Presented at SPIE Conference on Document Recognition and Retrieval VIII
San Jose, CA, Jan 2001.

Document Image Decoding using Iterated Complete Path Search

Thomas P. Minka
Massachusetts Institute of Technology
tpminka@media.mit.edu

Dan S. Bloomberg
Xerox Palo Alto Research Center
bloomberg@ieee.org

Kris Popat
Xerox Palo Alto Research Center
popat@parc.xerox.com

February 12, 2002

Abstract

The computation time of Document Image Decoding can be significantly reduced by employing heuristics in the search for the best decoding of a text line. By using a cheap upper bound on template match scores, up to 99.9% of the potential template matches can be avoided. In the Iterated Complete Path method, template matches are performed only along the best path found by dynamic programming on each iteration. When the best path stabilizes, the decoding is optimal and no more template matches need be performed. Computation can be further reduced in this scheme by exploiting the incremental nature of the Viterbi iterations. Because only a few trellis edge weights have changed since the last iteration, most of the backpointers do not need to be updated. We describe how to quickly identify these backpointers, without forfeiting optimality of the path. Together these improvements provide a 30x speedup over previous implementations of Document Image Decoding.

Keywords: document image decoding, dynamic programming, Viterbi, heuristic search, optical character recognition, iterated complete path, MAP, template matching

1 INTRODUCTION

In the *Document Image Decoding* (DID) approach of Kopec and Chou [1], a model of the generation process is combined with a noise model to allow determination of a *maximum a posteriori* (MAP) message (decoding) given an observed image. The model of the generation process includes templates for the possible printed characters, a language model specifying a prior on the templates or sequences of templates, a setwidth parameter for each template describing its width, and a grammar that describes the syntax of sequences of printed characters. The simplest grammar is a finite state machine for a line of text, specified by a start node, an end node, and a printing node that loops back to itself as each character is printed.

Early work in DID by Kopec and Chou [1] and by Kam and Kopec [2] used bilevel templates, each level corresponding to the foreground and background of an ideal printed binary image, which were then corrupted by position-independent asymmetric bitflip noise to produce the observed image. This bilevel channel was characterized by two parameters: α_0, the probability that a background pixel is printed as background and α_1, the probability that a foreground pixel is printed as foreground. Equivalently, all observed pixels were modeled by one of two probabilities of being ON: a high probability for foreground pixels and a low probability for background pixels. The *log normalized likelihood* of observing an image Z when a template Q is printed, derived in Kopec and Chou [1], is given by

$$\mathcal{L}(Z \mid Q) \;=\; \gamma\, \|Q \wedge Z\| \;+\; \beta\, \|Q\| \tag{1}$$

where $\|X\|$ denotes the number of 1's in X, $\wedge$ is the bitwise **and** operator and

$$\begin{aligned} \gamma &= \log \frac{\alpha_0 \alpha_1}{(1-\alpha_0)(1-\alpha_1)} && (2) \\ \beta &= \log \frac{1-\alpha_1}{\alpha_0} && (3) \end{aligned}$$

It should be noted that (1) corresponds to Eq. (25) of Kopec and Chou [1], and not to the log likelihood, which one would get by taking the log of Eq. (19) in that paper.

Because pixels in the observed image near an on-off transition are not well described by such bilevel templates, the model was generalized to multilevel templates in Kopec [6], where the pixels in each template are assigned to one of several (disjoint) sub-templates, each sub-template being modeled by a binary asymmetric bit-flip channel (or, equivalently, by a given probabilty that its pixels are ON in the observed image.) For an N-level template, N-1 levels contribute to the likelihood with terms similar to (1), and level 0 remains the *background* level, treated as *don't-care* pixels in the likelihood calculation. Thus, with $Q^{(l)}$ as the sub-template for the l^{th} level, (1) generalizes to

$$\mathcal{L}(Z \mid Q) \; = \; \sum_{l=1}^{L-1} \; \mathcal{L}(Z \mid Q^{(l)}) \; = \; \sum_{l=1}^{L-1} \; \left[\gamma_l \, \| Q^{(l)} \wedge Z \| \; + \; \beta_l \, \| Q^{(l)} \| \right] \tag{4}$$

where γ_l and β_l are given by (2) and (3) for each level; that is, with α_1 replaced by α_l.

A foreground level l is *write-black* if $\alpha_l > 1 - \alpha_0$ and *write-white* if $\alpha_l < 1 - \alpha_0$. For *write-black,* $\gamma_l > 0$ and $\beta_l < 0$, and $\mathcal{L}(Z \mid Q^{(l)})$ increases as the number of black pixels that fall under $Q^{(l)}$ increases. For *write-white,* γ_l and β_l each change sign, and $\mathcal{L}(Z \mid Q^{(l)})$ is maximal when no black pixels fall under $Q^{(l)}$.

Because DID implicitly constructs a trellis and finds the best path through it for the given image, it does not make segmentation errors due to broken or connected characters, as conventional OCR systems do, as long as the imaging model is accurate. However, the cost is significant: the system must consider the possibility that any character can occur at any (feasible) location in the image. For a line decoder, in which a Viterbi procedure can be used and the computation is linear in the number of pixels, virtually all the computation is in the evaluation of the bitwise **and** operation in equation (4). With 200 5-level templates, and an 8 million pixel image (8.5 x 11 inches at 300 ppi), the required number of bitwise **and** operations between a sub-template and a region of the binary image is about 6 billion!

Kam and Kopec [2, 3] reduced this computation by an order of magnitude for text blocks, by doing the full line decoding only in the vicinity (i.e., within approximately ± 2 vertical pixels) of a text line. They defined a separable model as one where the 2D page decoding can be effected as a sequence of 1D (e.g., line) decoders. They then found the approximate vertical locations of the text lines by a procedure they named *Iterative Complete Path* (ICP). The method is as follows. Suppose the image is w by h pixels. For each of the h raster lines, find an easily computable upper bound heuristic score for the actual score from decoding a horizontal line of text at that location. These scores constitute a one dimensional array of size h. Then, with the constraint that two adjacent text lines must be vertically separated by some minimum distance, use dynamic programming to get an estimate of the text line locations. For each of these locations, *rescore* the text line (and a few adjacent lines) by doing a full horizontal line decoding. Using these rescored values in the vertical array, recompute the location of the lines. Iterate until all selected lines have been previously rescored, or equivalently, until the score for the page does not change between successive iterations. Because the heuristic is an upper bound, this is guaranteed to give the same result as if each raster line had been fully scored. If the bound is reasonably tight, only text lines near the actual locations will have been fully decoded.

In the sequel we describe the use of ICP *on each text line* to greatly reduce the computation of MAP decoding. As a pre-processing step for the decoding of text lines, it is necessary to identify each text line and its baseline within a block of text. One of the simplest and most efficient methods (e.g., see Bloomberg et al. [4]) is to deskew the page to within about one milli-radian and then use projection profiles, as in Chen et al., [5] to identify the horizontal baselines for each line of text. Baselines determined in this way are typically within one pixel of the correct vertical location at all points along the text line.

In Sec. 2 we describe an upper bound heuristic for the template score at each location, that can be used, with ICP, to reduce the number of exact template scores that must be computed when decoding a text line. For typical text images, this reduces the number of exact template scores by about *three orders of magnitude*. The burden of computation is then shifted from exact template matching to the calculation of the heuristic and the Viterbi iterations. It has been observed that when ICP is used on text lines, the influence of changes in the current best path from iteration to iteration tends to be spatially localized. In Sec. 3, we exploit the locality of change by an *incremental Viterbi* procedure.

2 ICP WITH LINE DECODER

Kopec and Chou [1] showed that with a simple Markov source, corresponding to a unigram language model where the priors on the templates are independent of history, a MAP path for the line decoder is found by solving the recursive equation for the MAP function $\mathcal{F}$ at successive pixel locations x as a function of the likelihoods at previous pixel positions. With the prior transition probability a_t for a transition t that includes a matching score for a bilevel template Q_t accompanied by a translation Δ_t,

$$\mathcal{F}(x) \;=\; \max_{t \mid R_t = x} \{\mathcal{F}(x-\Delta_t) \;+\; \log a_t \;+\; \mathcal{L}(Z \mid Q_t[x-\Delta_t])\} \tag{5}$$

where the maximum is taken over all transitions originating from the template origin at $x - \Delta_t$ and whose right side R_t is located at x.

The only change introduced by multilevel templates is in the computation of the template match scores $\mathcal{L}(Z \mid Q_t[x])$, as in (4). It is seen that multilevel template matching typically requires about $L-1$ times as much computation as bilevel template matching.

2.1 Heuristic for Bilevel Templates

To guarantee that a MAP path will be found, a strict upper-bound for the matching score must be used. For bilevel templates, an upper bound for the two-dimensional bitwise **and** between template Q_t and image Z is a one-dimensional column-wise sum, where each term is *the minimum number of ON pixels in the template column and the corresponding column in that part of the image that is covered by the template*. This can be seen as a distribution of ON pixels in each image column in such a way as to cover the maximum number of ON pixels in the template column. This heuristic is much faster to calculate than the actual score for two reasons. First, the number of terms in the sum is reduced by a factor equal to the height of the template (typically about 40 pixels). Second, the actual two-dimensional sum must be calculated for several vertical positions of the template above and below the baseline, because the best position of the baseline is not exactly known. We typically find it at the five vertical positions within 2 pixels of the nominal baseline, and take the largest likelihood score. However, the heuristic gives an upper bound for *any* vertical alignment. With these factors, the heuristic for bilevel templates is found to be nearly two orders of magnitude faster to compute.

2.2 Heuristic for Multilevel Templates

The upper-bound heuristic for a multilevel template is again found as a one-dimensional column-wise sum, where here the ON pixels in each image column are distributed over ON pixels in the corresponding column in each sub-template in such a way as to maximize the score. For the multilevel template, each of the $L - 1$ levels has a γ_l weighting parameter that is multiplied by the number of ON pixels aligned between template and image. The score will be maximized when the image pixels are distributed on sub-templates starting with the level l with largest γ_l, and proceeding to levels with smaller γ_l. If there is no write-white level, the ON pixels in the image column are used in this way until either they or the ON sub-template pixels are exhausted. (This is a generaliztion of the minimum count for bilevel templates). The *write-white* level has $\gamma_l < 0$, so its score is maximized when all ON pixels in the sub-template are covered by OFF pixels in the image. If there are not enough OFF pixels in the image to cover the ON pixels in the *write-white* sub-template, we are forced to put ON pixels on the sub-template, which decreases the score.

The scores for decoding a text line can be represented by a w x M array, where w is the width of the line in pixels and M is the number of templates. For efficiency, the heuristic score for a column in a multilevel template should be pre-evaluated as a function of the number of ON pixels in the image column. Then the score for the template column can be found from the number of ON pixels in the image column by table look-up. Aside from the generation of these tables, the computation of the heuristic score array is comparable for bilevel and multilevel templates. However, because a multilevel template is represented as a set of $L - 1$ bilevel templates, the efficient computation of the the actual score for a multilevel template is approximately $L - 1$ times that for a bilevel template. Thus, with multilevel templates, ICP gains an additional factor of about $L - 1$ in efficiency over the non-iterative method for computation of the score array.

2.3 ICP Algorithm

Given an upper bound heuristic, the ICP algorithm for line decoding is:

```
initialize score array with upper-bound heuristics
$i \leftarrow 0$, score($0$) $\leftarrow 0$
do
    $i \leftarrow i + 1$
    find best path using Viterbi on current score array
    score($i$) $\leftarrow$ score of that best path
    rescore array along best path
until: score($i$) = score($i - 1$)
```

Algorithm for ICP with upper bound heuristic

In the rescoring operation, the actual template match score is found at typically five vertical positions around the nominal baseline, and the largest score is inserted in the score array. Also, to reduce the number of iterations, the template is also matched in the vicinity of the node, typically up to two pixels on each side, and the rescored values are inserted in the score array. Otherwise, a new path is likely to be found with the same template in a neighboring x location, because it is using the (usually larger) heuristic value in evaluating paths going through the neighboring nodes. It is also important to mark which nodes in the array have been rescored, so that the same node is not rescored more than once.

3 INCREMENTAL VITERBI

With ICP, the computational burden shifts from the score array toward the Viterbi iterations for the best path through the trellis. The Viterbi iteration is fairly expensive because with M templates it requires comparison of M scores at each pixel location x in the forward direction. As iterations proceed, the path through some locations, such as the space between words, often remains invariant. The regions where path changes occur, necessitating rescoring of nodes, narrow as convergence is approached. For some images, it is observed that one or a very few locations require many iterations before the best path is found. To exploit this localization phenomenon when updating the current best path, the Viterbi algorithm must be able to identify conditions under which it is guaranteed that a portion of the best path will not be different from the previous iteration, without actually performing the search for that portion of the path.

Each template has a setwidth, Δ_t, that appears in the recursion relation (5), and that expresses how far back it "looks" on the forward Viterbi pass. Suppose that we save the partial scores $\mathcal{F}$ and the final transition at each location x from the most recent Viterbi forward pass, and that a node was rescored at location x_r as a result of that pass. Denote the maximum setwidth over all the templates by Δ_{max}. In the forward Viterbi, the condition that this rescoring will have no further effect downstream is that over an interval of Δ_{max} pixel locations, the difference in partial scores between the previous and current iterations, $\Delta_{\mathcal{F}}(x)$, is a constant. If such an interval is found, it can be safely assumed that the score difference will remain constant until another rescored node is reached. This is because at the right end of the interval, back-pointers for all templates reach back into the interval, so we know that forward Viterbi will give the same set of transitions from that point onwards. Consequently, at this point in the forward Viterbi, we enter *skip-mode*, where the forward Viterbi is implemented using the previous iteration by simply propagating the score difference $\Delta_{\mathcal{F}}$ and saving the previous transition. We must leave *skip-mode* and re-enter full Viterbi whenever a rescored node is encountered. For all iterations but the first, we begin the line in *skip-mode*, leaving only when the first rescored node is reached. The control logic for incremental Viterbi is thus:

```
set skip-mode ON
for each $x$ from 0 to $w$
{
    if skip-mode is ON,
        if we hit a rescored node,
            set skip-mode OFF
            Icount $\leftarrow 0$

    otherwise if skip-mode is OFF
        if $\Delta_{\mathcal{F}}(x) = \Delta_{\mathcal{F}}(x-1)$,
            Icount $\leftarrow$ Icount $+1$
            if Icount $> \Delta_{max}$
                set skip-mode ON
        else
            Icount $\leftarrow 0$ (reset)
}
```

Algorithm for Incremental Viterbi

At first sight it might be expected that no *skip-mode* regions will be entered after a rescored node is reached. For it is likely that the rescored node will remain on the partial best path, and that some downstream nodes will point back to it. In turn, those downstream nodes are likely to be referenced by nodes further downstream. With some nodes being on the path through the rescored node and others not, the score differences $\Delta_{\mathcal{F}}(x)$ will vary from node to node as x is incremented. What is to stop this? The answer is that the full trellis typically has cut sets where all arcs converge. These usually occur in white space between words or characters, as mentioned above, and act to strictly limit the propagation of changes.

In the early iterations, few regions in the line are in *skip-mode*, but as convergence is approached, most of the line goes into *skip-mode*. Use of incremental Viterbi typically reduces the total Viterbi computation by a factor of between 2 and 3.

	Exact scores	*Bilevel matches*	*Time (sec)*
Matching	644,840	9,672,600	23.85
Viterbi			0.04
Total			23.89

Table 1: *Computation for decoding a textline of width 1960 pixels with 329 4-level templates, using the actual scores and one pass of Viterbi.*

4 DISCUSSION

In standard DID, without ICP, a matrix of exact matching scores is computed for all templates at all positions. For each pixel position across the text line, matching scores are performed with 32-bit aligned operations between each of the (32-bit aligned) templates and a buffer into which a fragment of the image starting at that pixel position has been placed. After each pair of 32-bit words is ANDed, table lookup is used to determine the number of ON pixels. The results are weighted for each sub-template by the appropriate γ factor for that level.

In Table 1, we show the compuation required for decoding a typical line of text using DID without ICP. The text line has 85 characters, including white space, and a width of 1960 pixels. The model has 329 4-level templates, requiring 3 bilevel matches for each template match. The node scores are found for the best vertical alignment among 5 vertical positions of the template near the baseline, and hence require 15 bilevel matches for each node score. All times are on a program built with the GNU C compiler, running on a 800 MHz Pentium III. Note in particular the disparity between the amount of time required to compute the matching scores and the time to find the best path.

Table 2 shows the compuation required for decoding the same line of text using ICP, both with and without incremental Viterbi. The heuristic ICP array is found in about 2$\%$ of the time required to generate the array of exact matching scores. This savings is slightly offset by the larger time required to run multiple iterations of Viterbi, and the result is a much better balance between time spent on template matching and on dynamic programming. The number of iterations and rescored nodes does not depend on whether incremental Viterbi is used. Approximately 60 nodes can be rescored in 1 msec. (Remember that each rescored node requires 5 separate matches at different vertical positions around the baseline.)

The number of rescored nodes includes two nodes with the same template that are spatially adjacent

	Elements	*Iterations*	*Fraction in slow mode*	*Time (sec)*
Heuristic Array	644,840			0.54
Rescored Nodes	1144			0.02
Full Viterbi		10	1.00	0.40
Incremental Viterbi		10	0.57	0.28
Total ICP, with full Viterbi				0.96
Total ICP, with incremental Viterbi				0.84

Table 2: *Computation for decoding a textline of width 1960 pixels with 329 4-level templates, using ICP with both full and incremental Viterbi. Two adjacent nodes are always rescored for every best path found by Viterbi. Without adjacent node rescoring, full Viterbi takes about 0.64 sec.*

to the identified path nodes. Rescoring these adjacent nodes requires little extra computation, and acts to significantly reduce the number of Viterbi iterations. This is shown in more detail in Table 3, which gives the effect of the number of adjacent rescored nodes on the computation. The use of two adjacent nodes gives about a 30% reduction in Viterbi time, due mainly to a reduction in the number of required iterations. With more than two adjacent nodes, the number of iterations is slightly reduced, at a cost of some extra rescored nodes, and the overall effect on computation time is negligible.

5 CONCLUDING REMARKS

We can summarize the approximate speedup factors over standard DID without ICP for a 4-level template model as follows:

- ICP with full Viterbi: *20x*
- ICP with incremental Viterbi: *24x*
- ICP with incremental Viterbi and adjacent node rescoring: *28x*

For 4-level templates, the speed up in computation of the heuristic array, compared to the array of exact scores, is typically about 50x. Of this, about 5x is due to the evaluation of exact matches at 5 different vertical positions, and the remaining 10x is the relative speed of the 1D versus 2D convolution. Part

	Adjacent nodes evaluated			
	0	2	4	6
Rescored nodes	644	1144	1646	2146
Viterbi iterations	17	10	9	8
Fraction in skip mode	0.15	0.19	0.19	0.20
Fraction in slow mode	0.55	0.57	0.54	0.57
Fraction in fast mode	0.30	0.24	0.29	0.23
Rescore node time (sec)	0.01	0.02	0.03	0.04
Viterbi time (sec)	0.42	0.28	0.27	0.26
Rescore + Viterbi time	0.43	0.30	0.30	0.30

Table 3: *Dependence of Viterbi computation on the number of adjacent nodes that are rescored at each Viterbi iteration.*

of the gain of the 1D is the 3x factor whereby the 2D exact matches must be made over each of the three sub-templates, whereas the 1D column scores are found by table lookup. This indicates that for a 2-level template, the intrinsic speed ratio of 1D versus 2D convolution is only about 3x. The 1D convolution is performed as integer operations on column sums, whereas the 2D convolution is done as word-parallel bit operations followed by table lookup.

Using ICP with incremental Viterbi and adjacent node rescoring, about two-thirds of the computation is in the heuristic scoring, and about one-third in the dynamic programming. For further improvements, a reduction of the heuristic scoring computation is possible with subsampling, and a discussion of this approach is forthcoming.

At present, the time to decode a 2000 pixel wide column of text using 400 4-level templates is about 1 second. This is perhaps 5x slower than commercial OCR packages, which decode such a text line in about 200 msec. However, noting that the decoding time with DID is approximately proportional to the number of templates that are used, the DID ICP line decoding time with 4-level templates can be estimated at 2.5 msec/template. Consequently, it should take about 250 msec to decode a line using ICP with 100 4-level templates.

ACKNOWLEDGEMENTS

This work is a direct outgrowth of the pioneering work by Gary Kopec and Phil Chou, and we wish to dedicate it to Gary, who is no longer with us to share in the fruits of his creativity. The authors wish, in particular, to acknowledge the generosity and guidance that Gary Kopec provided through the years of our association with him.

References

[1] G. Kopec and P. Chou, “Document image decoding using Markov source models”, *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 16, no. 6, June, 1994, pp. 602–617.

[2] A. Kam and G. Kopec, “Separable source models for document image decoding”, in *Document Recognition II*, L. Vincent and H. Baird, editors, Proc. SPIE vol. 2422, pp. 84–97, 1995.

[3] A. Kam and G. Kopec, “Document image decoding by heuristic search,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 18, no. 9, Sept. 1996, pp. 945-950.

[4] D. S. Bloomberg, G. E. Kopec and L. Dasari, “Measuring document image skew and orientation”, *Document Recognition II*, L. Vincent and H. Baird, editors, Proc. SPIE vol. 2422, pp. 302–316, 1995.

[5] F. Chen, D. Bloomberg and L. Wilcox, “Spotting phrases in lines of imaged text”, *Document Recognition II*, L. Vincent and H. Baird, editors, Proc. SPIE vol. 2422, pp. 256–269, 1995.

[6] G. Kopec, “Multilevel character templates for document image decoding” in *Document Recognition IV*, L. Vincent and J. Hull, editors, Proc. SPIE vol. 3027, pp. 168-179, 1997.