Template Matching in Arbitrary Time

Robert Finis Anderson

Department of Computer Science
University of Texas at Dallas
Richardson, TX USA

Abstract

The problem of finding a match for an image ('template') within a larger image is known as template matching. It is used in the lower levels of many key computer vision applications. Unaccelerated template matching is unfeasibly slow in most cases, therefore many algorithms have been created to accelerate it. Currently known template matching acceleration algorithms run in fixed time but without the guarantee of finding a best match or are guaranteed to find the best match but without a guaranteed runtime. We present an algorithm which is a novel combination of both. Our algorithm, in effect, adjusts dynamically to available computing power to find the best answer under the time constraints. It does this by finding a series of more and more tightly bounded approximate matches on the way to finding a guaranteed best match.

Keywords: Machine Vision, Template Matching, Machine Vision Applications, Real Time

1. Introduction

Template matching is a building block for many high level Computer Vision applications, such as face and object detection [1, 2, 3, 4, 5], texture synthesis [6], image compression [7, 8, 9, 10], and video compression [11, 12, 9]. Its runtime is often unfeasibly slow in raw form [13] and there has been much research into methods for accelerating template matching for various applications. These methods can be viewed as occupying one of two groups.

Email address: rfa061000@utdallas.edu (Robert Finis Anderson)

In one group, algorithms are capable of running in fixed time but are able to guarantee finding the best match according to the chosen error measure (for example [14, 15, 16, 17, 18, 19]). We will call these methods "approximate" for convenience. Recent research into template matching has produced a second group which guarantees finding the best match [20, 21, 22, 13, 23, 24], but whose runtime varies. We will call these algorithms "exact" for convenience. We observe that the runtime of these methods depends not only on the size of the search images and templates, but also their content, making predicting their runtimes difficult. The lone exception to this rule are Fast Fourier Transform (FFT) based algorithms, which accelerate the convolution or correlation of the image and template by performing it in the spectral domain [25]. w These methods typically only operate efficiently when the template is large compared to the image size [23].

1.1. Motivation for our algorithm

In a hard real-time environment all tasks must be completed within a given time limit, and it is the responsibility of a scheduler to ensure that this happens [26]. If the amount of time required for a task is unpredictable, the system must be designed assuming that the task will run in worst-case time to guarantee that the task will complete before the deadline [27]. Therefore, the greater the difference between average and worst-case run-time, the more CPU time will be wasted. Previous work in exact acceleration algorithms did not take this into consideration. According to results shown in [20] the cost of matching a 64x64 template to a location in a 512x512 image varied by a factor of over 1000 depending on noise level and image content. The results of [22] show variations in run-time by a factor of 50, and [13] shows variations of 10 or more depending only on noise. This would clearly lead to a lot of wasted processing power in a real-time setting.

Recently, there has been strong interest in computer vision algorithms running on 'non-standard' hardware. Modern implementation hardware might vary from a high-powered desktop PC [18, 23, 28] to a custom hardware solution [29, 30] to a mobile type processor with limited memory [31, 32], to a massively parallel GPU [33, 34]. Since application hardware varies substantially in both speed and design, it would be valuable if an algorithm adjusted to the power of its implementation hardware.

To rectify this, we present an algorithm which satisfies the goal of guaranteed run-time in template matching. It does this by taking advantage of available computing power to produce the best answer it can find within an

allotted time. In many cases the algorithm finds the best possible match, and otherwise returns a close approximation. The algorithm does this by keeping an 'answer set' of current potential best matches. When the algorithm is stopped, it evaluates the current 'answer set' and returns the best match from that set. Given enough computing power or time, our algorithm is guaranteed to find the best match.

Other algorithms can be pre-tuned to run faster and less accurately, or slower and more accurately (e.g. [35]), and it is always possible to subsample the search space with a corresponding loss in detail. However, these algorithms do not adjust dynamically to find the best answer in the current time limit, on the current data. This is the first algorithm to progressively search for the best possible match while maintaining the capability of running in fixed-time, while also being adaptable to differing input data on the fly.

This algorithm is an extension and elaboration of work presented in [36]. This paper is organized as follows. Section 2 describes the workings of the algorithm, including a brief proof of its correctness. Section 3 measures its performance in average cases for different situations. Lastly, we present concluding remarks and directions for future developments.

2. The Arbitrary Time Algorithm

Throughout this paper we make use of the l_2 norm based distance measure (i.e. the Euclidean distance) between template and image subwindow. We denote the l_2 norm of a vector x by $||x||$.

Let the template to be detected be represented by a vector $\lambda \in \Re^n$. This vector is formed by concatenating the rows of the template image together into one long sequence. We consider each subwindow y_i of the search image I a potential match. The subwindows may overlap, and all contain n pixels. For convenience we define $Y = \{y_1, y_2, \ldots y_m\}$ to be the set of all potential matches. In practice m (the number of potential matches) is slightly less than the number of pixels in I.

The error for the i^{th} candidate (or sub-window) is:

$$E_i = ||\lambda - y_i||^2$$

Given λ and I, a template matching algorithm attempts to find the y_i which minimizes E_i. Minimizing E_i is equivalent to minimizing $||\lambda - y_i||$ for real numbers, and is computationally less expensive. In the next section, we

briefly review the idea of using two bounds to accelerate template matching, proposed in [37].

2.1. Bounds

Let the projection of λ and y_i onto orthogonal subspaces W_d and C be y_i^w, λ^w and y_i^c, λ^c. Applying the triangle inequality

$$| x | - | z | \leq | x - z | \leq | x | + | z |$$

to the rightmost term in

$$||\lambda - y_i|| = ||\lambda^w - y_i^w|| + ||\lambda^c - y_i^c||$$

we get:

$$||\lambda^w - y_i^w||^2 + (||\lambda^c|| - ||y_i^c||)^2 \leq ||\lambda - y_i||^2 \tag{1a}$$
$$||\lambda - y_i||^2 \leq ||\lambda^w - y_i^w||^2 + (||\lambda^c|| + ||y_i^c||)^2 \tag{1b}$$

Therefore, if we split λ and y_i into orthogonal subspaces W_d and C, this allows us to calculate upper and lower bounds on the error E_i, while performing very few calculations in the subspace C. Note also that since $x^w \cdot x^c = 0$ for arbitrary x, we also have

$$||\lambda^c||^2 = ||\lambda||^2 - ||\lambda^w||^2 \tag{2}$$

2.2. Orthogonal Subspaces

Let us define W' to be a set of basis vectors for the space $\Re^n$ inhabited by y_i and λ. We impose an arbitrary ordering on W'. Then W_d represents the first d vectors from the set W'. We will call these vectors 'kernels' in keeping with other work in this field, i.e. [22]. Then C represents the orthogonal complement of W. During the course of the algorithm, each y_i is progressively projected onto more and more of the set W' (or, stated another way, d is increased a step at a time for each y_i individually). After each of these projections, we can quickly calculate an updated value for C using Eq. 2. For simplicity of notation, we define $l_d(y_i) = ||\lambda^w - y_i^w|| + (||\lambda^c|| - ||y_i^c||)$ and $u_d(y_i) = ||\lambda^w - y_i^w|| + (||\lambda^c|| + ||y_i^c||)$ as the lower and upper bounds after the projection of y_i and λ on the first d kernels. When we write $l(y_i)$ without an explicit value for d, we mean the highest d currently evaluated for y_i.

For our algorithm to work efficiently, it should be possible to quickly compute the projections of the subwindows on the kernels. Additionally,

projections on only a few kernels should tell us a great deal about the contents of the subwindows, as well as offering strong energy compaction (effectively, the kernels should be good candidates for lossy image compression). This second condition also has the effect of quickly tightening the bounds described above.

We currently make use of the Walsh Projection Kernels [21, 24, 22, 37], as that set of kernels satisfies the above requirements. Further, there have been recent advances in methods for computing the Walsh transform quickly, in as little as 2 operations per sub window per kernel [24, 22]. We compute those Walsh Kernels using the "Corner-Based" method described in [38], as it performs especially well in the case of template matching [38]. This method is based on integral-images" [39].

2.3. Initialization

At initialization the algorithm requires three parameters: k_{max}, k_{min} and d_0. The value of k_{max} represents the maximum number of potential best matches the algorithm tracks at any one time, and is unique to this work. The value of k_{min} represents the number of match locations we want the algorithm to return. This value might be greater than one as because the input image may contain more than one instance of the template. The value of d_0 represents a starting value for d in W_d - all subwindows y_i are initially projected onto this starting W_{d_0} before being updated individually during the run of the algorithm (after [37]). The value of d_0 is not strongly data dependent, and can effectively be set to 0 for all purposes.

The algorithm initialization follows these steps: The algorithm first computes the projections of all y_i onto W_{d_0}. Next, it computes the corresponding $l_{d_0}(y_i)$ and $u_{d_0}(y_i)$ for each y_i. Then algorithm then separates the data into two sets A and B where $A, B \subset Y$, and $A \cap B = \emptyset$. The algorithm initializes the value of k to k_{max}. If y_k is the y_i with the k^{th} smallest value of $l(y)$, then

$$A = \{\forall y_i \text{ s.t. } l(y_i) \leq l(y_k)\} \tag{3}$$

$$B = Y - A \tag{4}$$

Additionally, if there exists a $l(y_i) = l(y_k)$ and there are already k elements in A, one is arbitrarily chosen and placed in B. These initialization steps are generally similar to those in [36, 37].

At initialization, $|A| = k_{max}$. At this point, A represents a list of potential matches which are most likely to be the optimal match. Maintaining a 'short-list' of best candidates is what allows the algorithm to stop at any time

with a good estimate of the best match. As the algorithm runs, that list is shortened until it reaches k_{min} or until the algorithm is early terminated. These properties of A and B are maintained during the entire run of the algorithm. The algorithm also chooses a y_u and a y_{k+1}, where

$$y_u = \arg\max_{y_i} u(y_i)_n \text{ s.t. } y_i \in A$$

and

$$y_{k+1} = \arg\min_{y_i} l(y_i)_n \text{ s.t. } y_i \in B$$

They thus represent the candidate with the largest upper bound in A, and the candidate with the lowest lower bound in B, respectively. The initialization steps are represented in lines 1-7 in fig 1.

2.4. Iteration

Each iteration of the algorithm follows these steps: The algorithm first checks to see if the 'early termination' flag has been set, or if the 'done' flag has been set (line 8). If the early termination flag is true, the algorithm moves into the EarlyTerm subroutine described in 2.5. If the 'done' flag is true, the algorithm moves into the FindYBest subroutine (line 25), which simply computes E_i for all $y_i \in A$, and then returns the y_i which minimizes E_i (which we call y_{best}).

If neither of these conditions is true, the algorithm checks if $u(y_u) \leq l(y_{k+1})$. If so, this means that the set A represents the k smallest items, so the algorithm checks if $k = k_{min}$. If this is true, this means that the answer consisting of the k_{min} best matches is contained in A, and the 'done' flag is set (meaning the best answer has been found). If $k > k_{min}$, then the $y_i \in A$ with the largest lower bound is removed and placed in B, and k is decremented. Note that this is *not* necessarily y_u.

If none of the preceeding conditions are true, the algorithm projects y_u (which has already been evaluated on W_d) on W_{d+1} and computes $l_{d+1}(y_i)$ and $u_{d+1}(y_i)$. Since we have the projection y_i^w on W_d, then projection of y_i on W_{d+1} is equivalent to y_i^w plus the projection of y_i on the $d + 1^{\text{st}}$ vector. Using this fact along with eq. 2 we are able to quickly calculate $l_{d+1}(y_i)$ and $u_{d+1}(y_i)$ using $l_d(y_i)$ and $u_d(y_i)$ and the projection of y_i on W_{d+1}. At this point the algorithm checks whether $l(y_u) > l(y_{k+1})$. If this is true, then the algorithm calls the Swap(y_u, y_{k+1}) function which removes y_u from A and y_{k+1} from B, then places y_u in B and y_{k+1} in A. This step maintains the

'short-list' property of A. The algorithm then finds the current values for y_u and y_{k+1}. If it finds that $l(y_u) \leq l(y_{k+1})$, the algorithm merely finds the current value of y_u. These steps are outlined in pseudo code in Fig. 1.

2.5. Arbitrary Time

If the condition $u(y_u) > l(y_{k+1})$ on line 9 is false, indicating that $u(y_u) \leq l(y_{k+1})$, this means that we are guaranteed that $\forall y_i \in A$, $\forall y_j \in B$ $E_i \leq E_j$. Therefore A must contain the k_{min} y_i which minimize E_i. Thus we evaluate E_i for all A on line 25. If this is not the case, but the early termination flag is true, we can assume two things- first, it is likely that $Y_{\text{best}} \subseteq A$, but this is not guaranteed. Second, we have very little time to find a good approximation for Y_{best}.

Therefore, if the early termination flag is set, the algorithm computes an approximate answer set based on the short-list principle of A. Let

$$y_{\text{approx}} = \arg\min_{y_i} e(y_i), y_i \in A \tag{5}$$

where $e(y_i)$ is the expected value of y_i, defined as the average of the lower and upper bounds on $E(y_i)$:

$$e(y_i) = \frac{u(y_i) + l(y_i)}{2} = |\lambda_a - y_a|^2 + |\lambda_b|^2 + |y_b|^2 \tag{6}$$

This is represented on line 24 of fig 1. Since, in practice, we minimize $2e(y_i)$, this equates to a single arithmetic operation per candidate in A since we already have $u(y_i)$, $l(y_i)\, \forall y_i \in A$. Finding the minimum value in A consumes another arithmetic operation (plus one potential swap operation) per candidate. These steps can be combined for a total cost of at most $3k$ operations, independent of the input data. If $k_{min} > 1$, finding the k_{min} smallest in A requires a last step, which can be done using the Select algorithm in $O(k)$ time. Since in most practical cases $k \ll m$ this cost is negligible. When the expected value $e(y_i)$ is in line with the actual value of $E(y_i)$, the minimum expected value will be the best match currently in A. Experimental evidence shows that this is typically very close to the optimal match (see Sec.3).

Allowing for a $k_{max} > k_{min}$ (as opposed to a single value as in [36, 37]) has multiple benefits. Due to the imperfect nature of any bounding method, the k_{min} candidates with the lowest E_i values may not have the k_{min} lowest l_i initially. So, first, in early termination, this allows for a broader search space during the early termination phase. Secondly, for the same reasons, allowing

```
ARBITRARY TIME ALG.(λ, Y, k_max, k_min, d_0)
 1  CalcInitialKernels(Y, d_0)
 2  A ← InitA(Y, k)
 3  B ← InitB(Y, A)
 4  y_u ← Findy_u(A)
 5  y_{k+1} ← Findy_{k+1}(B)
 6  done ← False, earlyTerm ← False
 7  k ← k_max
 8  while !earlyTerm and !done
        do
 9          if u(y_u) ≤ l(y_{k+1})
10            then if k > k_min
11                    then A ← A − {y_k}
12                         B ← B ∪ {y_k}
13                         k − −
14                         y_u ← Findy_u(A)
15                    else  done ← True
16            else
17                  l(y_u), u(y_u) ← Project(y_u, W_{d+1})
18                  if l(y_u) > l(y_{k+1})
19                    then
20                         Swap(y_u, y_{k+1}, A, B)
21                         y_{k+1} ← Findy_{k+1}(B)
22                  y_u ← Findy_u(A)

23  if earlyTerm
24     then Y_best ← EarlyTerm(A, k_min)
25     else  Y_best ← FindYBest(A, k_min)

26  return Y_best
```

Figure 1: The Arbitrary Time Algorithm

for a larger k_{max} can allow for less swapping between the sets A and B if the value of k_{max} is chosen well. Setting k_{max} too high can result in excessive swapping. The effect of the value of k_{max} on the accuracy and performance of the algorithm is explored in 3.

2.6. Analysis and Correctness

Conceptually the body of the algorithm is trying to separate the sets A and B; as soon as it is shows that $\forall y_i \in A, \forall y_j \in B,\ E_i \leq E_j$ and that $|A| = k_{min}$, it terminates automatically.

Lemma. The algorithm is guaranteed to find the candidate with the lowest match value if allowed to run to completion.

Proof. (Sketch) At the end of each iteration in the algorithm, there are guaranteed to be k candidates in A. Recall that $l(y_{k+1})$ represents the $k + 1^{\text{st}}$ lowest lower bound out of all the potential matches. Additionally note that $\forall y_i \in A,\ E_i \leq u(y_u)$; thus we are always guaranteed at least k matches in A with an error below $u(y_u)$. Therefore, if the condition on line 9 in fig 1 is satisfied (that is, $u(y_u) \leq l(y_{k+1})$), then it must be true that $\forall y_i \in A, E_i \leq u(y_u) \leq l(y_k + 1) \leq E_{k+1}$. Thus we have k candidates guaranteed to have a match value less than or equal to the value of the $k + 1^{\text{st}}$ smallest match value. If $k = k_{min}$, this gives us our answer set. Suppose $\exists y_i \in B, y_j \in A \text{s.t.} E_i < E_j$ and the above condition holds. This would mean that $l(y_i) < l(y_{k+1})$. Therefore y_i is guaranteed to be in A by the condition in eq. 3. Since $A \cap B = \emptyset$, this is clearly impossible since y_i is already in B[1]. The procedure on line 25 guarantees that the algorithm returns the smallest among these k. □

In terms of memory the algorithm stores at most a single copy of each candidate, with the associated numerical bounds. While all the kernel projections of the template are typically pre-computed and stored, the projections of the various candidates are evaluated once and the magnitude of their variation from the template is evaluated before the projection is discarded. Thus the overall memory complexity of the algorithm is $O(m)$ assuming that the template is much smaller than the image ($n \ll m$).

[1]It may be true that $\exists y_i \in B, y_j in A \mid E_i = E_j$. This cannot be avoided, as there may be multiple candidates with the same match value, though in practice this is quite rare.

The worst case run-time is more difficult to analyze. The algorithm is guaranteed to terminate, given that one of the following two conditions holds:

a) $\exists d \,\text{s.t.}\, l_d(y_i) = E_i = u_d(y_i)$

b) We default to calculating E_i explicitly when d is greater than a certain threshold $\mathcal{D}$.

Although the first condition is true for Walsh Projections, we make use of the latter. Even the best methods for calculating the Walsh Projections require at least 2 operations per candidate per kernel [22], and thus it is currently faster to default to direct computation at some point. We call this maximum number of projections $\mathcal{D}$. In practice E_i is computed for very few candidate locations, and it is rare that any of these locations are outside A when the algorithm terminates.

Lemma. The algorithm is guaranteed to terminate.

Proof. (Sketch) At every iteration the algorithm must either break out the loop, remove an item from A, or update the bounds of a given y_i to a higher d (line 17), and possibly swap a candidate between A and B (line 20). First, note that there cannot be an unlimited number of updates — as noted previously, there is always a value for d for which the bounds are equal to the actual match value, and the candidate cannot be updated anymore (which we call $\mathcal{D}$). This means that the update step (line 17) can happen at most $O(m\mathcal{D})$ times. At this point, $\forall y_i \, l(y_i) = E_i = u(y_i)$, and it is trivial to show that the condition $u(y_u) > l(y_{k+1})$ on line 9 is false, and to find the k_{min} smallest amongs the matches. □

The average run time of the algorithm is examined experimentally in Sec 3, and is shown to be much lower than the upper bound given above.

2.7. Implementation

Our implementation of the algorithm uses a heap data structure to represent the set B described in section 2, as it is a natural fit for finding the minimum of a large set. Heap construction is $O(m)$, and removal of the min (or max) node, or insertion of a new node, is only $O(\log(m))$. The set A is implemented using a custom list-based data structure, which allows A to be sorted according to both $u(y_i)$ and $l(y_i)$ while also allowing rapid removal of an item. We call this structure an s-sorted linked list. Given an s-sorted

linked list containing n elements, and sorted according to s different metrics, removal takes $O(s)$ time. Updating an element in place takes time proportional to the number of positions moved in the list, which is typically small since the updates used in this algorithm are fairly fine grained. Insertion of elements can be slower, since the position for an element must be found in each ordering. In our case, we are only inserting elements at the end of the list, according to their lower bound values, and their upper bound values are likely to be similarly near the end of that ordering.

The Walsh projections are implemented using integral-image based methods, first proposed in [28], and explored further in [38]. A Walsh kernel w is composed of rectangular regions (each of which is uniformly positive or negative). The projection of the image p (of the same dimensions as w) on w can be expressed as follows:

$$\begin{aligned} p'w = \sum_{i,j} p(i,j)w(i,j) &= 2 \sum_{w(i,j)=1} p(i,j) - \sum_{i,j} p(i,j) \\ &= 2R_w - \alpha_{00} \end{aligned}$$

where α_{00} is the projection of p on the first Walsh kernel and R_w is the sum of the pixel values in p over the rectangles of value "1" (white rectangles) in w. These sums are computed in three operations per rectangular region using integral images. The complexity of this method is therefore proportional to the number of rectangular regions in a given Walsh kernel. We have performed studies comparing this straightforward method to the accelerated methods in [24, 22], and found that in the context of template matching, our method is considerably faster [38]. We order the Walsh Kernels by 'sequency' [40], a concept related to visual frequency in 2 dimensions. Sequency is used because lower sequency kernels are both easier to compute, and typically extract more information from the image than higher sequency kernels. This is in no way required by the algorithm, and alternative orderings have been explored [22].

2.8. Cost Analysis

To clarify analysis, we break down the costs of the algorithm as follows. We count the average number of Walsh Kernels evaluated at each location in the image $P = \frac{1}{m} \sum_{\forall y_i} d(y_i)$. Additionally we compute the average number of basic mathematical operations used to compute those kernels, $\bar{c}$. If $c(d)$ is the cost of computing the d^{th} kernel, then $\bar{c} = \frac{1}{P} \sum_{\forall y_i} d(y_i)c(d(y_i))$ We

also compute the average cost of explicit computation without projection (i.e., computing at $\mathcal{D}$ as described in Sec. 2.6). Define f as the number of candidates computed explicitly, and recall that there are n pixels per subwindow. Given that it takes 3 operations per pixel to calculate $E_i = \mid \lambda - y_i \mid^2$, we define this cost as $\bar{f} = \frac{1}{m}3nf$. Lastly we count the number of heap swaps required to construct and maintain A and B. We record the number of swaps during construction as

$$\hat{q} = \frac{5}{m}\text{swaps-construct}$$

The number of swaps used to maintain the heaps is

$$\bar{q} = \frac{5}{m}\text{swaps-maintain}$$

We multiply $\hat{q}$ and $\bar{q}$ by a constant (in our case, 5), because in our experiments the heap swap operation is correspondingly more expensive than the basic mathematical operations used in the other steps.[2] The sum of these values, plus the initialization cost of the integral images used to compute the Walsh kernel projections (a constant, 5 operations per pixel) is the total cost per candidate.

3. Experimental Results

To test the algorithm we made use of an experimental framework similar to that in [24]: for each image in our testing database (all of size 512x512), we evaluate the image using the Harris Edge Detector, and extract five random templates with high scores (strong corner features) of size 64x64. We then add zero mean Gaussian noise with σ varying from 5 to 75 to the templates and attempt to match them with their originating images. We then average the results over all templates and all images for each noise level. This experiment was repeated for various settings of k_{max} while holding k_{min} constant at 1. For each of these settings, we early-terminate the algorithm after a predetermined number of operations to simulate a real-time situation. In its

[2]The cost of a heap swap is dependent on many factors, including the size of the processor cache, the size of the heap, and the clock speed and latency of main memory. Specialized hardware, or fast memory, can reduce this cost. On different pieces of typical PC hardware we have seen this cost vary from 3 to 15.

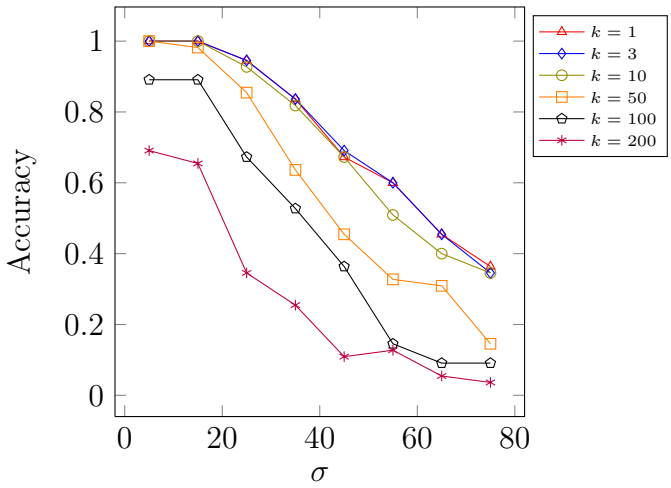

Figure 2: Average accuracy after allowing the algorithm to run for 25 operations pixel. Noise varies from $\sigma = 5$ to 75.

current form the algorithm needs at least 15 operations per pixel to perform a meaningful search, since creating the queues and calculating the integral images consumes approximately 12 operations per pixel.

For the purposes of describing the accuracy of the algorithm, we define y_{exact} to be the correct, best match in the image, and y_{approx} to be the answer the algorithm returns after early termination. We then measure the distance between y_{approx} and y_{exact} in terms of the Euclidean distance between their locations in the image. If the total overlap in area between y_{exact} and y_{approx} is greater than 80% of the total area of y_{exact}, we say that the approximate match is accurate. This is the approach adopted by [18] in measuring the accuracy of their approximation algorithm. An accurate answer in this context typically means that y_{approx} is within a few pixels of y_{exact}.

In figs. 2-4 the x axis represents the standard deviation (σ) of the noise added to the template, and the y axis represents the ratio of correct answers to total answers (1.0 is a perfect score). As can be seen in figs. 2, the algorithm performs quite well with small k with only 25 operations per pixel up to noise $\sigma = 35$. Beyond this level, the average accuracy falls steeply. For comparison, the state of the art exact method described in [23] requires 200-300 operations per pixel to find the correct answer at $\sigma = 25$.

When the algorithm is allowed to run to 35 operations per pixel (figs. 3) performance is strong up to $\sigma = 55$. For contrast, the exact method in [23]

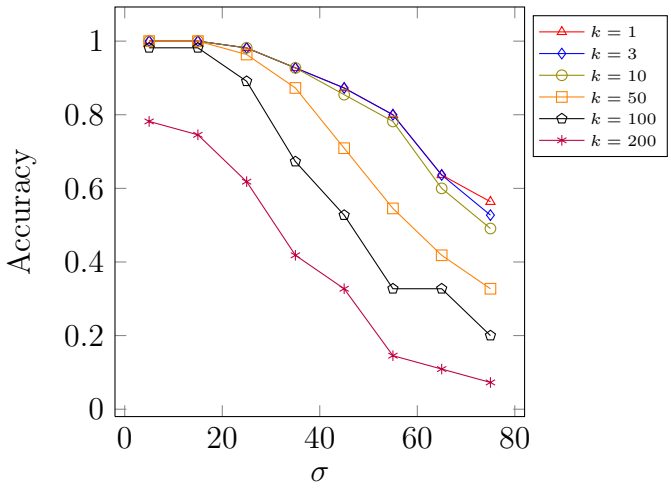

Figure 3: Results of allowing the algorithm to run to 35 operations per pixel, with noise varying from $\sigma = 0$ to $\sigma = 75$

requires over 1000 operations per pixel at these noise levels.

Allowing 200 operations per pixel guarantees strong performance accross all tested noise levels (fig. 4), with accuracy near .82 at noise $\sigma = 75$. Clearly, the accuracy of the algorithm drops with noise, however the performance remains strong when compared with other current methods.

3.1. Choosing k_{max}

Looking at the results in figs. 2-4, we can see that much of the time, the best performance is achieved by setting k_{max} equal to 1 or 10, and that larger values do not offer increased accuracy until we allow many more operations per pixel (on the order of 200). Using a larger value for k_{max} would seem to yield a better chance that the optimal answer will be contained in A at the time of early termination. However, as k_{max} becomes larger, A and B become correspondingly more expensive to maintain, costing more operations per pixel. Since the overal runtime for the algorithm is limited, this means that fewer operations are spent on extracting information from the image. This is illustrated in Fig. 5. In that figure we can see $\bar{c}$ and $\bar{f}$ drop while $\bar{q}$ rises as k_{max} increases. The values $\bar{c}$ and $\bar{f}$ not only indicate the amount of effort the algorithm expends in calculating the Walsh kernels and Euclidean distance, respectively, but they also directly denote the amount of information the algorithm has extracted from the image. This means that less of

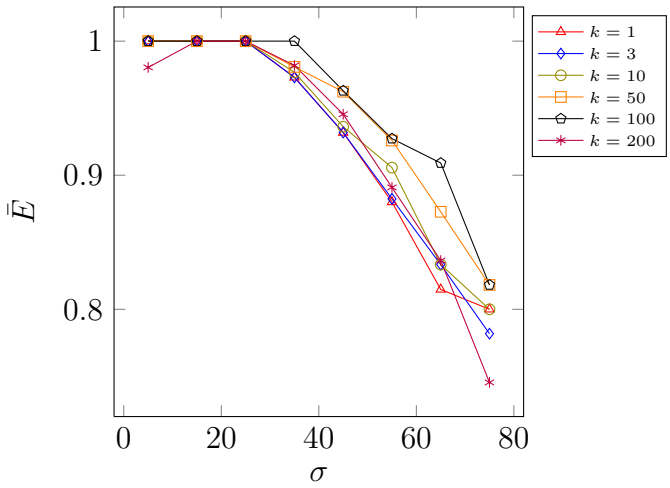

Figure 4: Results from allowing the algorithm to run to 200 operations per pixel. Note that in this plot the y axis (denoting accuracy) covers only the range .75 to 1.0.

the algorithm's time is spent finding the best answer, and more is spent on 'housekeeping' operations on A and B.

Additionally, as k_{max} increases, it becomes more and more difficult to separate $y_{k_{max}}$ from $y_{k_{max}+1}$. In fig. 6 we have ordered all y_i in an example image (lenna) by the value of E_i, and plotted the value $\delta(E_i) = E_{i+1} - E_i$. We can see that the difference between successive matches falls away quickly as i increases. Thus, when k_{max} is small, it is easier to tell $y_{k_{max}}$ from $y_{k_{max}+1}$, and the algorithm puts less effort into separating the two. When k_{max} is large, the values are very similar, and both must be evaluated to a higher value of d to differentiate them. This also results in more 'swap' operations between A and B, which increases the 'housekeeping' costs ($\bar{q}$) mentioned above.

3.2. An Example of the Algoritm's Operation

In fig. 8 we have run the algorithm on the classic image *lenna*, and early terminated the algorithm at varying times representing 15 to 30 operations per pixel with $k = 10$. Noise of $\sigma = 20$ has been added to the template. It can be seen that the algorithm initially finds a few poor matches for the image, and then rapidly focuses on a few neighbouring locations of the match. Effectively, by 18 ops/pixel, the best match has already been found, and the algorithm is trying to prove that the match is the best possible. The reader

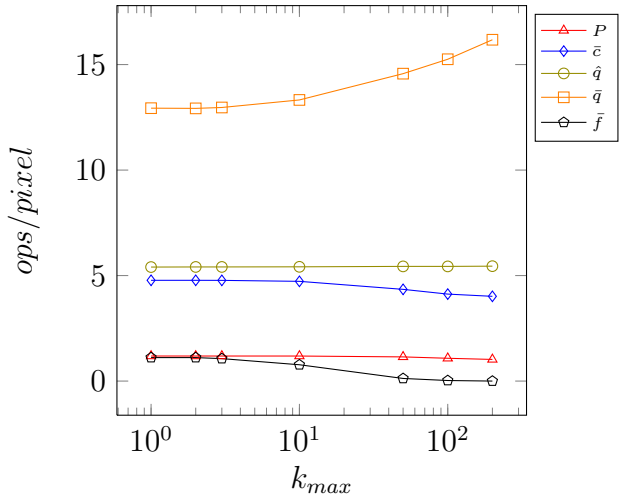

Figure 5: Components of the cost of the algorithm, described in Sec. 2.8, plotted against k varying from 3 to 1000. The algorithm was allowed to run the equivalent of 30 operations per pixel. It can be seen that as k increases, the cost of maintaining A and B rises, while the algorithm is forced to spend less time on computing the projections of Walsh kernels and exact distance.

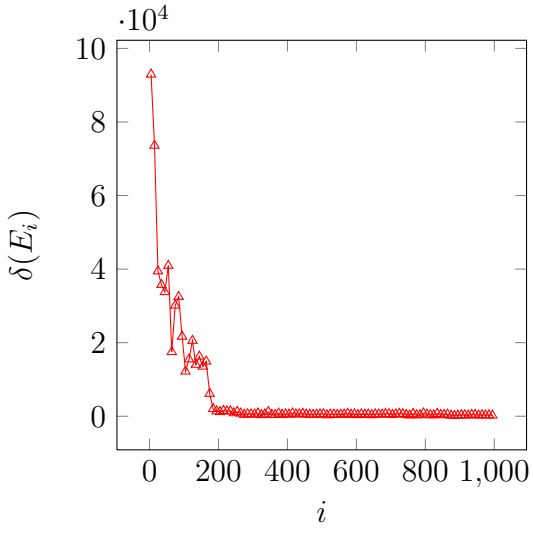

Figure 6: A plot of $\delta(E_i) = E_{i+1} - E_i$ against the value of i.

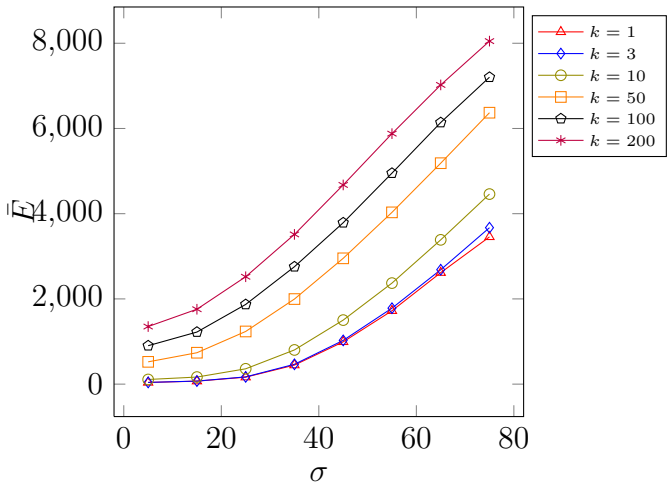

Figure 7: Results of letting the algorithm run to completion for noise levels ranging from $\sigma = 5$ to $\sigma = 75$.

is strongly encouraged to refer to a color copy for this image.

3.3. Cost of Finding Guaranteed Best Match

As noted in 2.6, if the algorthm is not forced into early termination mode, it will find an answer that is guaranteed to be the best match. In this experiment, we ran the algorithm to completion on the same dataset as the earlier experiments, with the exception that we did not interrupt the algorithm. Results are plotted showing the number of operations per candidate location on the y axis with the standard deviation of the added noise on the x axis. The results shown are competitive with state of the art exact template matching algorithms such as [37, 23, 13, 24].

4. Comparison to Current Methods

In a paper describing the current state of the art in approximate matching [18] a valid match is considered to be one where the sub-window and template share over 80 percent of their pixels. In their large scale test their algorithm typically required 92.9 operations per pixel, not counting initialization costs, to locate template of size 60x60. Our algorithm shows strong performance even at high noise levels with as few as 35 operations per pixel. In the case of a template with added Gaussian noise, their method took approximately 25 operations(double check this) to match a template with noise $\sigma = 25.5$

Figure 8: Lenna, with match locations shown by '+' signs. The template is shown at lower right. It has had Gaussian noise with $\sigma = 20$ added to it. The varying colors represent the number of operations per pixel given to the algorithm before termination. The matches shown in red indicate matches given very little time (≈ 15 operations per pixel). By 18 operations per pixel, the algorithm has already settled on a location at or within a pixel or two of the best match. This run-time includes pre-processing, and is exceptionally fast when compared to the current state of the art. The reader is strongly encouraged to refer to a color copy.

consisting of 1089 pixels. This is on par with our performance, as this is approximately one quarter the size of the templates we are searching for here.

Using a version of the highly optimized Fast Fourier Transform based Fast Convolution in the Open CV library [41] customized to count the required number of arithmetic operations to find a match, we found that the FFT algorithm requires on the order of 724 operations per location to find a match on a 64x64 template in a 512x512 image. This is regardless of noise level (and template size for that matter). Limiting our algorithm to 700 operations per pixel, we are able to get exact results up to a noise level of approximately $\sigma = 40$. At lower noise levels our algorithm reuires only a small fraction of this to achieve exact results.

5. Conclusion

In this paper we described a very efficient algorithm for template matching. The Arbitrary Time Algorithm very quickly locates the best match for a template in the search image, in as little as 25 operations per pixel. Experiments showed the algorithm typically finds a match which is either the best match, or very close to it, within user-defined time-limits. We showed that the algorithm converges quickly on a set of answers which are all very close to the best match. The experiments also showed that the algorithm is comparatively robust to image distortions caused by noise. The algorithm has a small memory footprint of the same approximate size as the search image, and can run in situations where computational resources are limited. Our experimental results indicate that the run-time of the algorithm compares favorably with both fixed-time approximate methods and data-dependent exact methods.

Future Improvements

It may be possible for the algorithm to automatically early-terminate when it has consistently found the same best match over many iterations, which could dramatically lower the time to run to completion, especially at higher noise levels. This algorithm could be adapted to image database search, as it does not rely on the fact that the potential matches all come from the same image, and the memory complexity is linear.

[1] W. Huang, Q. Sun, C. Lam, J. Wu, A robust approach to face and eyes detection from images with cluttered background, in: Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on, Vol. 1, 1998, pp. 110–113 vol.1.

[2] Z. Jin, Z. Lou, J. Yang, Q. Sun, Face detection using template matching and skin-color information, Neurocomput. 70 (4-6) (2007) 794–800.

[3] R. Brunelli, T. Poggio, Face recognition: features versus templates, Pattern Analysis and Machine Intelligence, IEEE Transactions on 15 (10) (1993) 1042–1052.

[4] K. Nallaperumal, R. Subban, K. Krishnaveni, L. Fred, R. Selvakumar, Human face detection in color images using skin color and template

matching models for multimedia on the web, in: Wireless and Optical Communications Networks, 2006 IFIP International Conference on, 2006, pp. 5 pp.–5.

[5] F. Smeraldi, O. Carmona, J. Bign, Saccadic search with gabor features applied to eye detection and real-time head tracking, Image and Vision Computing 18 (4) (2000) 323–329.

[6] A. A. Efros, W. T. Freeman, Image quilting for texture synthesis and transfer, in: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM, 2001, pp. 341–346.

[7] T. Luczak, W. Szpankowski, A suboptimal lossy data compression based on approximate pattern matching, Information Theory, IEEE Transactions on 43 (5) (1997) 1439–1451.

[8] M. B. de Carvalho, E. A. B. da Silva, W. A. Finamore, Multidimensional signal compression using multiscale recurrent patterns, Signal Processing 82 (11) (2002) 1559–1580.

[9] N. Rodrigues, E. da Silva, M. de Carvalho, S. de Faria, V. da Silva, On dictionary adaptation for recurrent pattern image coding, Image Processing, IEEE Transactions on 17 (9) (2008) 1640–1653.

[10] M. Atallah, Y. Genin, W. Szpankowski, Pattern matching image compression: algorithmic and empirical results, Pattern Analysis and Machine Intelligence, IEEE Transactions on 21 (7) (1999) 614–627.

[11] R. Li, B. Zeng, M. Liou, A new three-step search algorithm for block motion estimation, Circuits and Systems for Video Technology, IEEE Transactions on 4 (4) (1994) 438–442.

[12] M. Gharavi-Alkhansari, T. Huang, Fractal video coding by matching pursuit, in: Image Processing, 1996. Proceedings., International Conference on, Vol. 1, 1996, pp. 157–160 vol.1.

[13] S. Mattoccia, F. Tombari, L. D. Stefano, Fast Full-Search equivalent template matching by enhanced bounded correlation, Image Processing, IEEE Transactions on 17 (4) (2008) 528–538.

[14] A. Goshtasby, S. H. Gage, J. F. Bartholic, A Two-Stage cross correlation approach to template matching, Pattern Analysis and Machine Intelligence, IEEE Transactions on PAMI-6 (3) (1984) 374–378.

[15] A. Rosenfeld, G. Vanderburg, Coarse-Fine template matching, Systems, Man and Cybernetics, IEEE Transactions on 7 (2) (1977) 104–107.

[16] H. Masnadi-Shirazi, N. Vasconcelos, High detection-rate cascades for Real-Time object detection, in: Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on, 2007, pp. 1–6.

[17] W. Krattenthaler, K. Mayer, M. Zeiller, Point correlation: a reduced-cost template matching technique, in: Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference, Vol. 1, 1994, pp. 208–212 vol.1.

[18] O. Pele, M. Werman, Robust Real-Time pattern matching using bayesian sequential hypothesis testing, Pattern Analysis and Machine Intelligence, IEEE Transactions on 30 (8) (2008) 1427–1443.

[19] H. Schweitzer, J. Bell, F. Wu, Very fast template matching, in: Computer Vision ECCV 2002, Springer Berlin / Heidelberg, 2002, pp. 145–148.

[20] M. Gharavi-Alkhansari, A fast globally optimal algorithm for template matching using low-resolution pruning, Image Processing, IEEE Transactions on 10 (4) (2001) 526–533.

[21] M. Gharavi-Alkhansari, A fast full-search equivalent algorithm using energy compacting transforms, in: Image Processing, 2001. Proceedings. 2001 International Conference on, Vol. 2, 2001, pp. 713–716 vol.2.

[22] G. Ben-Artzi, H. Hel-Or, The Gray-Code filter kernels, Pattern Analysis and Machine Intelligence, IEEE Transactions on 29 (3) (2007) 382–393.

[23] F. Tombari, S. Mattoccia, L. D. Stefano, Full-Search-Equivalent pattern matching with incremental dissimilarity approximations, Pattern Analysis and Machine Intelligence, IEEE Transactions on 31 (1) (2009) 129–141.

[24] Y. Hel-Or, Real-time pattern matching using projection kernels, Pattern Analysis and Machine Intelligence, IEEE Transactions on 27 (9) (2005) 1430–1445.

[25] A. Rosenfeld, Picture processing by computer, ACM Comput. Surv. 1 (3) (1969) 147–176.

[26] H. M. Deitel, P. J. Deitel, D. R. Choffnes, Operating Systems, 3rd Edition, Prentice Hall, 2003.

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom, The worst-case execution-time problem\—overview of methods and survey of tools, ACM Trans. Embed. Comput. Syst. 7 (3) (2008) 1–53.

[28] P. Viola, M. J. Jones, Robust Real-Time face detection, International Journal of Computer Vision 57 (2) (2004) 137–154.

[29] L. Matthies, M. Maimone, A. Johnson, Y. CHENG, R. Willson, C. Villalpando, S. Goldberg, A. Huertas, A. Stein, A. Angelova, Computer vision on mars, International Journal of Computer Vision 75 (1) (2007) 67–92.

[30] M. Sen, I. Corretjer, F. Haim, S. Saha, S. Bhattacharyya, J. Schlessman, W. Wolf, Computer vision on FPGAs: design methodology and its application to gesture recognition, in: Computer Vision and Pattern Recognition - Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on, 2005, p. 133.

[31] J. Hannuksela, P. Sangi, J. Heikkilä, Vision-based motion estimation for interaction with mobile devices, Computer Vision and Image Understanding 108 (1-2) (2007) 188–195.

[32] Y. Ijiri, M. Sakuragi, S. Lao, Security management for mobile devices by face recognition, in: Mobile Data Management, 2006. MDM 2006. 7th International Conference on, 2006, p. 49.

[33] N. Cornelis, L. V. Gool, Fast scale invariant feature detection and matching on programmable graphics hardware, in: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, 2008, pp. 1–8.

[34] J. Fung, S. Mann, OpenVIDIA: parallel GPU computer vision, in: Proceedings of the 13th annual ACM international conference on Multimedia, ACM, Hilton, Singapore, 2005, pp. 849–852.

[35] J. Sochman, J. Matas, WaldBoost - learning for time constrained sequential detection, in: Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, Vol. 2, 2005, pp. 150–156 vol. 2.

[36] R. F. Anderson, H. Schweitzer, Fixed time template matching, in: Proceedings of the International Conference on Systems, Man, and Cybernetics, IEEE, 2009.

[37] H. Schweitzer, R. F. Anderson, R. A. Deng, A near optimal Acceptance-Rejection algorithm for exact Cross-Correlation search, in: Proceedings of the International Conference on Computer Vision, Kyoto, Japan, 2009, poster Session.

[38] R. A. Deng, R. F. Anderson, A Corner-Based method for computing the walsh transform at multiple locations in an image, Technical Report UTDCS-17-09, University of Texas at Dallas, Computer Science Dept. (Jul. 2009).

[39] P. Viola, M. J. Jones, Robust Real-Time object detection, Technical Report CRL 2001/01, Compaq Computer, Cambridge Research Laboratory (Feb. 2001).

[40] E. W. Weisstein, Walsh function – from wolfram MathWorld (Jun. 2009).

[41] Open computer vision library.