THE DESTINATION WORD ACCUMULATION IMPLEMENTATION OF BINARY MORPHOLOGY

Destination word accumulation (*dwa*) is a much more efficient method of implementing binary morphology. We go through the destination image one (32-bit) word at a time, computing its value based on the source image and the *Sel*, and then write that word to the destination image. This is more efficient than the rasterop version because the inner loop for each destination word is unrolled into the simplest set of operations, with no branches or iteration loops. Unlike the rasterop implementation, the *dwa* requires specialized routines for erosion and dilation using each *Sel*. However, it will be seen that this is not an impediment to its use.

To avoid writing special case functions for words that are at or near the image boundaries, *dwa* is implemented on interior pixels that are more than 32 pixels from the actual image boundary. The outermost 32 pixel border of pixels within the image are read from but not written to. If in an application it is desired to operate on *all* pixels in the image, the user must add a border of frame pixels before doing a *dwa*dilation or erosion. It is also necessary to add the frame pixels with the correct initial color. For dilation, the pixels added to the destination are always 0 (OFF). For erosion, they are OFF for ABC and ON for SBC. This could be automated if the pix had a field specifying the frame size, but I felt that adding this much machinery to slightly simplify the use of *dwa* morphology was not warranted. Just keep in mind that *dwa usage requires a border of extra frame pixels if you don't want to get get boundary artifacts, and you must initialize the pixels properly according to the convention you are using.*

Auto-generated dwa code interfaces

We provide two implementations that automatically generate code for *dwa* morphology.

- **Hit-only Sels**. The function `fmorphautogen()` in *fmorphauto.c* takes a *Sela*, an integer and an optional filename, and writes the C code for dilations, erosions, opening and closing, using each of the *Sel*s in the *Sela*. The integer is used to give all functions a unique name so that the C code created by multiple invocations of `fmorphautogen()` using different *Sela*s will all compile and link together in an application. For example, the program *prog/fmorphautogen* makes a *Sela* of the 58 *Sel*s that are generated by `selaAddBasic()`, and generates dwa C code for the four morphological operations for each*Sel*. It uses the integer "1" to generate the code in two files: *fmorphgen.1.c* and *fmorphgenlow.1.c*. We have compiled these files into the library. The

functions `pixMorphDwa_1()` and `pixFMorphopGen_1()` in *fmorphgen.1.c* are then available to be called by any application to perform a morphological operation:

- `PIX * pixMorphDwa_1(PIX *pixd, PIX *pixs, INT32 operation, char *selname);`
- `PIX * pixFMorphopGen_1(PIX *pixd, PIX *pixs, INT32 operation, char *selname);`

The use of the first two arguments is standard, as described above for the rasterop implementation. The operation is one of the set {`L_MORPH_DILATE, L_MORPH_ERODE, L_MORPH_OPEN, L_MORPH_CLOSE`}, and the *Sel* to be used is specified by the name string (the selname) associated with that *Sel*. The list of allowed name strings for the *Sela* is automatically extracted and placed in the file *fmorphgen.1.c*.

- **Hit-miss Sels**. The function `fhmtautogen()` in *fhmtauto.c* takes a *Sela*, an integer and an optional filename, and writes the C code for the hit-miss transform, using each of the *Sel*s in the *Sela*. The parameter usage is identical to that of the hit-only *Sel*s. The program *prog/fhmtautogen* makes a *Sela* of the 6 *Sel*s that are generated by `selaAddHitMiss()`, and generates dwa C code for the hit-miss transform for each *Sel*:
  - `PIX * pixHMTDwa_1(PIX *pixd, PIX *pixs, char *selname);`
  - `PIX * pixFHMTGen_1(PIX *pixd, PIX *pixs, char *selname);`

  The code generated by `fhmtautogen()` with index = 1 is in files *fhmtgen.1.c* and *fhmtgenlow.1.c*, and has been put in the library. The only restriction beyond size on the hit-miss *Sel*s is that each one must have at least one hit; otherwise, the code generated aborts with an error message.

There are several other practical things to note about using the *dwa* implementations:

- **You must add a border before dwa operations and remove it afterwards**. Add a 32 pixel border of frame pixels. For symmetric boundary conditions, initialize the added frame pixels to 0 for dilation and 1 for erosion. For asymmetric boundary conditions, always initialize the added frame pixels to 0. The functions `pixAddBorder()` and `pixRemoveBorder()` in *pix2.c* have been provided for this purpose.

- **Higher-level autogen'd functions are provided**:

  1. `pixMorphDwa_*() and pixHMTDwa_*()` automatically add the border and set the border pixels appropriately.
  2. `pixFMorphopGen_*() and pixFHMTGen_*` set the border pixels appropriately, but assume that these border pixels exist. If they don't exist, pixels in the proper image will be treated as border pixels.

- **Limit on *Sel* size**. To simplify the code that automatically generates dwa code, the hits in the *Sel* must not exceed 31 pixels in any direction away from the *Sel* origin. The autogen code will truncate any*Sel* that's larger, and the generated code may not compile. This limit means that with a centered *Sel* origin, the *Sel* can not be more than 63x63 pixels. This should be sufficient for most applications.

- **Higher-level functions for brick *Sel*s exit**. These are in *morphdwa.c*, and take care of all the grungy low-level details, including border pixels and their initialization.

- **Two programs that autogen code are provided**.

    - *Hits-only*. The program is `prog/fmorphautogen`.
    - *Hit-Miss Transform*. The program is `prog/fhmtautogen`

- **Reserved indices for autogen'd code** We reserve indices 1-9 for code built with either fmorphauto() and fhmtauto(). Hit-only code using indices 1 and 2, and hit-miss code using index 1 has been generated and inserted in the library. It is suggested that you use numbers larger than 9 in your own code to avoid collisions.

- **Autogen'd code can be linked directly into applications**. You have the option to add the code to the library and compile it there, or to compile and link the code directly into an executable. The latter is simpler.

- **Programs are provided for verification of correctness of autogen code**. The programs *prog/binmorph*_reg*, *dwamorph*_reg* and *fhmtauto_reg* do a variety of tests on the morphological functions. Most of these compare the dwa and full image rasterop implementations. Note that when a border is added for dilation, the added border pixels must be initialized to 0; for erosion, the value of the border pixels depends on whether you are using the asymmetric or symmetric boundary condition convention.

- **There is more information in the source code**. Consult the source for further details on usage.

Source : http://www.leptonica.com/binary-morphology.html