Fast Superpixels for Video Analysis

Fabio Drucker and John MacCormick
Dickinson College, Carlisle, PA

Abstract

The computational cost of video and motion analysis can be dramatically reduced by over-segmenting each frame of video into "superpixels". But most superpixel algorithms are themselves computationally expensive, and are thus inappropriate for use with real-time video. This paper advocates and analyzes the use of superpixels derived from minimum-cost paths that can be computed by dynamic programming. It is shown that superpixels can be computed comfortably in real time using such methods (30–40 times faster than the most efficient alternative), while sacrificing about 3% in the accuracy of the superpixels. The efficacy of the approach is demonstrated with a simple video analysis application.

1. Introduction

The partitioning of an image into "superpixels" — contiguous regions that are perceptually similar — is an important pre-processing step in certain computer vision algorithms. Existing algorithms for producing superpixels are both accurate and efficient, and have been used with great success in large-scale applications such as the segmentation of all images in a large database [8]. Superpixel preprocessing also has the potential to vastly reduce the cost of video analysis, since it can reduce the number of correspondences between frames from millions of pixels to perhaps a thousand superpixels (e.g. [11, 15]). However, for at least some video analysis applications, existing superpixel techniques are too expensive for real-time processing of video. Hence, there is a strong motivation to reduce the cost of superpixel computation. This paper does exactly that, by describing and analyzing PathFinder, an extremely low cost superpixel algorithm based on computing least-cost paths via dynamic programming.

It is important to distinguish clearly between two different but related tasks, defined in this paper as follows. *Segmentation* partitions an image into a small number (say, 1–50) of regions that correspond to real-world objects. *Oversegmentation* partitions an image into a moderate number (say, a few hundred) of superpixels, each of which corresponds to *part of* a real-world object. Clearly, these are not mathematically rigorous definitions, as they depend on the psychological concept of an "object". Nevertheless, we find these definitions useful in practice.

2. Related work and this paper's contribution

To the best of our knowledge, the fastest existing superpixel algorithm is the "Efficient Graph-Based Image Segmentation" (EGBIS) of Felzenszwalb and Huttenlocher [5]. Section 4.2 demonstrates that PathFinder is 30–40 times faster than EGBIS on moderately-sized images, while sacrificing about 3% in the accuracy of the superpixels.

Another extremely efficient superpixel technique is the watershed algorithm [14]. Although inexpensive in its vanilla form, the watershed algorithm requires additional (expensive) morphological operations for some applications (e.g.[15]), and appears vulnerable to highly-textured regions. In this paper, we therefore concentrate on comparisons with EGBIS, while acknowledging that a more complete future study should compare with watershed, and possibly other approaches.

Numerous other high-quality and relatively efficient segmentation/superpixel algorithms exist, including mean shift [3] and weighted aggregation [12]. While they produce excellent segmentations, these algorithms are at least several times slower than EGBIS. To our knowledge, the considerable remaining literature on image segmentation comprises algorithms many times slower than those mentioned above (e.g. [7, 10, 13]), rendering them unsuitable for real-time applications.

The superpixel lattices (SL) of Moore *et al.* [9] are closely related to PathFinder. Both approaches form superpixel boundaries from minimum-cost paths that traverse the entire image. SL possesses two options for computing minimum-cost paths: graph-cuts, and dynamic programming. The graph-cut variant is more computationally expensive than PathFinder, and is not discussed further here. The superpixel lattice dynamic programming (SLDP) approach, on the other hand, is very similar to PathFinder and has approximately the same computational cost. The main differences lie in the way that topological constraints on the paths are enforced. SLDP was motivated by the objective

that the superpixels lie in a regular grid, so it forces each path to lie in a vertical or horizontal strip and prevents paths of the same orientation from intersecting. Pathfinder has no requirement for a regular grid of superpixels, so it allows paths to wander arbitrarily and simply terminates paths that intersect with an existing path of the same orientation.

Seam carving [1] is another algorithm using dynamic programming to find minimum-cost paths in an image. The seam carving technique extracts paths across (or down) the entire image just as Pathfinder does, but its authors emphasize the application to image resizing rather than superpixels for video analysis.

Although PathFinder was developed independently, it is clear that it has close similarities to both SLDP and seam carving, and therefore we claim no novelty for the PathFinder algorithm itself. We do claim to provide a useful and novel analysis of the computational cost of such algorithms, since the SL paper focused on accuracy tradeoffs resulting from the superpixels' grid structure, and did not address computational cost. Thus, the main contribution of this paper lies in its advocacy and analysis of very inexpensive superpixel algorithms based on dynamic programming (whether SLDP, PathFinder, or some other variant). Specifically, we (i) show that dynamic programming approaches can compute superpixels cheaply enough for real-time video analysis; (ii) discuss potential weaknesses of dynamic programming approaches and suggest ways of alleviating them; (iii) show that the use of a dynamic programming approach sacrifices only a modest amount of accuracy; and (iv) demonstrate the usefulness of the resulting superpixels in a simple video analysis application. The conclusion is that PathFinder-like algorithms should be attractive to researchers attempting real-time analysis of motion in video.

3. The PathFinder algorithm

As already discussed, the Pathfinder algorithm is similar to SLDP (the dynamic programming variant of Superpixel Lattices), and we claim no particular novelty for it. However, because there are numerous differences between Pathfinder and SLDP, this section gives a complete and concrete description of PathFinder. The description can also be regarded as a minor but important contribution of this paper, since the dynamic programming aspect of Superpixel Lattices was described only in a brief summary by Moore *et al.*

3.1. Strongest vertical paths

We assume that any image can be segmented into contiguous regions that are "perceptually similar". The definition of perceptual similarity is application-dependent, and further discussion of it is outside the scope of this paper. For our purposes, it is sufficient to assume that given any particular image, there is some unknown, "true" segmentation of this image into perceptually similar regions, assigning each pixel to exactly one contiguous region. We say there is a *vertical boundary* at the image location (i, j) if the pixels at locations (i, j) and $(i+1, j)$ lie in distinct perceptually similar regions. (Note the slight asymmetry in the definition: vertical boundaries occur between *horizontal* neighbors, so a vertical boundary "at" (i, j) is actually just to the east of (i, j).)

The origin is taken to be at the top left of the image, so increasing the (vertical) j-coordinate of a location moves it downwards. A *downwards path* is a sequence of pixel locations whose successive vertical locations increase by exactly 1 and whose successive horizontal locations differ by at most 1. Formally, if the path consists of locations $(i_1, j_1), (i_2, j_2), \ldots, (i_N, j_N)$, then $j_{n+1} - j_n = 1$ and $|i_{n+1} - i_n| \leq 1$ for $n = 1, \ldots, N-1$. Similarly, an *upwards path* moves up through the image, deviating horizontally by at most one pixel location in each row, so $j_{n+1} - j_n = -1$ and again $|i_{n+1} - i_n| \leq 1$. A *vertical path* is an upwards or downwards path.

The input to a vertical PathFinder algorithm is an array of values we will call *horizontal perceptual difference (HPD) strengths*, and denote $s(i, j), i = 1, \ldots, I,\ j = 1, \ldots, J$. A definition of these strengths is again application-dependent, but for the algorithm to be useful, the HPD strengths must satisfy two properties: (i) A high value of $s(i, j)$ indicates a high probability of a vertical boundary at location (i, j). (ii) Strengths should be additive, so that the sum of the strengths along a vertical path is a good proxy for detecting vertical boundaries: given two vertical paths, the one with the higher sum of HPD strengths should have a higher probability of coinciding with a "true" boundary between perceptually similar regions.

In practice, the HPD strengths would typically be the output of a horizontally-oriented filter or some other simple edge detector with very low computational cost. We don't insist that the HPD strengths represent genuine probabilities or likelihoods derived from a probabilistic generative model, or that the additivity property follows from a generative model with suitable independence assumptions. Rather, the above two properties are stated as ideal goals that should be verified empirically, as being very approximately true.

Hence, it makes sense to define the *strength* of a vertical path as the sum of its HPD strengths. Formally, for a vertical path $V = ((i_1, j_1), (i_2, j_2), \ldots, (i_N, j_N))$, its strength $S(V) = \sum_{n=1}^{N} s(i_n, j_n)$.

The *strongest vertical path passing through* (i, j), denoted $V(i, j)$, is defined in the obvious way:

$$V(i, j) = \underset{V|(i,j)\in V}{\arg\max}\, S(V) \tag{1}$$

where the argmax runs over all vertical paths V that pass through (i, j).

It should be immediately clear that strongest vertical paths can be computed by a simple dynamic programming algorithm. We state the algorithm here for completeness, again without claiming any novelty. First, arrays of *upward path strengths* $U(i, j)$ and *upward path directions* $u(i, j)$ are constructed. $U(i, j)$ is the strength of the strongest upward path beginning at (i, j), and $u(i, j)$ is the horizontal location through which the strongest such path passes in row $j - 1$. In other words, if $u(i, j) = i'$, then $i' \in \{i-1, i, i+1\}$, and the strongest upward path beginning at (i, j) passes through $(i', j - 1)$. Note that $U(i, j)$ is defined for $i = 1, \ldots I, j = 1, \ldots J$, whereas $u(i, j)$ is defined for $i = 1, \ldots I, j = 2, \ldots J$ (i.e. the top row is omitted from its domain). To avoid special notation for corner cases, we define $U(i, j) = -\infty$ for invalid values of i.

The values of U and u are computed recursively, beginning at the top of the image and moving down. The first row of U is initialized to the HPD strengths:

$$U(i, 1) = s(i, 1), \quad i = 1, \ldots, I. \tag{2}$$

Subsequent rows (with $i = 1, \ldots, I$ and $j = 2, \ldots, J$) are computed according to:

$$u(i, j) = \underset{i' \in \{i-1, i, i+1\}}{\arg\max} U(i', j - 1) \tag{3}$$

$$U(i, j) = S(i, j) + U(u(i, j), j - 1) \tag{4}$$

Arrays for downward path strengths $D(i, j)$ and directions $d(i, j)$ are defined analogously, except that to avoid double counting when we compute entire vertical paths later on, the downward definitions are not perfectly symmetric with the upward ones. While the upward paths include the current location (i, j), the downward ones do not. Thus, $D(i, j)$ is the strength of the strongest downwards path from (i, j), excluding the HPD strength of (i, j) itself; and if this strongest downwards path passes through $(i', j + 1)$, then $d(i, j) = i'$. $D(i, j)$ is defined for $i = 1, \ldots I, j = 1, \ldots J$ (and equals $-\infty$ for invalid i-values), while $d(i, j)$ is defined for $i = 1, \ldots I, j = 1, \ldots J - 1$. Arrays D and d are computed from the bottom up, initializing

$$D(i, J) = s(i, J), \quad i = 1, \ldots, I. \tag{5}$$

and computing subsequent rows (with $i = 1, \ldots, I$ and $j = J - 1, \ldots, 1$) via:

$$d(i, j) = \underset{i' \in \{i-1, i, i+1\}}{\arg\max} (S(i', j + 1) + D(i', j + 1)) \tag{6}$$

$$D(i, j) = S(d(i, j), j + 1) + D(d(i, j), j + 1) \tag{7}$$

Finally, we can define the total strength of $V(i, j)$, denoted $P(i, j)$, as the sum of the strongest upwards and downwards paths from that point:

$$P(i, j) = U(i, j) + D(i, j). \tag{8}$$

Note that this is where we use the asymmetric definitions of U and D to avoid double counting the location (i, j).

3.2. Backtracking to find a strongest path

Once the arrays u, d of strongest path directions have been computed, strongest vertical paths can be obtained via a standard backtracking technique that incurs very little computational cost. Suppose we are interested in computing $V(i^*, j^*)$, the strongest vertical path passing through (i^*, j^*). Write the individual locations in the desired path, running from the top of the image to the bottom, as $(i_1, 1), (i_2, 2), \ldots, (i_J, J)$. Then clearly we have $i_{j^*} = i^*$, since the path must pass through (i^*, j^*). From this point, we can recurse upwards, setting $i_{j-1} = u(i_j, j)$ for $j = j^*, \ldots, 2$; and downwards, setting $i_{j+1} = d(i_j, j)$ for $j = j^*, \ldots, J - 1$.

3.3. Strongest horizontal paths

For concreteness and simplicity, the description so far has focused on *vertical* paths. However, horizontal paths can be defined in an analogous manner. Perhaps the most concise way to do this is to think of horizontal paths as vertical paths on the transpose of the original image. Let $\mathcal{I}$ be the original image, $\mathcal{I}'$ its transpose, extend our earlier notation with subscripts to indicate which image strongest paths should be computed on, and use a prime to denote the transpose of an entire path (i.e. swapping the coordinates of each location in the path). Then $H_{\mathcal{I}}(i, j) = V_{\mathcal{I}'}(j, i)'$ is the strongest horizontal path through location (i, j).

Incidentally, this technique of working with the transposed image is also a good way of implementing horizontal paths in practice, since it guarantees that horizontal and vertical paths are computed using exactly the same criteria. Furthermore, any optimizations for speed that take advantage of the in-memory layout of an image (e.g. row-major) will be automatically reused, at the expense of the single fixed cost of transposing the original image.

3.4. Creating an over-segmentation

Our ultimate goal is to create an over-segmentation of an image, or equivalently to generate superpixels — and we would like to do so using the strongest path notions discussed above. Clearly, the basic strategy is to select some subset of strongest horizontal and vertical paths, and let these paths define the boundaries of the superpixels. The key trade-off is between selecting enough paths that a high proportion of perceptual boundaries are identified, and selecting few enough paths that the resulting number of superpixels is manageable. Another way of stating this is that we would like to choose paths that are *strong* (because strong paths are more likely to follow perceptual boundaries) yet *sparse* i.e. the paths are not too close to each other. Our

experiments showed that many simple strategies perform moderately well. Here we describe one particular greedy heuristic that works well in practice and has only one tuning parameter, without making any claims for its optimality.

We first describe a simplified version of the strategy for selecting a set of vertical paths, and later give some refinements. The approach for selecting a set of horizontal paths is perfectly analogous.

The approach picks a sequence of *seed locations* $(i_1, j_1), (i_2, j_2), \ldots, (i_M, j_M)$; the set of chosen vertical paths consists of paths seeded from these locations, i.e. $V(i_1, j_1), V(i_2, j_2), \ldots, V(i_M, j_M)$. The single tunable parameter of the approach is termed the *seed gap*. This is a positive integer, denoted g, specifying the smallest permitted horizontal distance between a new seed location and any previously-selected vertical path. The algorithm is simple: to choose a new seed location (i, j), we restrict attention to locations that meet the seed gap requirement, and choose among these the location with maximal strength $P(i, j)$, from equation (8).

The final result can be thought of as a deformed grid structure on the image, in which the gridlines are deformed up to a maximum of 45 degrees in order to follow the strongest perceptual boundaries possible. Look ahead to the results in Figure 5, in which the white lines represent deformed horizontal "gridlines" and yellow lines represent deformed vertical "gridlines".

3.4.1 Heuristics for sparsity

Some useful refinements, which have been found empirically to improve the sparsity of the results without significant sacrifice of perceptual boundaries, are:

1. Further restrict attention to seed locations (i, j) which are a local maximum of $P(i, j)$, with respect to i.

2. Further restrict attention to a coarse sub-sampling of the rows of the image. That is, consider only locations (i, j) for which j is a multiple of some integer G. To keep vertical and horizontal sparsity on a similar scale, we found $G = 2g$ works well in practice.

3. When backtracking from a seed location (i, j) as described in Section 3.2, terminate the backtracking early when a location that already contains a previously-chosen path is reached.

4. Results

In all experiments, the HPD strengths are computed using a simple horizontal edge detection technique. Each of the three color channels is convolved with a $1 \times 2R$ "step" filter of the form $[-1, \ldots, -1, 1, \ldots, 1]$ consisting of R -1's followed by R 1s — so R can be thought of as the "radius" of the filter, which is centered on the final -1 in order to respect the definition of vertical boundary given earlier. Experiments took $R = 3$, but the results were very similar for $R \in [1, 10]$. The absolute values of the filter results are summed over the three channels to produce the final HPD strengths.

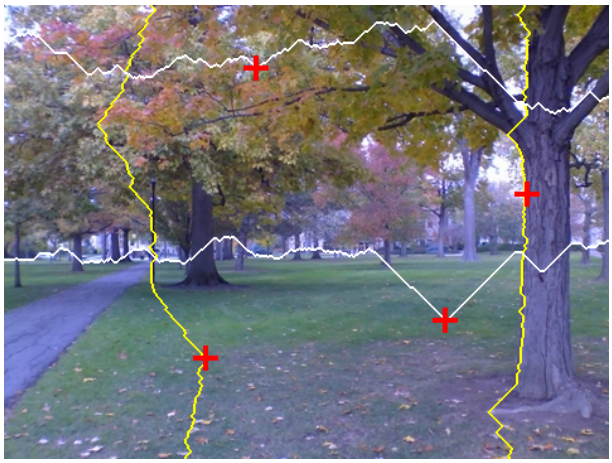

Figure 1. **Strongest paths can demarcate perceptually important boundaries, but can also wander through perceptually similar regions or "run away" to distant, strong edges.** Two vertical and two horizontal strongest paths are shown, with seed locations as red crosses.

Figure 1 shows some typical results for constructing strongest paths seeded from certain points. The test image — an outdoor scene of trees, lawn and a paved path — has resolution 480 by 360, and is used as a running example in this paper. The four seed locations are marked with red crosses, with strongest horizontal paths in white and strongest vertical paths in yellow. Figure 1 shows us three important properties of strongest vertical and horizontal paths, which we call the "boundary-seeking", "wandering" and "runaway" properties.

First, strongest paths can demarcate large, important perceptual boundaries: for example, the right-hand vertical path was deliberately seeded on the left-hand side of a significant tree trunk which it demarcates almost perfectly, and the lower horizontal path captures most of the horizon despite being deliberately seeded far from it. This "boundary-seeking" property is, of course, beneficial and lies at the heart of the PathFinder approach.

Second, because these paths are compelled to follow a given direction (either horizontal or vertical) to within 45 degrees, they frequently wander through the interior of regions that most humans would regard as perceptually similar. For example, the left-hand vertical path captures a small, perceptually-important piece of lamp post, but for the most part wanders through seemingly similar pieces of lawn and foliage. The upper horizontal path is qualitatively

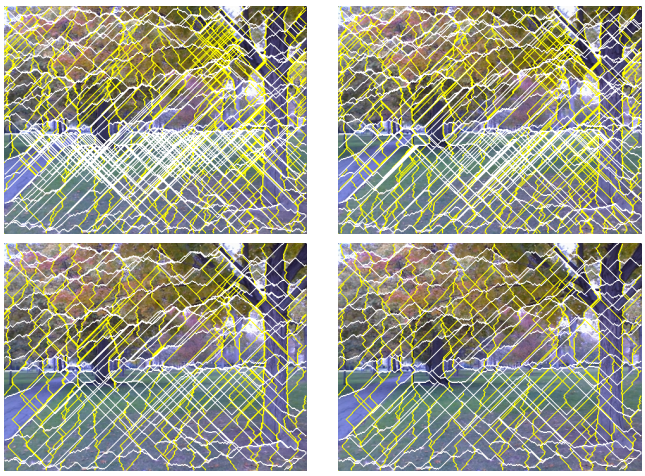

Figure 2. **Several heuristics improve sparsity of the results.** Heuristics 1, 2 and 3 are described in Section 3.4.1. Top left: no heuristics; top right: with heuristic 1; bottom left: with heuristics 1 and 2; bottom right: with heuristics 1, 2, and 3.

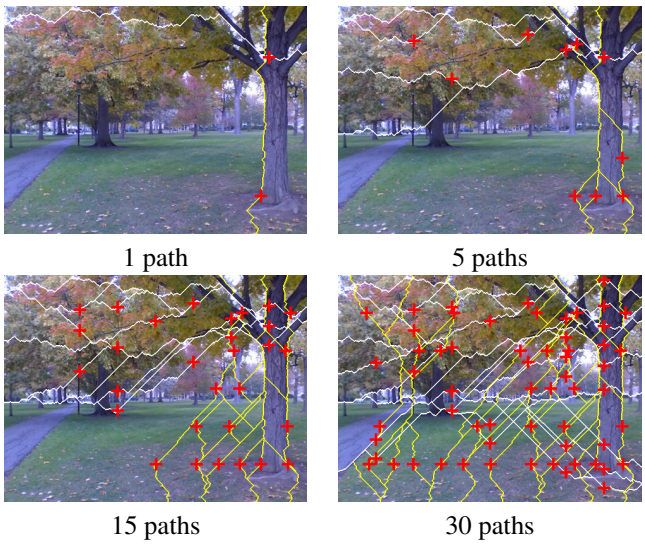

Figure 3. **PathFinder greedily adds the best possible paths, subject to sparsity heuristics.** PathFinder's intermediate results are shown, after greedily adding 1, 5, 15, and 30 strongest horizontal and vertical paths ($g = 30$). Selected seed locations are shown as red crosses.

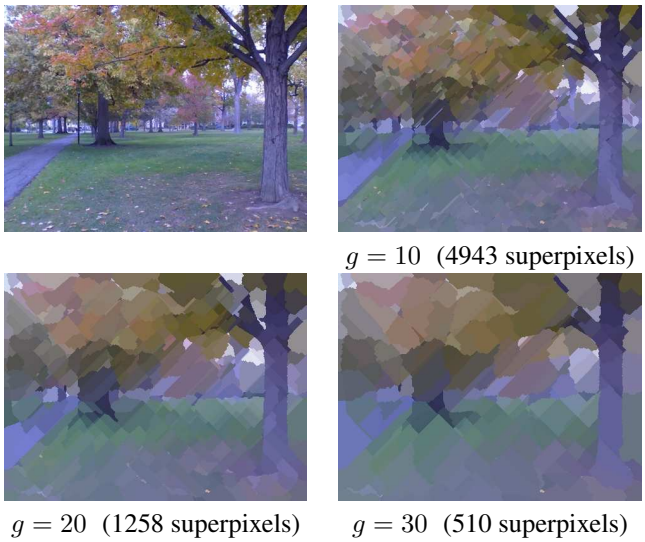

Figure 4. **PathFinder's single tuning parameter affects the average size of superpixels.**

similar, capturing some important branches but also dividing some foliage fairly arbitrarily. This "wandering" property is not necessarily a weakness of the approach, since we are seeking an *over*-segmentation: paths through perceptually similar regions are permitted, although we would like to avoid having too many of them.

Third, distant strong edges can overwhelm nearby weak ones. For example, the lower horizontal path prefers to detour up to the horizon from its seed location, rather than splitting the lawn between its slightly brown and slightly green portions. This "runaway" property can be a weakness of the approach, since it is possible for perceptually important but relatively weak edges to be ignored. The heuristics for choosing seed locations are designed to ameliorate this problem, but the reader will still see some evidence of it in the results (e.g. Figure 5). Indeed, any long stretch of path at a 45° angle is an example of a runaway path, deviating as quickly as possible towards a distant but strong edge. The 45° maximum deviation arises from the fact that by definition, a strongest vertical path's horizontal coordinate differs by at most 1 pixel in consecutive vertical locations. Simple variants of PathFinder's dynamic programming algorithm could permit maximum deviations other than 45°, or penalize deviations, but such experiments are left to future work.

Figure 2 shows the effects of the three sparsity heuristics described in Section 3.4.1. Figure 3 demonstrates PathFinder's greedy selection of seed locations. The accuracy and usefulness of the superpixels can be assessed qualitatively by creating a new image from the original, in which every pixel is assigned the average color of its containing superpixel. Figure 4 shows some examples, and also shows the effect of varying the seed gap g.

Figure 5 shows some results on the first few images from a publicly-available database of images for which ground truth segmentations are available [8].

4.1. Quality of superpixels

Figure 6 compares the results of PathFinder with EGBIS visually. Each algorithm was run on the same input, with parameters adjusted to produce approximately 1200 superpixels. It is immediately clear that EGBIS produces superpixels that are more visually appealing to humans. For example, in this case EGBIS correctly isolates more details of the foreground tree branches and the trunks of more distant

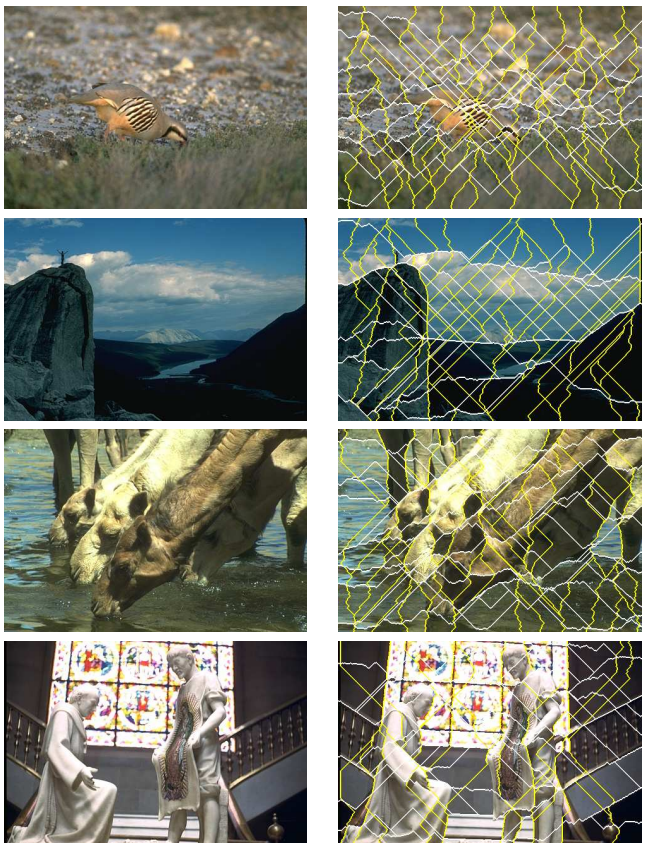

Figure 5. **PathFinder produces reasonable results on the first few images of a public segmentation data set.**

Figure 6. **EGBIS produces superpixels that are more visually appealing, but PathFinder superpixels are more regular in size and simpler in geometry**. Left: PathFinder with $g = 20$ (1258 superpixels). Right: EGBIS with $k = 60$ [5] (1216 superpixels).

trees on the horizon. In contrast, PathFinder accurately detects the boundaries of major visual components (the foreground tree, the paved path, and the horizon, for instance) but misses many finer details. Because PathFinder produces what is essentially a deformed grid structure, its superpixels are more regular in size, and simpler in geometry, than those of EGBIS.

A quantitative comparison of the accuracy of the two approaches is perhaps more important than these qualitative differences. For this, we use the "mean accuracy" as defined by Moore *et al.* [9]. Mean accuracy of an over-segmentation

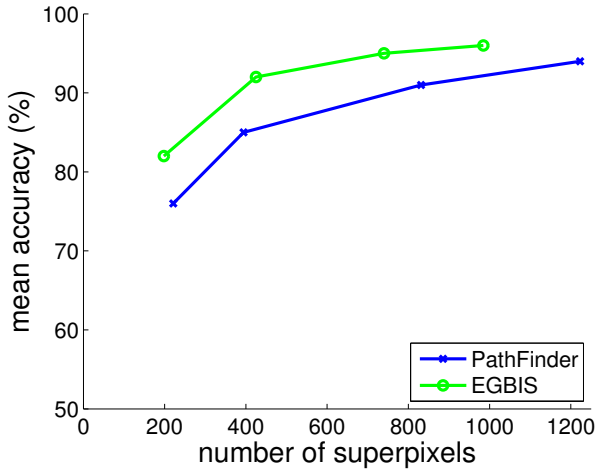

Figure 7. **PathFinder's accuracy is inferior to EGBIS', but achieves 94% accuracy at 1200 superpixels.**

is computed with reference to a ground truth segmentation (importantly, the ground truth is not an *over*-segmentation). Each superpixel in the over-segmentation is assigned to a "home" segment — the ground truth segment of maximum overlap. The mean accuracy is the proportion of pixels that lie in the home segment of their superpixel, averaged over the images from a ground truth database. Our experiment's average is taken over 50 images from the database of Martin *et al.* [8]; results are also averaged over several ground truth segmentations provided by different human subjects for each image in this database. Obviously, mean accuracy tends to one as the number of superpixels approaches the number of pixels, regardless of the quality of the over-segmentation. Therefore, algorithms should be compared using parameter settings that produce the same number of superpixels.

Figure 7 shows the results of such a comparison. EGBIS is clearly more accurate than PathFinder, but the margin decreases as the number of superpixels increases, and is less than 3% for 1000 or more superpixels. We hope there are at least some video analysis applications for which the 30-fold speed-up of PathFinder over EGBIS is worth this 3% sacrifice in accuracy.

4.2. Computational cost

The dominant computational cost for PathFinder is incurred in the computation of the upward and downward path strengths U and D via equations (2)–(7). If the image contains N pixels, it is easy to see these path strength computations cost $O(N)$. The backtracking operations that construct the actual superpixels cost in the worst case $O(N^{3/2})$, but in all experiments the backtracking expense was negligible. For practical purposes, then, the cost of PathFinder is $O(N)$. EGBIS also has excellent theoretical complexity, of

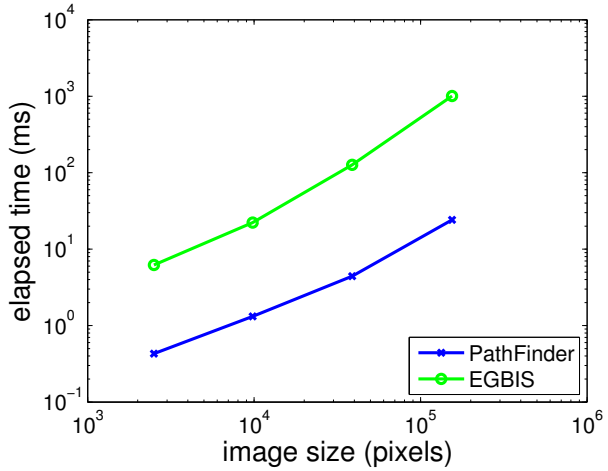

Figure 8. **PathFinder is 30–40 times faster than EGBIS.**

$O(N \log N)$.

Figure 8 shows the results of an experiment verifying these theoretical claims and comparing the actual running times. Java implementations of both algorithms were run on the same collection of images, subsampled to produce various image sizes. Both algorithms are in principle parallelizable, but we use single threaded implementations for a fair comparison. EGBIS is run with the default parameters recommended by its authors; PathFinder takes $g = 20$, which results in a similar number of superpixels. Experiments with other parameter settings produced very similar results. All experiments were conducted on a 2 GHz Intel Centrino processor. The slope of the lines in the log-log graph of Figure 8 is approximately one, confirming the linear cost of both algorithms. More importantly, PathFinder is consistently faster than EGBIS, by a factor of over 40 at around 100,000 pixels. The absolute time required by this Java implementation of PathFinder on a 480x320 image is 24ms, placing it easily within the realm of real-time analysis.

4.3. PathFinder variants

In the above experiments, the notion of perceptual distance between pixels is extremely simplistic, being derived from the elementary step filter described at the start of Section 4. Can we achieve better results using more sophisticated filtering techniques? To investigate, we re-ran the accuracy experiments, substituting a state-of-the-art object boundary detector — the Boosted Edge Learning (BEL) of Dollár *et al.* [4] — for the step filter. This produced only a modest improvement (about 1% accuracy at 1000 superpixels), but with a computational cost several orders of magnitude higher. Hence, it appears that the choice of a very simple fast filter is pragmatic and preferable for Pathfinder.

One interesting property of the step filter is that it is anisotropic; vertical paths are computed using horizontal edges, and vice versa. Experiments using an isotropic filter (essentially a Sobel operator) for both horizontal and vertical paths yielded negligible differences in accuracy.

5. A simple video application

To demonstrate the efficacy of fast superpixel algorithms using dynamic programming, an elementary video analysis tool was implemented. This tool matches each superpixel in the first frame of a video with a superpixel in each subsequent frame, and can thus be thought of as a coarse motion analysis. The matching is done recursively: if superpixel i in frame 0 (denoted $S_{i,0}$) is matched to superpixel j in frame N (denoted $S_{j,N}$), then the potential matches considered are the superpixels in frame $N + 1$ whose bounding boxes overlap the bounding box of $S_{j,N}$. The potential match whose average RGB value is closest (in Euclidean norm) to $S_{i,0}$'s average RGB value is selected. Clearly, this is not intended to be a state-of-the-art motion analysis tool; it is presented merely as a proof of concept, demonstrating the potential usefulness of Pathfinder-like algorithms.

Figure 9 shows several frames from the application of this superpixel matching to a publicly-available video [6]. Video files accompanying this submission show the entire sequence, and another sequence with a moving camera from the Middlebury optic flow data set [2]. The first column shows the original frame; the second shows superpixels obtained by PathFinder with their average RGB values; the third shows the superpixels of frame 0, but with RGB values taken from the matched superpixel in the current frame; the fourth is the "matching error" — the absolute difference between the second and third columns, multiplied by 2 in this case to increase the visibility of the discrepancies.

A perfect matching of superpixels leads to zero matching error, but the reverse is not true: an incorrect matching can also lead to a small error. Therefore, the results shown do not necessarily imply a high-quality motion analysis. But that is not our objective here. Instead, we wish to investigate the computational cost of using the superpixel approach on video. The application is written in Java, and was run on a single core of a 2.4GHz Athlon 64 X2 4600+ processor. On QCIF resolution (176x144) video, the application runs at 40ms per frame, with the computational cost divided roughly equally between (i) PathFinder, (ii) computing superpixel statistics such as mean RGB values and locations, and (iii) the recursive matching process described above. This demonstrates the real-time potential of the approach on low-resolution video, but further work is clearly required to improve the quality of the scene analysis and to achieve better performance at higher resolutions.

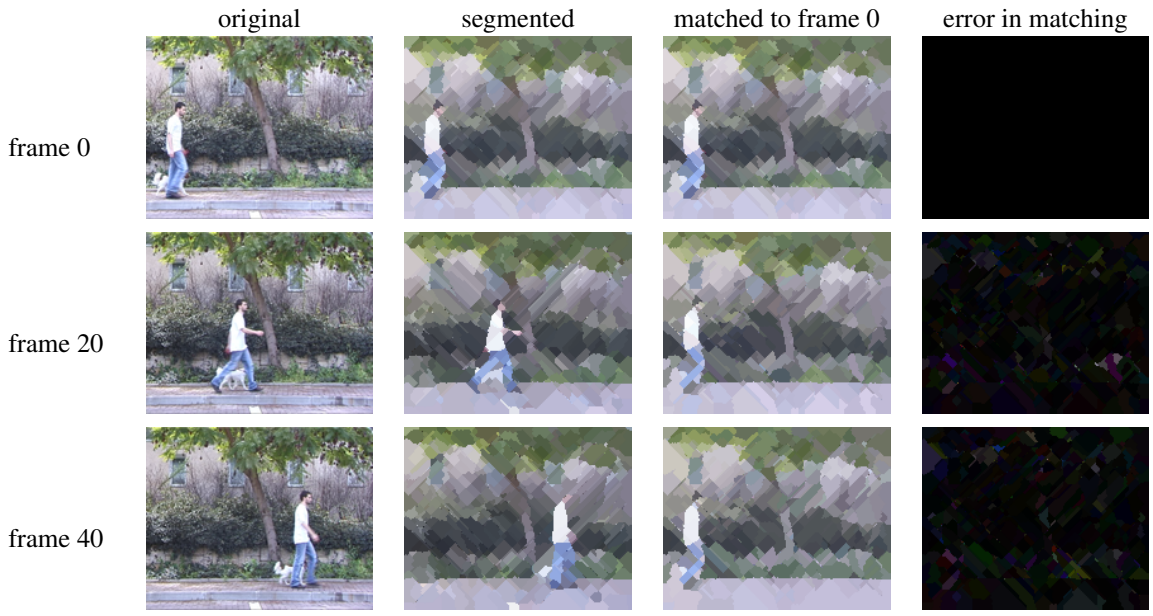

Figure 9. **A simple superpixel-matching algorithm can find plausible tracks for each superpixel at 40ms per frame.** See the text for a detailed explanation.

6. Conclusion

We demonstrated that superpixel algorithms like Pathfinder, based on least-cost paths computed via dynamic programming, run comfortably at video frame rate and 30–40 times faster than EGBIS, the next-fastest competing approach. The potential for real time video analysis using such superpixels was demonstrated with a simple example application. The superpixels obtained from Pathfinder are about 3% less accurate than EGBIS. Hence, for some video and motion applications, PathFinder-like algorithms may be the most attractive approach for real-time processing.

References

[1] S. Avidan and A. Shamir. Seam carving for content-aware image resizing. *ACM Trans. Graph.*, 26(3):10, 2007.

[2] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. In *Proc. Int. Conf. on Computer Vision*, 2007.

[3] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Trans. Pattern Analysis Machine Intell.*, 24(5):603–619, 2002.

[4] P. Dollár, Z. Tu, and S. Belongie. Supervised learning of edges and object boundaries. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2006.

[5] P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2004.

[6] L. Gorelick, M. Blank, E. Shechtman, M. Irani, and R. Basri. Actions as space-time shapes. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 29(12):2247–2253, December 2007.

[7] D. Hoiem, A. Efros, and M. Hebert. Automatic photo pop-up. In *ACM SIGGRAPH*, 2005.

[8] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.

[9] A. P. Moore, S. J. D. Prince, J. Warrell, U. Mohammed, and G. Jones. Superpixel lattices. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2008.

[10] G. Mori. Guiding model search using segmentation. In *Proc. 10th Int. Conf. Computer Vision*, volume 2, pages 1417–1423, 2005.

[11] I. Patras, E. Hendriks, and R. Lagendijk. Video segmentation by map labeling of watershed segments. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 23(3):326–332, 2001.

[12] E. Sharon, M. Galun, D. Sharon, R. Basri1, and A. Brandt. Hierarchy and adaptivity in segmenting visual scenes. *Nature*, 442:810–813, 2006.

[13] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 2000.

[14] L. Vincent and P. Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.

[15] D. Wang. A multiscale gradient algorithm for image segmentation using watershelds. *Pattern Recognition*, 30(12):2043–2052, 1997.