Linear Time Maximally Stable Extremal Regions

David Nistér and Henrik Stewénius

[1] Microsoft Live Labs
[2] Google Switzerland

Abstract. In this paper we present a new algorithm for computing Maximally Stable Extremal Regions (MSER), as invented by Matas et al. The standard algorithm makes use of a union-find data structure and takes quasi-linear time in the number of pixels. The new algorithm provides exactly identical results in true worst-case linear time. Moreover, the new algorithm uses significantly less memory and has better cache-locality, resulting in faster execution. Our CPU implementation performs twice as fast as a state-of-the-art FPGA implementation based on the standard algorithm.

The new algorithm is based on a different computational ordering of the pixels, which is suggested by another immersion analogy than the one corresponding to the standard connected-component algorithm. With the new computational ordering, the pixels considered or visited at any point during computation consist of a single connected component of pixels in the image, resembling a flood-fill that adapts to the grey-level landscape. The computation only needs a priority queue of candidate pixels (the boundary of the single connected component), a single bit image masking visited pixels, and information for as many components as there are grey-levels in the image. This is substantially more compact in practice than the standard algorithm, where a large number of connected components must be considered in parallel. The new algorithm can also generate the component tree of the image in true linear time. The result shows that MSER detection is not tied to the union-find data structure, which may open more possibilities for parallelization.

1 Introduction

Extraction of invariant regions has recently been the focus of intense study [1,2,3,4,5,6,7,8,9,10,11,12], supporting a wide variety of applications such as recognition, image retrieval, mosaicing, 3D reconstruction, tracking, robot navigation and more. Testing in a common framework was enabled by Mikolajczyk et al. [5] with a dataset that allows testing the repeatability of a region detector under disturbances such as blur, viewpoint change, zoom, rotation, lighting changes and JPEG compression.

Maximally Stable Extremal Regions (MSER) described by Matas el al [3] have become one of the commonly used region detector types, partly because of their high repeatability and partly because they are somewhat complementary to many other commonly used detectors [7,4,13]. They are viable with a relatively

D. Forsyth, P. Torr, and A. Zisserman (Eds.): ECCV 2008, Part II, LNCS 5303, pp. 183–196, 2008.

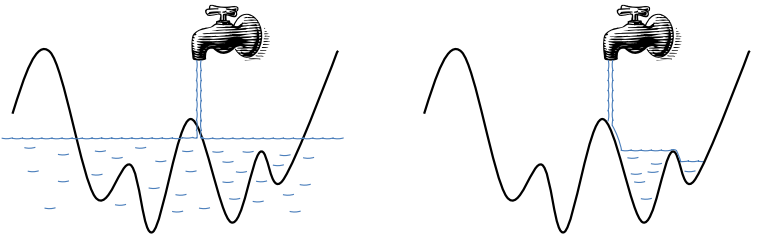

Fig. 1. The two immersion analogies. On the left, the standard immersion. On the right, the immersion corresponding to the new linear time algorithm.

small number of regions per image, and are therefore well suited for large scale image retrieval tasks [14,15]. They have also been used in recognition [16] as well as tracking [11] and have been extended to color [17] and volumetric images [18]. The detection has even been implemented on FPGA [19].

The standard algorithm for computing MSER follows the same lines as a very popular flooding simulation algorithm for computing a watershed segmentation, suggested in [20]. The watershed segmentation has a long history and has been intensely studied, see [21] for a review. In the immersion analogy, Fig 1, the grey-level profile of the image is imagined as a landscape height-map. The water level is raised gradually until the whole landscape is immersed. In the standard immersion analogy, the level is raised equally at all places in the landscape (image). That is, one either thinks of the landscape as porous so that the water level will be equal everywhere, or equivalently, imagines that a hole is pierced in each local minimum of the landscape, allowing water to enter. The algorithm keeps tracks of the connected components of water (connected components of pixels) using a union-find data structure with path-compression. The union-find data structure with path-compression supports quasi-linear time in the number of pixels [21,22,23]. More precisely, the time required can be shown to be bounded by $O(n\alpha(n))$, where n is the number of pixels and $\alpha(n)$ is the inverse of the Ackermann function, whose value is smaller than 5 if n is of the order 10^{80}. Hence the expression 'quasi-linear' is well motivated, although a true linear time algorithm for the union-find problem to our knowledge requires limiting assumptions such as for example the existence of a computer with $log(n)$ word length and incremental growing of the data structure [24].

In this paper, an algorithm is presented that can compute MSER and the so-called the component tree (of connected components as they evolve during the flooding), all in true linear time in the number of pixels. Perhaps even more importantly, the algorithm works with a single connected component of pixels, resulting in less memory usage and faster execution. The algorithm is natural and simple, sharing many properties with the previous algorithm, but with one critical difference. Instead of flooding the image with the same water level everywhere, the flooding occurs as if the landscape was opaque and water is being poured on at some arbitrarily selected point (pixel), see Fig. 1. The water first fills up the basin where water is poured on, and then spills over to other parts

Fig. 2. Typical progress stages for the algorithm on the first pass. The top left pixel is used as the source pixel. The flooding occurs, with preference for dark regions, MSER are detected in the process, and eventually the whole image is flooded. The process will then be repeated with preference for bright regions.

of the landscape as they become accessible to the current body of water. That is, the flooding occurs in a very physically plausible manner. The water adapts to the actual landscape and remains one connected component connected to the point where it is being poured onto the landscape. The algorithm keeps track of the 'downhill stream' of water, which at any time amounts to information for at most as many pixel components as there are grey-levels in the image. When the downhill stream finds a minimum, the water starts filling it up, and when it fills back up, the corresponding pixels are processed for MSER.

Thus, the progress of the algorithm can be roughly described as a physical flood-fill adapting to the landscape, see Fig 2. In the following section, we will first give an algorithm overview, followed by algorithm details and a description of a recommended actual implementation.

2 Algorithm Overview

In this section we will give a high-level description of the algorithm. For simplicity, we will work on the dark to bright pass. Note however that in reality, the MSER detection algorithm is simply carried out twice, one from dark to bright and one from bright to dark (with the landscape flipped upside down).

2.1 Basic Definitions

Let the image I consist of n pixels indexed by the variable x. Let each pixel be assigned the grey-level value $f(x)$, taken from a set of m linearly ordered grey-level values. Let also the pixels be assigned some neighboring relation encoded by the function $N(x)$, where $N(x)$ denotes the set of pixels neighboring pixel x. An example is a two-dimensional image with four-connected pixels taking on grey-levels $[0, \ldots, 255]$. The generalization to arbitrary undirected finite connected graphs is straightforward.

A *path* from a pixel x to y is a sequence of pair-wise neighboring pixels starting with x and ending with y. A *connected set* X is a set of pixels that has a path entirely within X between any pair of pixels in X. A *minimum* is a connected set X of pixels with equal grey-level values, such that no pixel in X is connected to a pixel outside X with equal or smaller grey-level value. A *sub-level set*, parameterized by grey-level y, is the set of pixels $\{x \in I \,|\, f(x) \leq y\}$ with grey-level smaller than or equal to y. As we trace through the sub-level sets by varying the grey-level from dark to bright, we get a sequence of sets on which we can (in principle) perform connected component analysis on. The way the connected components evolve defines a tree, which we will refer to as the *component tree*, defined roughly as follows. When a new component appears from one grey-level

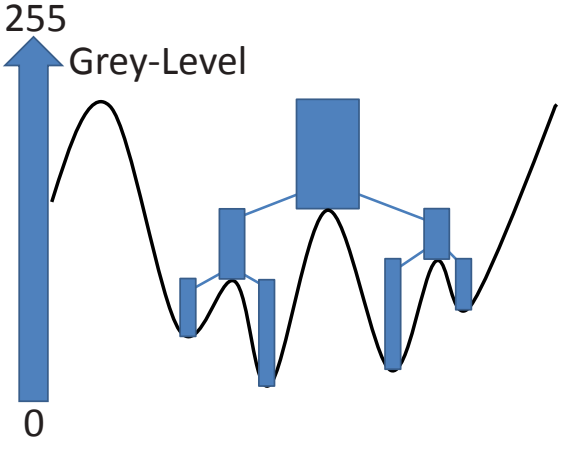

Fig. 3. The component tree with corresponding grey-levels. The nodes of the tree are drawn as vertical boxes where the vertical extent of the box shows the grey-level life span of each component.

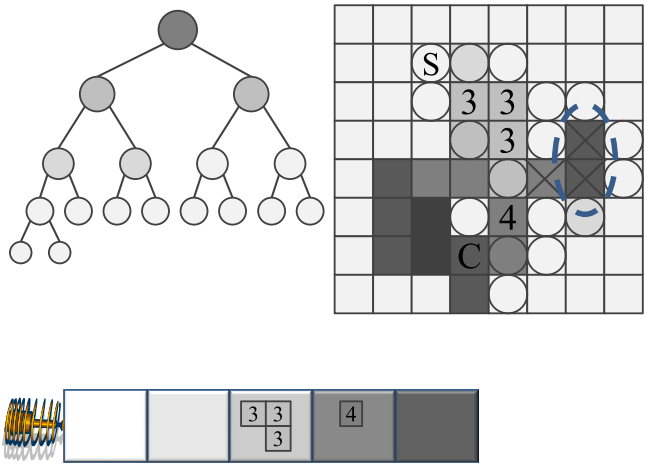

Fig. 4. The data-structures. Pixel status is shown on the top right. The source pixel is marked S. The current pixel is marked C. The pixels waiting in the heap of boundary pixels (left) are marked with circles. The pixels that belong to a component on the component stack (bottom) are marked with their component number. A detected MSER is shown by a dashed ellipse. Pixels that have been fully processed are marked with an X. Note how the stack holds components with decreasing grey-level.

to the next (in the sense that we get a component that consist entirely of pixels that were not in the previous sub-level set), this component of pixels is a minimum, which is made into a leaf node. When, as we increase the grey-level threshold, two or more components become joined into one, the joined component is assigned a new node and made into a parent of the original nodes. This process continues until the whole image is one component, which corresponds to the root node.

As the components evolve, we can also keep track of component information, which will differ dependent on what we wish to extract. For example, it could consist of a linked list of the pixels in the component. It could also just consist of the first and second order moments of the regions, which can in principle be encoded by six sums $\sum 1, \sum x, \sum y, \sum xx, \sum xy, \sum yy$ over the pixels in the component (although for numerical reasons, it is better to center these moments on something close to the centroid of the region). This is the output we need if we wish to turn the MSER into elliptical regions at the end of detection. An *MSER* is a component for which the relative growth-rate attains a minimum, see Matas [3] for further details.

2.2 Basic Algorithm

We will now describe the algorithm from an abstract point of view. See Section 2.3 on how to accomplish the steps efficiently, and Section 3 for analysis. The algorithm needs the following data-structures:

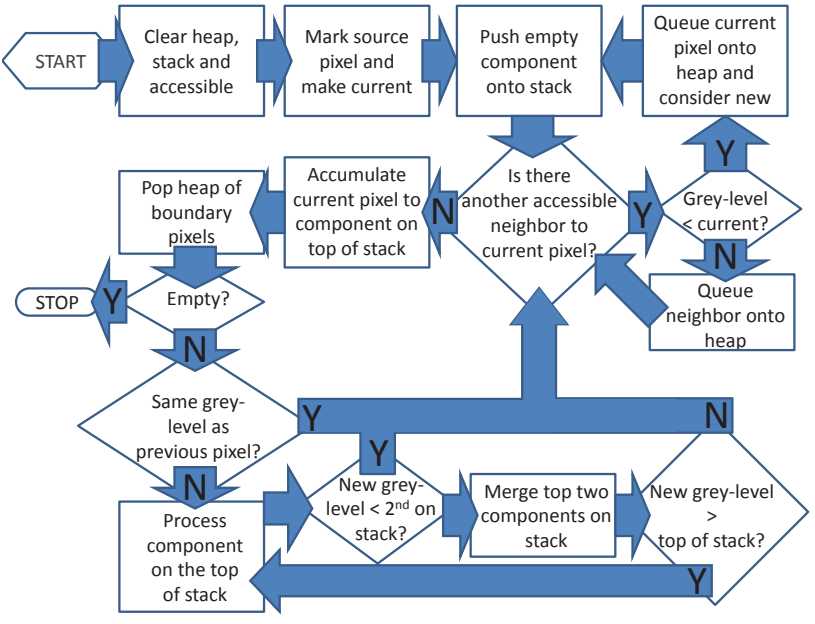

Fig. 5. State graph for the algorithm

- A binary mask of accessible pixels. These are the pixels to which the water already has access.
- A priority queue of boundary pixels, where priority is minus the grey-level. These pixels can be thought of as partially flooded pixels in the sense that water has access to the pixel in question, but has either not yet entered, or not yet explored all the edges out from the pixel. Along with the pixel id, an edge number indicating the next edge to be explored can be stored.
- A stack C of component information. Each entry holds the pixels in a component and/or the first and second order moments of the pixels in the component, as well as the size history of the component and the current grey-level at which the component is being processed. The maximum number of entries on the stack will be the number of grey-levels.

During execution of the algorithm, the components in the component info stack C may not correspond to components in the component tree. Rather, there will a number of components representing the 'down-stream' of water streaming downhill towards a minimum, where each component is the set of pixels at a single grey-level that is part of the down-stream. A single component represents the pixels covered by the water currently filling back out of a minimum. The algorithm in a sense has two states, one where the down-stream is flowing downhill in search of a minimum, and one where a minimum has been found, and the water level is currently rising out of it.

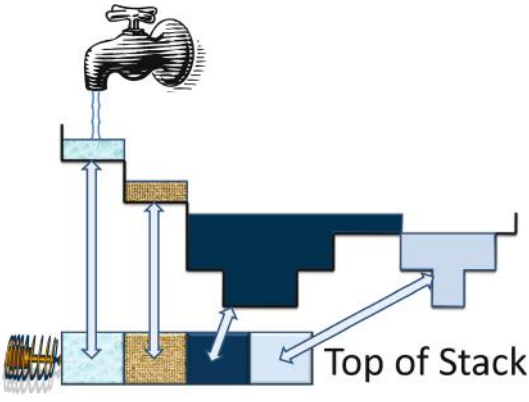

Fig. 6. The state of the component stack just before a merge of two components. The water has filled up one basin (dark blue component) and returned out of it to spill over into a second (light blue component) and the bright component is just about to merge with the dark component. Notice the textured components that are waiting as part of the 'downhill stream' of water.

To execute the algorithm, a pixel from which flooding will proceed is arbitrary chosen. This pixel can be thought of as the point at which water is being poured on, and the output result will be the same regardless of which pixel is selected, so we may simply pick the upper left corner of the image. We will refer to this as the source pixel. The algorithm is as follows, see also Fig 5:

1. Clear the accessible pixel mask, the heap of boundary pixels and the component stack. Push a dummy-component onto the stack, with grey-level higher than any allowed in the image.
2. Make the source pixel (with its first edge) the current pixel, mark it as accessible and store the grey-level of it in the variable *current_level*.
3. Push an empty component with *current_level* onto the component stack.
4. Explore the remaining edges to the neighbors of the current pixel, in order, as follows: For each neighbor, check if the neighbor is already accessible. If it is not, mark it as accessible and retrieve its grey-level. If the grey-level is not lower than the current one, push it onto the heap of boundary pixels. If on the other hand the grey-level is lower than the current one, enter the current pixel back into the queue of boundary pixels for later processing (with the next edge number), consider the new pixel and its grey-level and go to 3.
5. Accumulate the current pixel to the component at the top of the stack (water saturates the current pixel).
6. Pop the heap of boundary pixels. If the heap is empty, we are done. If the returned pixel is at the same grey-level as the previous, go to 4.
7. The returned pixel is at a higher grey-level, so we must now process all components on the component stack until we reach the higher grey-level. This is done with the *ProcessStack* sub-routine, see below. Then go to 4.

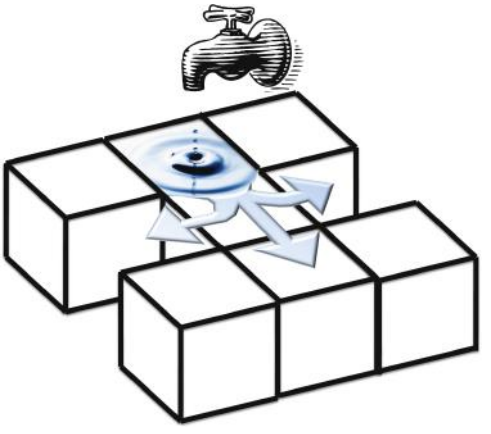

Fig. 7. For a ridge pixel, it is important for theoretical correctness of the algorithm that water is not allowed to 'spill' simultaneously into several directions. Therefore, if a lower pixel is found as any of the neighbors of a pixel, the current pixel has to be stacked for later and the newly found pixel tended to fully before proceeding to the other neighbors.

The *ProcessStack* sub-routine is as follows:
Sub-routine *ProcessStack(new_pixel_grey_level)*

1. Process component on the top of the stack. The next grey-level is the minimum of *new_pixel_grey_level* and the grey-level for the second component on the stack.
2. If *new_pixel_grey_level* is smaller than the grey-level on the second component on the stack, set the top of stack grey-level to *new_pixel_grey_level* and return from sub-routine (This occurs when the new pixel is at a grey-level for which there is not yet a component instantiated, so we let the top of stack be that level by just changing its grey-level.
3. Remove the top of stack and merge it into the second component on stack as follows: Add the first and second moment accumulators together and/or join the pixel lists. Either merge the histories of the components, or take the history from the winner. Note here that the top of stack should be considered one 'time-step' back, so its current size is part of the history. Therefore the top of stack would be the winner if its current size is larger than the previous size of second on stack.
4. If(*new_pixel_grey_level*>*top_of_stack_grey_level*) go to 1.

Here, the implementation of how to process a component follows the same lines as the standard algorithm, and depends on the information one wants about an MSER. It entails determining if a relative growth rate minimum has occurred, and if so, harvesting the required information from the component and declaring an MSER detected. The data-structure of a component holds information about the previous sizes of the component, and at which grey-levels they occurred.

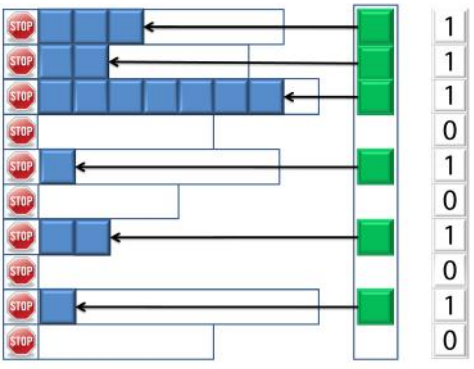

Fig. 8. The heap of boundary pixels can be efficiently implemented by a binary bitmask with number of grey-level bits, where bits are set if there are pixels at that grey-level, and a system of stacks, one for each grey-level.

It is immaterial to the result which order is used to explore the edges of a particular pixel, but it is important that if a lower grey-level pixel is found at the end of one of the edges, this pixel is tended to before any further edges are explored, see Fig 7. The reason is that otherwise water could enter a 'ridge' pixel that has several edges to lower grey-level pixels from distinct basins, and water would 'spill' into both basins simultaneously if we are not careful to process one of the edges first and fully saturate it before moving to the next edge. In practice, it is possible that this would not be a common problem, it would perhaps just make the algorithm 'myopic' in the sense that water could 'tunnel' through ridges, but by carefully saturating the edges in the correct order, our result will adhere exactly to the strict definition of MSER.

2.3 Implementation Details

In this section we give some implementation details. For efficiency, we implemented the heap of boundary pixel as a system of stacks of pixels at various grey-levels, see Fig 8. A bitmask is keeping track of which out of the 256 grey-levels have pixels waiting. This allows us to use a single instruction to find which is the smallest (or largest) occupied grey-level by using a machine instruction that retrieves the id number of the least significant or most significant bit set (for *x86* platforms *bsf* and *bsr*, or the more general 64-bit compliant *_BitScanForward*,*_BitScanReverse*). This is of course limited to as many grey-levels as the processor word-length, but for 256 grey-levels we simply chain multiple such instructions. This is essentially a way to get the hardware to do the work, and lower the time it takes to push and pop from the heap. This time is inherently constant with regards to the number of pixels, and logarithmic in the number of grey-levels. We also perform an explicit check for current grey-level before doing the general pop. This can be done since the

grey-level is never smaller than the current. When it is equal, which is common, some work can be saved.

The heap could be dynamically allocated and implemented as linked lists, but we prefer to count the number pixels at each grey-level in a single pre-sweep of the image, just like the standard algorithm does. This allows us to both pre-allocate and use fixed arrays for the stacks without stacks ever running into one another. The total number of entries in the stacks is the number of pixels plus the number of grey-levels (due to one stop-element for each stack).

Finally, to avoid having to perform explicit checks for the image boundary, a border of one pixel around the image is used, and the accessible mask is always set when a sweep starts. This can actually be implemented without explicitly sweeping the mask except for in a detector initialization step, since the sweep sets the whole mask. We then simply flip the border bits and invert the meaning of the mask on the dark-to-bright sweep, finally leaving the mask in its original state.

3 Analysis

An informal analysis for the algorithm is straightforward. It is easy to see that only one copy of a pixel can ever be on the heap of boundary pixels, and that a pixel can only return onto the heap at most as many times as it has neighbors. Moreover, the components that are processed must have at least one new pixel in them, so the total amount of component processing is linear in the number of pixels. As it takes at most $\log(m)$ time to enter or pop a pixel from the heap, the worst case execution time is therefore bounded by

$$O((n + e) \log(m)), \tag{1}$$

where n is the number of pixels, m the number of grey-levels, and e is the number of edges in the image graph (such as $e \approx 2n$ for four-connected images). If we regard m and the connectivity as constants, this simply amounts to $O(n)$, linear time.

The memory usage is roughly a four-byte integer per pixel, which allows both the heap of boundary pixels, the accessible mask bit and the edge number to be stored while allowing a sufficient number of bits for the pixel coordinates. Storage for only m components is required (apart of course from the storage required by detected MSER, which would clearly be needed for any algorithm.

4 Experimental Validation

As the results of our detector are the same as the original MSER detector, which has been found in several studies to have excellent repeatability, our experimental focus is on execution time. As a reference, the recent paper [19] reports on an FPGA implementation with careful attention to detail, but based on the standard union-find algorithm, performing at 25 fps on images up to 350×350

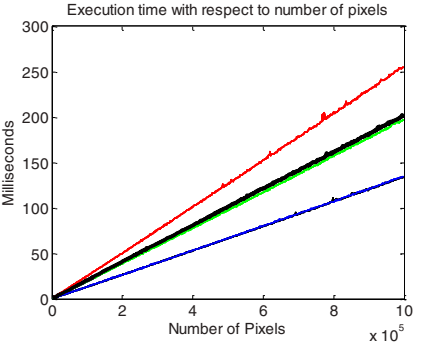

Fig. 9. Execution time as a function of image size on square images ranging from one pixel to one mega-pixel. Curves are shown, from fastest to slowest: for a blank image and an image filled with ramps (blue and black, almost identical), a pixel-size checkerboard pattern (thin green line), natural image (thick black line) and a random image (thin red line). Note the completely linear timings, which include both passes.

pixels or 54 fps detection on image streams with 320×240 pixels resolution. These execution times are for a single pass, so with both passes (bright-to-dark and dark-to-bright), this corresponds to 1.5 or 2 mega-pixel per second. Our implementation of the new algorithm, which is relatively carefully written but not fully optimized, is running at least twice as fast as this, at around 5 mega-pixel per second on a single core consumer grade CPU.

We ran experiments with single-threaded code on a laptop with 3GB of RAM and an Intel T7400 2.16GHz processor. To try to show the 'best' and 'worst' realistic performance of the algorithm, we give execution times with respect to the number of pixels in Fig 9 for various synthetic images and natural images. To avoid process scheduling noise, results were produced by running five independent times

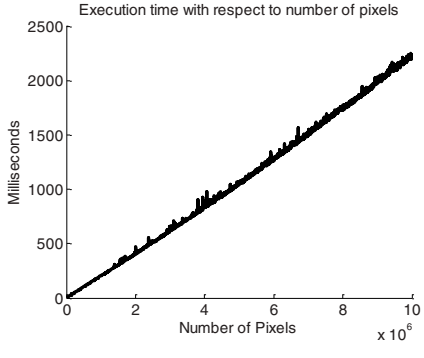

Fig. 10. Actual execution times up to ten mega-pixel. A tiny amount of curvature can be perceived, but the performance is very close to the theoretical linear.

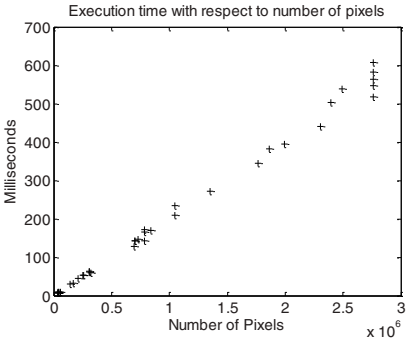

Fig. 11. A scatter plot of execution times with respect to number of pixels for a number of randomly selected photographs with random resolution. Note the clear linear trend of around five mega-pixel per second.

and taking the fastest run. Blank or smooth images execute the fastest, while completely random images execute the slowest. There is approximately a factor two difference between the two, which likely is connected to how many times each pixel is put back on the heap. It is worth noting that random images and checkerboard images are particularly challenging to the standard algorithm since it causes half as many concurrent components as there are pixels in the image. A natural image is typically somewhere in between. To provide a reproducible reference speed-curve, the natural image curve in this figure was produced by running on the 512×512 Lena image wrapped endlessly. The linearity persists to very large images, Fig 10 shows the timings up to ten mega-pixel.

A more general measurement of execution time for natural images is shown in Fig 11, where the execution times for randomly selected images of varying size are given. Note the clear linearity and the typical speed of roughly 5 mega-pixel per second on natural images.

5 Conclusions

We have described a new MSER detection algorithm, which runs in linear time and behaves like a true flood-fill, meaning a flood-fill where the next pixel filled must be the lowest point accessible so far. The performance tests validate the linear run-times and indicate the speed on a single core consumer grade CPU to be at around five mega-pixels per second for natural images, roughly the pixel rate of a PAL TV signal and more than twice as fast as a very recent FPGA implementation. The algorithm shows that MSER detection does not necessarily need the union-find implementation commonly used for watershed segmentation. This allows true linear time and opens up for further studies into parallelization of the algorithm.

References

1. Tuytelaars, T., Gool, L.V.: Matching widely separated views based on affine invariant regions. International Journal of Computer Vision 59(1), 61–85 (2004)
2. Schmid, C., Mohr, R., Bauckhage, C.: Evaluation of interest point detectors. International Journal of Computer Vision 37(2), 151–172 (2000)
3. Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide baseline stereo from maximally stable extremal regions. In: British Machine Vision Conference, pp. 384–393 (2002)
4. Mikolajczyk, K., Schmid, C.: Scale and affine invariant interest point detectors. International Journal of Computer Vision 60(1), 63–86 (2004)
5. Mikolajczyk, K., Tuytelaars, T., Schmid, C., Zisserman, A., Matas, J., Schaffalitzky, F., Kadir, T., Gool, L.V.: A comparison of affine region detectors. International Journal of Computer Vision 65(1–2), 43–72 (2005)
6. Kadir, T., Zisserman, A., Brady, M.: An affine invariant salient region detector. In: European Conference on Computer Vision, pp. 404–416 (2004)
7. Lowe, D.: Distinctive image features from scale invariant keypoints. International Journal of Computer Vision 60(2), 91–110 (2004)
8. Lindeberg, T.: Feature detection with automatic scale selection. International Journal of Computer Vision 30(2), 77–116 (1998)
9. Triggs, B.: Detecting keypoints with stable position, orientation and scale under illumination changes. In: European Conference on Computer Vision, vol. 4, pp. 100–113 (2004)
10. Rothganger, F., Lazebnik, S., Schmid, C., Ponce, J.: 3d object modeling and recognition using local affine-invariant image descriptors and multi-view spatial constraints. International Journal of Computer Vision 66(3), 231–259 (2006)
11. Donoser, M., Bischof, H.: Efficient maximally stable extremal region (mser) tracking. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 553–560 (2006)
12. Brown, M., Lowe, D.: Invariant features from interest point groups. In: British Machine Vision Conference, pp. 656–665 (2002)
13. Harris, C., Stephens, M.: A combined corner and edge detector. In: Proceedings of the 4th Alvey Vision Conference, pp. 147–151 (1988)
14. Sivic, J., Zisserman, A.: Video google: A text retrieval approach to object matching in videos. In: International Conference on Computer Vision, vol. 2, pp. 1470–1477 (2003)
15. Nistér, D., Stewénius, H.: Scalable recognition with a vocabulary tree. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), vol. 2, pp. 2161–2168 (2006)
16. Obdrzalek, S., Matas, J.: Object recognition using local affine frames on distinguished regions. In: British Machine Vision Conference, vol. 1, pp. 113–122 (2002)
17. Forssén, P.E.: Maximally stable colour regions for recognition and matching. In: IEEE Conference on Computer Vision and Pattern Recognition (2007)
18. Donoser, M., Bischof, H.: 3d segmentation by maximally stable volumes (msvs). In: ICPR 2006: Proceedings of the 18th International Conference on Pattern Recognition, pp. 63–66 (2006)
19. Kristensen, F., MacLean, W.: Fpga real-time extraction of maximally-stable extremal regions. In: IEEE International Symposium on Circuits and Systems (2007)
20. Vincent, L., Soille, P.: Watersheds in digital spaces: an efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence 13, 583–598 (1991)

21. Roerdink, J., Meijster, A.: The watershed transform: definitions, algorithms and parallelization strategies. Fundamenta Informaticae 41, 187–228 (2000)
22. Couprie, M., Najman, L., Bertrand, G.: Quasi-linear algorithms for the topological watershed. Journal of Mathematical Imaging and Vision 22(2), 231–249 (2005)
23. Tarjan, R.: Data Structures and Network Algorithms. SIAM, Philadelphia (1983)
24. Gabow, H., Tarjan, R.: A linear-time algorithm for a special case of disjoint set union. Journal of Computer and System Sciences 30(2), 209–220 (1985)