{f}' = \min(\mathbf{f}, \mathbf{f}^2)$. Alternatively, higher orders of the function can be used, such as $\mathbf{f}^3, \mathbf{f}^4$ and so forth. One problem is that simplest enrichment algorithm takes $O(k^2)$ time to find distances to the k^2 neighbors, while the higher order algorithms take $O(k^3), O(k^4)$ time, thus for high numbers of neighbors k the algorithms do not scale well. Therefore we propose two variants of enrichment

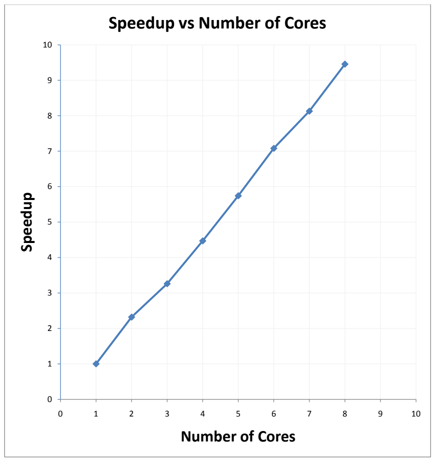

Fig. 3. Approximately linear speedup of parallel algorithm with increasing number of cores. We attribute deviations from linearly to cache effects.

that take $O(k)$ time: we can either take k random samples from the function $\mathbf{f}^n$, or we can take the top $\sqrt[n]{k}$ elements of $\mathbf{f}$ before computing $\mathbf{f}^n$. As shown in Figure 2 these alternative algorithms generally underperform the simplest $\mathbf{f}^2$ enrichment algorithm.

Similarly, **inverse enrichment** walks the nearest-neighbor pointers backwards to produce candidates for improving the NNF. The canonical algorithm is $\mathbf{f}^{-1}$, representing the multi-valued inverse of function $\mathbf{f}$. The improved NNF is given by $\mathbf{f}'' = \min(\mathbf{f}, \mathbf{f}^{-1})$. As before, we can consider higher inverse powers of $\mathbf{f}$, such as $\mathbf{f}^{-2}$ or $\mathbf{f}^{-3}$. These functions have varying numbers of neighbors at each offset, but averaged over the domain, will have k^2 or k^3 neighbors, respectively, per patch, therefore, the same strategies for reducing running time to $O(k)$ can be applied to inverse enrichment. As shown in Figure 2, the simplest $\mathbf{f}^{-1}$ inverse enrichment outperforms the alternatives, when it is coupled with $\mathbf{f}^2$ forward enrichment (interestingly, the $\mathbf{f}^{-2}$ inverse enrichment slightly outperforms $\mathbf{f}^{-1}$ for large numbers of iterations, however, this performance advantage disappears when it is combined with forward enrichment).

5 Speed-up for parallel tiled algorithm

We tested the parallel tiled algorithm, described in Section 3.6 of the submission, on two 8 core machines. On one machine we obtained a slightly sub-linear speedup. On the other machine we obtained a slightly super-linear speedup, as shown in Figure 3. We attribute the variation in speed-up to cache effects.

One implementation detail of the parallel tiled algorithm is that resource conflicts can occur where information is shared between adjacent nearest neighbors during propagation. In particular, it is possible for the bottom row of a tile to be written while the top row of the next tile is using the data for propagation (or vice versa on even iterations, where scanlines are processed bottom-to-top). Therefore, as a special case, we write back the last row of each tile only after the critical section that is used to synchronize the tiles. (Alternatively to avoid this synchronization issue, on an n core machine, one can simply split the input into $2n$ tiles vertically, and process even tiles in parallel, followed by odd tiled in parallel.)

6 Discussion

We have illustrated that our heap algorithm generally outperforms ten other algorithms based on PatchMatch, as well as kd-tree and FLANN, for various parameter settings. We have also discussed other variants of enrichment, and demonstrated that the simple forward ($\mathbf{f}^2$) and inverse ($\mathbf{f}^{-1}$) enrichment algorithms are good choices for accelerating convergence. We finally demonstrated that the algorithm can be further accelerated on multicore architectures, with a roughly linear speed-up on an eight core machine. We believe these properties show that our algorithm is a flexible and efficient tool for dense, global correspondence problems.