Direct Binary Search with Adaptive Search and Swap

Sagar Bhatt, John Harlim, Joel Lepak, Robert Ronkese, John Sabino
Mentor: Chai Wah Wu

August 10, 2005

Grayscale halftoning is the technique of approximating a grayscale image by a black-and-white image. Black-and-white printers use halftoning to replicate grayscale images on paper. The printers have only the black of ink and the white of paper as the colors at their disposal. Although the printers may vary the size of dots they drop, they cannot vary the intensity of the droplets. All droplets are black. The printers therefore actually produce black-and-white *halftones* instead of grayscale prints. The task is to make the black-and-white image indistinguishable from the grayscale one. We consider a halftoned image to be a good approximation to the original grayscale in the obvious sense: perceptually, the images look like one another. How to quantify this metric, however, is less than obvious. A complete model for the human visual system is an open research problem and must take many things into account: for example, how the eye blurs together nearby pixels, the sensitivity of the eye to changes in texture and luminescence, the attenuated response of the eye to changes along diagonal directions, and how optical illusions fool the brain. Understanding that the approximation problem must take into account visual perception, we define the grayscale halftoning problem:

The Halftoning Problem *Given a grayscale image I, find a black and white image O that minimizes $\|G(O) - G(I)\|_2$, where $G(\cdot)$ is the image perceived when viewing $\cdot$.*

For halftoning, the main aspect we want to model is the blurring phenomenon. The eye perceives patches of black-and-white pixels as some kind of average gray when viewed from sufficiently far away. Several models account for blurring, and a particularly effective yet simple model is to treat vision as a Gaussian filter. In this case, G is a two-dimensional convolution operator with a Gaussian kernel.

A particularly restrictive constraint in designing a practical halftoning algorithm is that the method must perform extremely fast. For high-volume printing, the standard is that the algorithm must produce an acceptable halftone in only order kN work, where N is the number of pixels in the original image, and k is a small (say, $k \lesssim 10$) constant. Several algorithms produce fast halftones, but printer manufacturers desire higher-quality halftones. We will review some of the fastest methods as well as a slower method called Direct Binary Search (DBS). While DBS produces higher-quality halftones, its cost is too high. The purpose of this paper is to present improvements to DBS that preserve most of the quality of its halftones but with a number of operations currently deemed acceptable.

1 Halftoning methods

Halftoning algorithms can be categorized into three categories based on their computational complexity [1]. Among the simplest methods are ones that treat each pixel individually, not taking into account neighbors. These methods are very fast, but the quality of the halftones is usually unsatisfactory. Region-based methods take into account the effect neighbors have on determining local tonal levels. These methods produce better halftones, and are generally fast enough for production-quality printers. Finally, the methods which produce the best halftones try to minimize the error over the entire image, not just region by region. However, these methods are far too slow for images of even small size. (The integer programming formulation of the problem must search of a feasible set with 2^N points.)

In all of the following algorithm descriptions, I is the input image and O is the output halftone image. Additionally, we represent gray colors as numbers in $[0, 1]$, varying smoothly from white at 0 to black at 1. We think of images as discrete, finite two-dimensional arrays of numbers. We refer to entries in the image arrays as *pixels*.

1.1 Dithering

Dithering is a simple approach for generating a halftone. Whereas rounding each grayscale value to white or black based upon whichever it is nearer creates unsatisfactory halftones (in other words, ignoring blurring by taking G to be the identity), dithering rounds each pixel to 0 or 1 based on varying criteria. Figure 1.1 describes the dithering process. Notice that rounding corresponds to taking the entries in A to be uniformly $1/2$.

We have several ways of picking the screen A. A first approach is to pick the entries of A randomly. While markedly better than rounding, halftones produced this way tend to exhibit unpleasant graininess. A way that works better in practice is to pick A so that it works well when applied to a set of test images, such as several tonal levels of constant gray [2]. A good screen is usually found by applying a more costly halftoning algorithm to the test images, finding good halftones, and designing a screen A that gives back those halftones when applied to the test images. The hope is that when the screen is applied to nonuniform images, at least the smoothly-varying regions will look good. Although less so than random screens, the halftones produced with these screens often still look somewhat grainy.

Input: Image I, screen A of size $R \times C$
Output: Halftone image O
1. For each pixel (i, j) in I, set $O(i, j) = \begin{cases} 1, & \text{if } I(i, j) \geq A(i \bmod R, j \bmod C), \\ 0, & \text{otherwise.} \end{cases}$

Figure 1: Screening algorithm

1.2 Error diffusion

Slightly more computationally expensive than dithering are algorithms that consider the value of nearby pixels when computing the value of a pixel in the halftone. Error diffusion is the most widely used example of this approach. As indicated in Figure 1.2, the error introduced by rounding each pixel is spread to a neighborhood of the pixel, using the weights shown in a matrix Q. The matrix Q in Figure 1.2 was proposed by Floyd and Steinberg [4], but others have proposed different matrices for spreading the error [5, 6, 7].

1.3 Direct binary search

The most computationally intensive class of algorithms are iterative algorithms: those which require several passes through the image before converging to the final halftone image. Direct binary search (DBS) is an algorithm of this type that is the primary focus of this paper.

DBS proceeds by generating an initial halftone image (possibly using screening or another fast method), and then performs a local search on the halftone space by swapping (swapping the colors of neighboring pixels) and toggling (changing an individual pixel to the opposite color). An outline of a basic version of the algorithm is shown in figure 3.

The error is computed using a model of the human visual system that essentially blurs nearby pixels together. Section 2 describes in detail how to update the error with reduced computational cost.

Note that throughout the paper, "processing" a pixel should be interpreted to mean performing step 3(a) of figure 3 for that pixel, i.e. checking for a good swap or toggle and performing it if one is found. Also,

1. For each pixel (i, j) in I:
   (a) Set $O = \text{round}(I)$.
   (b) Set $e = I(i, j) - O(i, j)$
   (c) Update the neighborhood of $I(i, j)$:
   $I(i-1:i+1, j-1:j+1) = I(i-1:i+1, j-1:j+1) + eQ$, where

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7/16 \\ 3/16 & 5/16 & 1/16 \end{bmatrix}$$

Figure 2: Error diffusion algorithm

1. Generate an initial halftone O
2. Compute the error $\mathcal{E}$ in approximating I by O
3. Iterate the following until an iteration produces no change:
   (a) For each pixel (i, j):
      i. Check which of the following changes to O decreases $\mathcal{E}$ the most:
         - swapping pixel (i, j) with one of its 8 nearest neighbors
         - toggling pixel (i, j) to the opposite color
      ii. if any of the possible actions decreases $\mathcal{E}$, perform the best one
      iii. update $\mathcal{E}$

Figure 3: Basic DBS algorithm

we will say that a trial has been performed each time that the potential error change of a swap has been evaluated. In summary, the relevant statistics we will compare for various DBS schemes are

- perceived error in the halftone
- number of swaps
- number of toggles
- number of trials.

2 DBS: prior work

2.1 Computing the error

The error function used in the DBS algorithm is based on a model of the human visual system (HVS). The HVS is modeled as a low-pass filter [3]; this filter is applied to the difference between the input image and the halftone, $O - I$.

The error of approximation $\mathcal{E}$ is defined to be

$$\mathcal{E} = ||\tilde{e}||_2 ,$$

where $\tilde{e}$ is the perceived error. The perceived error is computed by applying the HVS filter $\tilde{p}$ to the error $O - I$:

$$\tilde{e}(x, y) = \tilde{p} \star\star (O - I)(x, y),$$

where $\star\star$ denotes 2-dimensional convolution.

For a more accurate model, a model of the printer spot profile (the actual appearance of the ink dots on the printed page) should also be taken into account. In this paper we will follow [3] and assume that the spot profile is small enough (compared with the blurring effect of the HVS) as to not affect the model.

The HVS filter $\tilde{p}$ is stored as a $P \times P$ matrix; in our case studies we use $P = 11$, with

$$\tilde{p}(i, j) = \exp\left(-(i^2 + j^2)/5\right).$$

2.2 Updating the error

The DBS algorithm needs to evaluate many trial toggles and swaps; recomputing the full matrix $\tilde{e}$ at each trial is infeasible. As described in [1], the fact that swapping or toggling only introduces a small, localized change to $\tilde{e}$ allows us to keep lookup tables that allow much faster computation of $\Delta\mathcal{E}^2$ for a trial change.

Using the scheme described in [1], $\Delta\mathcal{E}^2$ can be evaluated using four table lookups for a trial swap and two table lookups for a trial toggle. If a swap or toggle is performed, approximately $(2P - 1)^2$ table entries must be updated when using a $P \times P$ HVS model $\tilde{p}$. A swap is twice as expensive to perform as a toggle.

Due to the number of table entries that must be updated, a swap or toggle is much more expensive to perform than a trial, especially for large values of P. For this reason, in our work described below we concentrate primarily on reducing the number of changes (swaps or toggles) a DBS algorithm needs to make to the initial halftone. Additionally, several of our methods also reduce the number of trials necessary as well.

2.3 Further computation reductions

Techniques for further reducing computation in DBS are outlined in [1].

Rather than testing all eight nearest neighbors for possible swaps, only the neighbors in the anticausal neighborhood should be tested, as the others will have already been tested against the center pixel while processing previous pixels.

The DBS implementation should also attempt to avoid swaps and toggles that do not decrease the error significantly. One method is to process the pixels in non-overlapping cells and allow only one change per cell, rather than one change in every possible neighborhood. This will eliminate many changes that may be insignificant. An optimal neighborhood size has been found empirically to be 4×5 [1]. Our work gives several other ways of reducing these unproductive changes.

3 DBS: new techniques

After implementing the standard DBS algorithm, we experimented with several variations on the method. One of the first was changing the order in which the pixels were processed. Interestingly, sweeping through the image in random order seemed to improve the convergence rate when compared to the normal 'scanline' path. Further tests showed that it was the spacing of the search pixels that accounted for the difference. That is, a random search order meant pixels processed consecutively were relatively spread out, thereby ensuring that excessive changes were not made to small areas. Thus one way to improve the algorithm was to go through the pixels with a large enough step size, or at the least, ensure a sparsely distributed search set.

Another approach was to focus on pixels corresponding to high perceived errors. The idea was to sort the pixels according to their perceived errors to get the order of traversal. Though computationally expensive, the error reduction was rapid. In fact, after sorting, just one iteration of DBS was needed to reach a satisfactory overall error level. Instead of altering pixels in an arbitrary order, high-priority areas are fixed

first. In this manner, less changes are needed on subsequent passes. Clearly, sorting (even once) drastically increases the number of operations. To reduce this cost, we partitioned the image into smaller blocks and only sorted each block.

Instead of giving higher priority to pixels at which the error is higher, one might emphasize the pixels where the most improvement can be made. Note that these are not equivalent. To implement the latter method, we keep a running average of the change in error from swaps in order to decide which action to perform. Thus an average change in error is stored from all swaps made so far in the current iteration. If swapping the current pixel can improve the error by more than a given fraction of the average improvement, the swap is accepted. If not, a toggle is considered and accepted if it will reduce the error. Otherwise, no action is performed. This saves computation since sorting is not needed and better (and therefore fewer) swaps are accepted so that less updating is performed.

Besides the processing order and criteria for making a change, we considered the effect of reducing the size of the search set as well. We realized that it was not necessary to process the full image at every iteration in order to obtain a better image. Taking this into account, we let the search set for one iteration only include the fraction of the image where changes were made in the prior iteration. Thus if a toggle is made at pixel p, the pixel is added to the search set for the next iteration. Alternatively, if a swap is made with pixel q, both p and q are added to the search set. Another strategy is to add to the search set entire neighborhoods of the pixels where changes were made.

For the DBS algorithm, one can optimize the operations performed during each iteration to gain some speed. But one can achieve even greater enhancements by decreasing the number of pixels processed and altered while still maintaining acceptable levels of improvement in the image. The trick is wisely selecting search sets and deciding when to accept changes carefully.

4 Implementation details and case studies

This section describes the use of the techniques of the previous section, both alone and in combination with each other.

4.1 Ending criteria

Several different criteria for terminating DBS can be used depending on the circumstances. If runtime is not a large concern, the algorithm can iterate until no changes are made in an iteration; this would be appropriate for situations such as generating screens. To limit the number of iterations that do not reduce the error by a significant amount, the algorithm can terminate when the relative error decrease in an iteration falls below a tolerance.

In each example shown below, the ending criteria is as follows: let $\mathcal{E}_i$ be the error after iteration i; terminate the algorithm at the end of iteration n if $(\mathcal{E}_n - \mathcal{E}_{n-1})/\mathcal{E}_{n-1} < 0.01$.

4.2 Sorting

For simplicity of notation we assume we are sorting entries based on $|\tilde{e}|$. However, in actual implementation, the sorting is based on table entries which are different from, but related to, $|\tilde{e}|$. Results have been observed to be essentially the same when using the related quantity.

The major issue with sorting the pixel locations based on $|\tilde{e}|$ is the time spent sorting. The computational cost can be reduced using two techniques, both of which have been observed to give essentially the same results as regular sorting.

4.2.1 Local sort

Due to the local nature of error propagation, it is not essential to sort the entire image. The image can be broken into constant size blocks and each block sorted separately without significant degradation of performance. An outline of this revised DBS algorithm is shown in figure 4.

1. Split the image into $b \times b$ size blocks
2. Sort each block based on $|\tilde{e}(i,j)|$
3. For $r = 1 \ldots b^2$:
   (a) Process the rank r pixel of each block according to DBS rules

Figure 4: Local sort DBS algorithm

1. Generate an initial search set S
2. Repeat the following until the ending criteria is met:
   (a) For each pixel (i,j) in S:
      i. Remove (i,j) from S
      ii. Process pixel (i,j)
      iii. if pixel (i,j) is changed, add its neighborhood to S

Figure 5: Search set refinement DBS algorithm

Tests on sample images using a block size of only 4×4 have performed nearly as well as globally sorting the pixel locations.

4.2.2 Approximate sorting

Another method that can be used is to only approximately sort the pixel locations. This can be accomplished simply by hashing the locations into a larger array based on their error value relative to the maximum error. This technique should be combined with local sort for larger images to save memory.

4.3 Search set refinement

Rather than sorting to generate a priori a set of pixels whose processing should provide a good error decrease, we can use information from previous iterations of the algorithm to generate the set of pixels to process. Search set refinement is a technique that uses the following heuristic: if a pixel has been changed, nearby pixels may also need to be changed. This variant of DBS proceeds by generating an initial sparse set of trial pixels, and then expanding the search set to the neighborhoods of any modified pixels from the initial search set, repeating until a satisfactory halftone is reached. The technique is outlined in figure 5. The neighborhood size can be changed for varying situations. When search set refinement is used in the examples below, the neighborhood radius is zero (the neighborhood is just the single center pixel) or one (the neighborhood is a 3×3 grid).

4.3.1 Threshold refinement

A technique that can reduce the number of changes (although not necessarily the number of tests) is to only allow a swap that decreases the error by a significant amount.

We use a threshold that is updated as the image is processed. Let $\overline{\Delta\mathcal{E}^2}$ denote the average change in error from all swaps made in the current iteration. Note that $\overline{\Delta\mathcal{E}^2}$ is negative. In this case we use a fraction of $\overline{\Delta\mathcal{E}^2}$ as the threshold; when a pixel is processed, if it is determined that a swap can reduce the error

Threshold weight: β. A typical value is $\beta = 1/2$.

1. Set $\overline{\Delta\mathcal{E}^2} = 0$
2. For each pixel to process:
   (a) If the best trial change is a toggle, perform the change and continue to the next pixel. Otherwise:
   (b) If the best trial swap gives $\Delta\mathcal{E}^2 < \beta\overline{\Delta\mathcal{E}^2}$, do the following:
      i. Perform the change
      ii. Update $\overline{\Delta\mathcal{E}^2}$

Figure 6: Threshold refinement DBS algorithm

by more than the threshold, that action is performed. This is outlined in figure 6. Toggles are performed without regard to the threshold, primarily because there are much fewer toggles than swaps, and a toggle may be more necessary to bring the average gray level of the halftone in a given region into agreement with the original image.

4.4 Case studies

In figures 7, 8, and 9 we compare the performance of four different implementations of DBS variants. We used 20 test images with sizes ranging from 287296 pixels to 1893376 pixels. The four implementations are:

- Unoptimized DBS: For comparison, this implementation uses standard 3×3 neighborhood size, and is implemented just as described in figure 3.
- Search set refinement (SSR): This implements the algorithm in figure 5 with neighborhood size one and an initial set S that is a uniform grid of 1/16 of the pixels.
- Local sort, dense initial set: This implementation uses a 4×4 block size, and runs just as in figure 4.
- Local sort, sparse initial set: This implementation begins by sorting only the elements in a sparse grid consisting of 1/16 of the pixels. It then processes the elements in order and uses search set refinement (figure 5) to expand the set of pixels to process.
- Regular spacing: This used a 16×16 grid, choosing one point from each block and iterating through the blocks until each pixel has been processed.

All but the DBS reference implementation use threshold refinement with $\beta = 0.5$. Note that because swaps are twice as expensive as toggles, the plot in figure 8 shows a relevant statistic. Each technique clearly improves on unoptimized DBS.

5 Conclusions and future work

We have presented several methods to reduce both the number of pixel tests and number of changes to the halftone by a DBS algorithm. These methods improve runtime performance while still generating a halftone with low error.

Future work will include more extensive testing of the methods, including optimization of algorithm parameters.

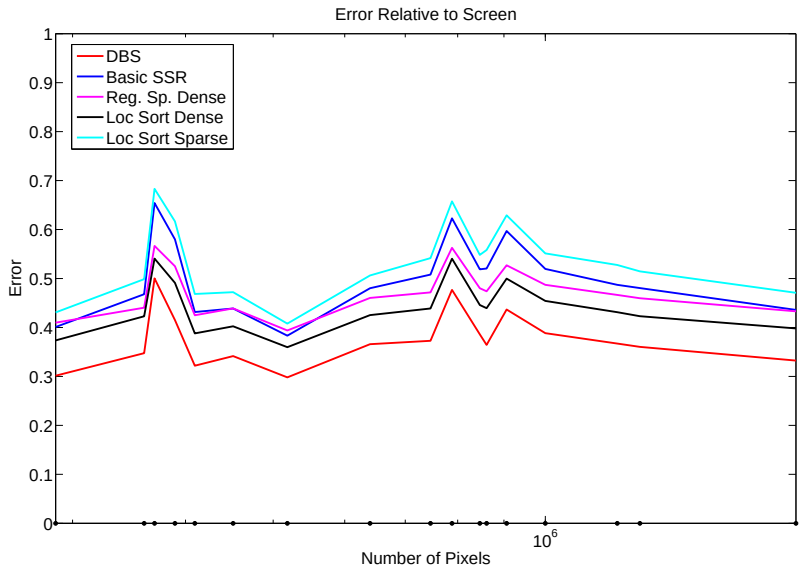

Figure 7: Error comparison for the implementations

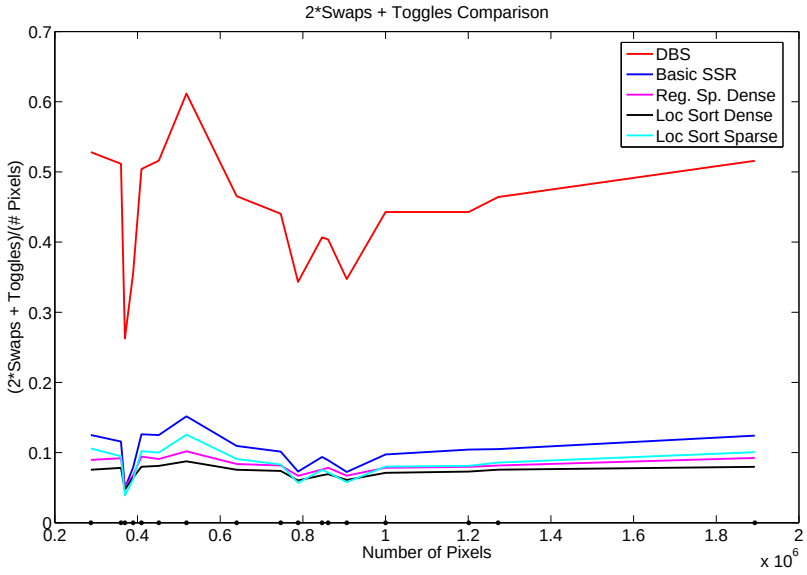

Figure 8: Swap and toggle comparison for the implementations

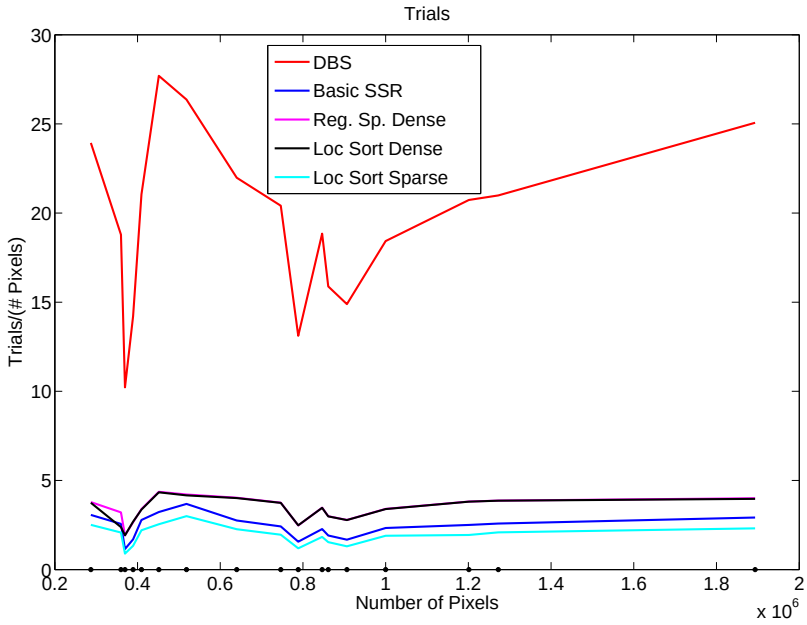

Figure 9: Trial comparison for the implementations

References

[1] Jan P. Allebach. "DBS: retrospective and future directions", *Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts VI*. Proc. SPIE vol. 3963, 2000.

[2] Kevin E. Spaulding, Rodney L. Miller, and Jay Schildkraut. "Methods for generating blue-noise dither matrices for digital halftoning", *Electronic Imaging*, vol. 6(2), pp. 208-230.

[3] Sang Ho Kim and Jan P. Allebach. "Impact of HVS Models on Model-based Halftoning", *Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts VI* Proc. SPIE vol. 4000, 2001.

[4] Robert W. Floyd and Louis Steinberg. *An Adaptive Algorithm for Spatial Grayscale*. Proceedings of the Society for Information Display **17** (2) 75-77, 1976.

[5] J.F. Jarvis, C.N. Judice and W.H. Ninke, *A Survey of Techniques for the Display of Continuous Tone Pictures on Bi-level Displays*. Computer Graphics and Image Processing **5** 13-40, 1976.

[6] P. Stucki, *MECCA - A Multiple Error Correcting Computation Algorithm for Bi-level Image Hard Copy Reproduction*. Research report RZ1060, IBM Research Laboratory, Zurich, Switzerland, 1981.

[7] Zhigang Fan, *A Simple Modification of Error-Diffusion Weights*. In the Proceedings of SPIE'92.