Code for CS 40 problems on Interfaces, Implementations, and Images

Norman Ramsey

Contents

1 Two-dimensional arrays

1.1 Interface

The interface file uses the same boilerplate as Hanson's CII.

2a ⟨*uarray2.h* 2a⟩≡

```
#ifndef ARRAY2_INCLUDED
#define ARRAY2_INCLUDED
#define T UArray2_T
typedef struct T *T;

⟨exported type definitions and function declarations 2b⟩
#undef T
#endif
```

The interface is modelled on Hanson's `UArray` interface, without operations that resize or copy the array, and with the addition of row-major and column-major map functions. The only significant design decision is the type of the apply functions. I've departed from Hanson in passing an additional argument to the apply function, which is the value of the array being mapped.

I also provide type definitions for the apply and map functions primarily to make it easy to declare a variable holding a map function.

2b ⟨*exported type definitions and function declarations* 2b⟩≡ (2a)

```
typedef void UArray2_applyfun(int i, int j, T array2, void *elem, void *cl);
typedef void UArray2_mapfun(T array2, UArray2_applyfun apply, void *cl);

extern T     UArray2_new   (int width, int height, int size);
extern void  UArray2_free  (T *array2);
extern int   UArray2_width (T array2);
extern int   UArray2_height(T array2);
extern int   UArray2_size  (T array2);
extern void *UArray2_at    (T array2, int i, int j);
extern void  UArray2_map_row_major(T array2, UArray2_applyfun apply, void *cl);
extern void  UArray2_map_col_major(T array2, UArray2_applyfun apply, void *cl);
```

1.2 Implementation

The key design decision is to represent the 2-dimensional array as an array of rows, each of which is itself an array of elements. With this array I keep the width, height, and size of the 2-dimensional array. This decision is not mere convenience, since if the array has height zero, there would be no way for me to extract the underlying width and size.

3a ⟨*uarray2.c* 3a⟩≡

```
#include "assert.h"
#include "mem.h"
#include "uarray.h"
#include "uarray2.h"

#define T UArray2_T

/*
 * Element (i, j) in the world of ideas maps to
 * rows[j][i] where the square brackets stand for access
 * to a Hanson UArray_T
 */
struct T {
        int width, height;
        int size;
        UArray_T rows; /* UArray_T of 'height' UArray_Ts,
                          each of length 'width' and size 'size' */
};
⟨private functions 3b⟩
⟨exported functions 4b⟩
```

Extracting a row requires the use of `UArray_at`, which is always tricky, because it's easy to forget to cast and deference the result. To reduce my chances of making a mistake, I define a `static inline` function to extract a row. The `static inline` property guarantees that the compiler will eliminate the procedure-call overhead normally associated with functions.

3b ⟨*private functions* 3b⟩≡ (3a) 4a▷

```
static inline UArray_T row(T a, int j)
{
        UArray_T *prow = UArray_at(a->rows, j);   /* Ramsey idiom */
        return *prow;
}
```

The `row` function is not protected with an assertion, mostly because I want it to be a very low-overhead operation. I can get away with it because the function is `static`, so I can identify every call site and be sure each call site is protected by an assertion.

Here's a function to check the representation invariant.

4a ⟨*private functions* 3b⟩+≡ (3a) ◁3b

```
static int is_ok(T a)
{
        return a && UArray_length(a->rows) == a->height &&
               UArray_size(a->rows) == sizeof(UArray_T) &&
               (a->height == 0 || (UArray_length(row(a, 0)) == a->width
                                   && UArray_size  (row(a, 0)) == a->size));
}
```

1.2.1 Memory management

Given the representation, as Fred Brooks said, the code almost writes itself. To create an array, I intialize the structure, and I put a fresh array in every element of the `rows` field. The elements of the rows themselves are not initialized, but they are fresh memory, as guaranteed by `UArray_new`.

4b ⟨*exported functions* 4b⟩≡ (3a) 4c▷

```
T UArray2_new(int width, int height, int size)
{
        int i;  /* interates over row number */
        T array;
        NEW(array);
        array->width  = width;
        array->height = height;
        array->size   = size;
        array->rows   = UArray_new(height, sizeof(UArray_T));
        for (i = 0; i < height; i++) {
                UArray_T *rowp = UArray_at(array->rows, i);
                *rowp = UArray_new(width, size);
        }
        assert(is_ok(array));
        return array;
}
```

To free an array, I undo everything I did in `UArray_new`. I have to be careful to call `UArray_free` and not `FREE`, or I will create a memory leak. (Such leaks can be discovered and eliminated using `valgrind`.)

4c ⟨*exported functions* 4b⟩+≡ (3a) ◁4b 5a▷

```
void UArray2_free(T *array2)
{
        int i;
        assert(array2 && *array2);
        for (i = 0; i < (*array2)->height; i++) {
                UArray_T p = row(*array2, i);
                UArray_free(&p);
        }
        UArray_free(&(*array2)->rows);
        FREE(*array2);
}
```

1.2.2 Location function

To point to a boxed element, first I get the row, then I point to the element within the row. The assertions protect me against a core dump in case somebody passes in a null pointer.

5a ⟨*exported functions* 4b⟩+≡ (3a) ◁4c 5b▷

```
void *UArray2_at(T array2, int i, int j)
{
        assert(array2);
        return UArray_at(row(array2, j), i);
}
```

1.2.3 Array metrics

Functions to return metrics are trivial.

5b ⟨*exported functions* 4b⟩+≡ (3a) ◁5a 6a▷

```
int UArray2_height(T array2)
{
        assert(array2);
        return array2->height;
}

int UArray2_width(T array2)
{
        assert(array2);
        return array2->width;
}

int UArray2_size(T array2)
{
        assert(array2);
        return array2->size;
}
```

1.2.4 UArray mapping functions

The map functions are the last to be implemented. The only fancy bit is that in row-major form I can pull the call to `row` out of the inner loop. In order to avoid excess memory traffic, I load `array2->height` and `array2->width` into local (register) variables. I know that these values never change, but it's harder for the compiler to tell that a call to `UArray_at` can't change the value of `array2->height`, for example.

6a ⟨*exported functions* 4b⟩+≡ (3a) ◁5b 6b▷

```
void UArray2_map_row_major(T array2,
                           void apply(int i, int j, T array2,
                                      void *elem, void *cl),
                           void *cl)
{
        assert(array2);
        int h = array2->height;  /* keeping height and width in registers */
        int w = array2->width;   /* avoids extra memory traffic           */
        for (int j = 0; j < h; j++) {
                /* don't want row/UArray_at in inner loop */
                UArray_T thisrow = row(array2, j);
                for (int i = 0; i < w; i++)
                        apply(i, j, array2, UArray_at(thisrow, i), cl);
        }
}
```

In column-major form, `row` must be in the inner loop.

6b ⟨*exported functions* 4b⟩+≡ (3a) ◁6a

```
void UArray2_map_col_major(T array2,
                           void apply(int i, int j, T array2,
                                      void *elem, void *cl),
                           void *cl)
{
        assert(array2);
        int h = array2->height;  /* keeping height and width in registers */
        int w = array2->width;   /* avoids extra memory traffic           */
        for (int i = 0; i < w; i++)
                for (int j = 0; j < h; j++)
                        apply(i, j, array2, UArray_at(row(array2, j), i), cl);
}
```

2 Two-dimensional bitmaps

The design and implementation are as for arrays.

2.1 Interface

7 ⟨*bit2.h* 7⟩≡

```
#ifndef BIT2_INCLUDED
#define BIT2_INCLUDED
#define T Bit2_T
typedef struct T *T;
extern T    Bit2_new (int width, int height);
extern void Bit2_free(T *bit2);
extern int  Bit2_width(T bit2);
extern int  Bit2_height(T bit2);
extern int  Bit2_get(T bit2, int i, int j);
extern int  Bit2_put(T bit2, int i, int j, int bit);
extern void Bit2_map_row_major(T bit2,
                               void apply(int i, int j, T bitmap,
                                          int bit, void *cl),
                               void *cl);
extern void Bit2_map_col_major(T bit2,
                               void apply(int i, int j, T bitmap,
                                          int bit, void *cl),
                               void *cl);
#undef T
#endif
```

2.2 Implementation

8a ⟨*bit2.c* 8a⟩≡

```
#include <stdlib.h>

#include "assert.h"
#include "bit.h"
#include "bit2.h"
#include "mem.h"
#include "uarray.h"

#define T Bit2_T

struct T {
        int width, height;
        UArray_T rows; /* UArray_T of 'height' Bit_Ts, each of length 'width'
                        * Element (i, j) in the world of ideas maps to
                        * bit number 'i' in *(Bit_T*)UArray_at(rows, j)
                        */
};
⟨private functions 8b⟩
⟨exported functions 9a⟩
```

The architecture of the solution is almost identical to the architecture of the 2-dimensional array.

8b ⟨*private functions* 8b⟩≡ (8a) 8c▷

```
static inline Bit_T row(T a, int j) {
        return *(Bit_T *)UArray_at(a->rows, j);   /* cast is Hanson idiom */
}
```

Invariant check.

8c ⟨*private functions* 8b⟩+≡ (8a) ◁8b 10b▷

```
static int is_ok(T b) {
        return b && UArray_length(b->rows) == b->height &&
                (b->height == 0 || Bit_length(row(b, 0)) == b->width);
}
```

2.2.1 Memory management

Allocate.

9a ⟨*exported functions* 9a⟩≡ (8a) 9b▷

```
T Bit2_new(int width, int height)
{
        int i;  /* interates over row number */
        T bitmap;
        NEW(bitmap);
        bitmap->width  = width;
        bitmap->height = height;
        bitmap->rows   = UArray_new(height, sizeof(Bit_T));
        for (i = 0; i < height; i++) {
                Bit_T *element = UArray_at(bitmap->rows, i);
                *element = Bit_new(width);
        }
        assert(is_ok(bitmap));
        return bitmap;
}
```

Free.

9b ⟨*exported functions* 9a⟩+≡ (8a) ◁9a 9c▷

```
void Bit2_free(T *bitmap)
{
        int i;
        assert(bitmap && *bitmap);
        for (i = 0; i < (*bitmap)->height; i++) {
                Bit_T p = row(*bitmap, i);
                Bit_free(&p);
        }
        UArray_free(&(*bitmap)->rows);
        FREE(*bitmap);
}
```

2.2.2 Get and put functions

9c ⟨*exported functions* 9a⟩+≡ (8a) ◁9b 10a▷

```
int Bit2_get(T bitmap, int i, int j)
{
        assert(bitmap != NULL);
        return Bit_get(row(bitmap, j), i);
}

int Bit2_put(T bitmap, int i, int j, int bit)
{
        assert(bitmap != NULL);
        return Bit_put(row(bitmap, j), i, bit);
}
```

2.2.3 Bitmap metrics

10a ⟨*exported functions* 9a⟩+≡ (8a) ◁9c 10c▷

```
int Bit2_height(T bitmap)
{
        assert(bitmap != NULL);
        return bitmap->height;
}

int Bit2_width(T bitmap)
{
        assert(bitmap != NULL);
        return bitmap->width;
}
```

2.2.4 Bitmap mapping functions

For row-major mapping I use `Bit_map`, which means I have to wrap the original (apply, closure) pair in a new closure. The new closure also tracks the current `Bit2_T` and the current row number, because they need to be passed to the original `apply` function, but they aren't provided by `Bit_map`.

10b ⟨*private functions* 8b⟩+≡ (8a) ◁8c

```
struct bitcl {
        /* original apply */
        void (*apply)(int i, int j, T bitmap2, int bit, void *cl);
        void *cl;   /* original cl */
        int rownum; /* row number of this bitmap */
        T bitmap;   /* bitmap we're visiting */
};

/* apply the original */
static void bit_apply(int n, int bit, void *cl)
{
        struct bitcl *orig = cl; /* contains the original 'apply' and 'cl' */
        orig->apply(n, orig->rownum, orig->bitmap, bit, orig->cl);
}
```

10c ⟨*exported functions* 9a⟩+≡ (8a) ◁10a 11▷

```
void Bit2_map_row_major(T bitmap,
                        void apply(int i, int j, T bitmap,
                                   int bit, void *cl),
                        void *cl)
{
        assert(bitmap != NULL);
        int h = bitmap->height;
        for (int i = 0; i < h; i++) {
                struct bitcl wrapper = { apply, cl, i, bitmap };
                Bit_map(row(bitmap, i), bit_apply, &wrapper);
        }
}
```

Column-major mapping uses a straightforward nested loop.

11 ⟨*exported functions* 9a⟩+≡ (8a) ◁10c

```
void Bit2_map_col_major(T bitmap,
                        void apply(int i, int j, T bitmap,
                                   int bit, void *cl),
                        void *cl)
{
        assert(bitmap != NULL);
        int h = bitmap->height;
        int w = bitmap->width;
        /* Noah Mendelsohn fixed bug in order of first two
           arguments to "apply" 9/15/14 */
        for (int i = 0; i < w; i++)
                for (int j = 0; j < h; j++)
                        apply(i, j, bitmap, Bit_get(row(bitmap, j), i), cl);
}
```

3 Sudoku checker

Here's the structure of the implementation:

12 ⟨*sudoku.c* 12⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include "mem.h"
#include "pnmrdr.h"
#include "assert.h"
#include "bit.h"
#include "uarray2.h"

/*
 * grayscale pixmap structure
 */
typedef struct graymap {
        unsigned denominator;    /* shared among all pixels         */
        UArray2_T pixels;        /* 2D array of unsigned numerators */
} *GM;

GM pgmread(FILE *fp);
void pgmwrite(FILE *fp, GM graymap);
⟨definitions of pgmread and pgmwrite 13⟩
⟨functions for visiting rows, columns, and blocks 15a⟩
⟨definition of main 16⟩
```

3.1 Reading and writing portable graymaps

Because a portable graymap has its pixels in row-major order, I demonstrate the use of `UArray2_map_row_major` to read the pixels.

13 ⟨*definitions of* `pgmread` *and* `pgmwrite` 13⟩≡ (12) 14a▷

```
static void mapread(int i, int j, UArray2_T pixels, void *elem, void *cl)
{
        (void)elem;       /* ignore uninitialized array element */
        Pnmrdr_T r = cl;
        unsigned *p = UArray2_at(pixels, i, j);
        *p = Pnmrdr_get(r);
}

GM pgmread(FILE *fp)
{
        GM graymap;
        Pnmrdr_T r = Pnmrdr_new(fp);
        Pnmrdr_mapdata data = Pnmrdr_data(r);
        UArray2_T pixels = UArray2_new(data.width, data.height, sizeof(unsigned));
        UArray2_map_row_major(pixels, mapread, r);
        NEW(graymap);
        graymap->denominator = data.denominator;
        graymap->pixels = pixels;
        Pnmrdr_free(&r);
        return graymap;
}
```

Writing a portable graymap is more complicated, because the output is supposed to be no more than 70 columns wide. I track the next column so I can put in newlines. I also track the row number so that I can put a newline after each row.

14a ⟨*definitions of* `pgmread` *and* `pgmwrite` 13⟩+≡ (12) ◁13

```
struct writecl {
        FILE *fp;         /* output file */
        unsigned column;  /* number of chars written to last row of file */
        int row_number;   /* row number of the next element to be written */
};

static void mapwrite(int i, int j, UArray2_T pixels, void *elem, void *cl)
{
        struct writecl *wcl = cl;
        unsigned *pnum = elem;    /* pointer to numerator             */
        (void) pixels;            /* not used; avoid a warning        */
        (void) i;                 /* column not used; avoid a warning */
        ⟨if we need a newline, write one and reset the closure's row and column 14b⟩
        fprintf(wcl->fp, "%1u", *pnum);
}

void pgmwrite(FILE *fp, GM graymap)
{
        struct writecl wcl = { fp, 0, 0 };
        fprintf(fp, "P2\n# written by pgmwrite in in sudoku.nw\n%d %d\n%d\n",
                UArray2_width(graymap->pixels), UArray2_height(graymap->pixels),
                graymap->denominator);
        UArray2_map_row_major(graymap->pixels, mapwrite, &wcl);
        fprintf(fp, "\n");
}
```

This sort of thing, which is peripheral to the main algorithm, belongs in a separate code chunk.

14b ⟨*if we need a newline, write one and reset the closure's row and column* 14b⟩≡ (14a)

```
        if (wcl->column++ >= 69 || wcl->row_number != j) {
                fprintf(wcl->fp, "\n");
                wcl->column = 1;  /* account for char to be written below */
                wcl->row_number = j;
        }
```

3.2 Declarations of the array-visitor functions

Here are the functions talked about in the design document: I can visit a row, a column, or a block within an array. Although a row and a column are both special cases of blocks, I've chosen to define three functions for clarity.

15a ⟨*functions for visiting rows, columns, and blocks* 15a⟩≡ (12) 15b▷

```
static void visit_row  (UArray2_T array, int row,
                        UArray2_applyfun apply, void *cl);
static void visit_col  (UArray2_T array, int col,
                        UArray2_applyfun apply, void *cl);
static void visit_block(UArray2_T array, int col, int row,
                        int width, int height,
                        UArray2_applyfun apply, void *cl);
```

3.3 Detecting sudoku violations

The way the main program works is to check each row, column, and relevant block for a case where the same element appears twice. If this violation of the Sudoku rule occurs, the matrix is not a solved Sudoku puzzle, and we immediately exit with a nonzero code. Function `fail_on_duplicate` does the work; `cl` is a bit vector tracking the set of elements already seen.

15b ⟨*functions for visiting rows, columns, and blocks* 15a⟩+≡ (12) ◁15a 17a▷

```
static void fail_on_duplicate(int i, int j, UArray2_T bitmap,
                              void *elem, void *cl)
{
        (void)bitmap;     /* not used, avoid compiler warning */
        (void)i;          /* not used, avoid compiler warning */
        (void)j;          /* not used, avoid compiler warning */
        Bit_T seen = cl;
        unsigned *pnum = elem;   /* pointer to numerator */
        unsigned n = *pnum;
        if (Bit_get(seen, n))
                exit(1);         /* saw a duplicate */
        else
                Bit_put(seen, n, 1);
}
```

16 ⟨*definition of* main 16⟩≡ (12)

```
int main(int argc, char *argv[])
{
        assert(argc <= 2);
        FILE *input;
        if (argc == 2) {
                input = fopen(argv[1], "rb");
                if (!input) {
                        perror(argv[1]);
                        exit(1);
                }
        } else {
                input = stdin;
        }
        GM graymap = pgmread(input);
        if (input != stdin)
                fclose(input);
        if (UArray2_height(graymap->pixels) != 9
            || UArray2_width(graymap->pixels) != 9)
                exit(1);
        Bit_T rowset = Bit_new(10);  /* set of numbers seen in a row    */
        Bit_T colset = Bit_new(10);  /* set of numbers seen in a column */
        for (int i = 0; i < 9; i++) {
                Bit_clear(rowset, 1, 9);
                Bit_clear(colset, 1, 9);
                visit_row(graymap->pixels, i, fail_on_duplicate, rowset);
                visit_col(graymap->pixels, i, fail_on_duplicate, colset);
        }
        Bit_free(&rowset);
        Bit_free(&colset);
        Bit_T blockset = Bit_new(10);  /* set of numbers seen in a 3x3 block */
        for (int i = 0; i < 9; i += 3) {
                for (int j = 0; j < 9; j += 3) {
                        Bit_clear(blockset, 1, 9);
                        visit_block(graymap->pixels, i, j, 3, 3,
                                    fail_on_duplicate, blockset);
                }
        }
        Bit_free(&blockset);
        UArray2_free(&graymap->pixels);
        FREE(graymap);
        return 0;
}
```

3.4 Definitions of the array-visitor functions

Nothing interesting to see here. Move along.

17a ⟨*functions for visiting rows, columns, and blocks* 15a⟩+≡ (12) ◁15b 17b▷

```
static void visit_row(UArray2_T array, int row,
                      UArray2_applyfun apply, void *cl)
{
        int w = UArray2_width(array);
        for (int i = 0; i < w; i++)
                apply(i, row, array, UArray2_at(array, i, row), cl);
}
```

17b ⟨*functions for visiting rows, columns, and blocks* 15a⟩+≡ (12) ◁17a 17c▷

```
static void visit_col  (UArray2_T array, int col,
                        UArray2_applyfun apply, void *cl)
{
        int h = UArray2_height(array);
        for (int j = 0; j < h; j++)
                apply(col, j, array, UArray2_at(array, col, j), cl);
}
```

17c ⟨*functions for visiting rows, columns, and blocks* 15a⟩+≡ (12) ◁17b

```
static void visit_block(UArray2_T array, int col, int row,
                        int width, int height,
                        UArray2_applyfun apply, void *cl)
{
        for (int i = col; i < col + width; i++)
                for (int j = row; j < row + height; j++)
                        apply(i, j, array, UArray2_at(array, i, j), cl);
}
```

4 Removal of black edges

This is what we promise:

18a ⟨*unblackedges.h* 18a⟩≡

```
#include "bit2.h"

extern void remove_black_edges(Bit2_T bitmap);
  /* remove black edge pixels from the bitmap given */
```

Here's the structure of the implementation:

18b ⟨*unblackedges.c* 18b⟩≡

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "pnmrdr.h"
#include "assert.h"
#include "unblackedges.h"
#include "seq.h"
#include "pbmio.h"
```

⟨*functions to detect and remove black edge pixels* 19a⟩

4.1 Black-edge detection

A pixel is considered a black edge pixel if it is black and lies on an edge, or if it is next to a black edge pixel. Finding all of the black edge pixels requires treating the bitmap as an undirected graph in which pixels are vertices and there is a graph edge connecting each pair of adjacent black pixels. The black edge pixels are all those reachable from the periphery of the image.[1]

I cannot immediately turn black edge pixels to white, because that might stop me from finding an adjacent black edge pixel in a perpendicular direction. So I use two bitmaps. The original, called `bitmap`, contains the original image. The second bitmap, called `blackedge`, starts out as all zeroes, but eventually it contains a one for every black edge pixel. With the two maps in place, it's a simple matter to clear the black edge pixels using `Bit2_map`:

19a ⟨*functions to detect and remove black edge pixels* 19a⟩≡ (18b) 19b▷

```
static void clear_blackedge_bit(int i, int j, Bit2_T bitmap, int bit, void *cl)
{
        Bit2_T blackedge = cl;
        (void)bit; /* avoid compiler warning for unused argument */
        if (Bit2_get(blackedge, i, j))
                Bit2_put(bitmap, i, j, 0);
}

static void clear_blackedge_bits(Bit2_T bitmap, Bit2_T blackedge)
{
        Bit2_map_row_major(bitmap, clear_blackedge_bit, blackedge);
}
```

To set the black edge pixels, I require a classic graph traversal. I dare not try depth-first search by simple recursion; the depth of recursion is limited only by the length of the longest path through black edge pixels, and my experiments show that a scanned page with modest black edges could nevertheless require a stack depth of over a million. So I put the stack in an auxiliary data structure. Using an explicit stack actually makes it easy to try breadth-first search as well, and it turns out that breadth-first search can use forty times less space than depth-first search.

Hanson's CII provides `Seq_T`, an abstraction that is ideal for implementing both stacks and queues very efficiently. Unfortunately a `Seq_T` stores only pointers, or things the size of a pointer, and I need to store (i, j) coordinate pairs. So I use two queues `iq` and `jq` in tandem.

The fundamental operation on which others are built is the static inline function `maybe_enqueue`, which enqueues a pixel's coordinates if the pixel is black.

19b ⟨*functions to detect and remove black edge pixels* 19a⟩+≡ (18b) ◁19a 20▷

```
static inline void maybe_enqueue(Bit2_T bitmap, int i, int j,
                                 Seq_T iq, Seq_T jq)
{
        if (Bit2_get(bitmap, i, j)) {
                Seq_addhi(iq, (void *)(uintptr_t)i);
                Seq_addhi(jq, (void *)(uintptr_t)j);
        }
}
```

[1] From here on, I use the word "periphery" to refer to the boundaries of the image and the word "edge" to refer to the implicit graph.

By enqueuing at the high end and dequeueing at the low end, I create a breadth-first algorithm. Should I wish to try depth-first, I need only to change `addhi` to `addlo`.

In my algorithm, if I pull a pixel from the queue and the pixel is not visited, I wish for it to be safe to enqueue all of its neighbors. I establish safety through these invariants:

- An (i, j) pair stored in the queue either refers to an *interior* pixel or is already marked in the `visited` map.
- Before any pixel is removed from the queue, all pixels on the perpiphery of the image (i.e., all non-interior pixels) have been visited and so marked.

Here's the outline of the search: create auxiliary data structures; visit every pixel along the periphery and enqueue every black neighbor of a black pixel; visit pixels from the queue until the queue is empty; and finally, free the data structures.

20 ⟨*functions to detect and remove black edge pixels* 19a⟩+≡ (18b) ◁19b 22b▷

```
static void set_blackedge_bits(Bit2_T bitmap, Bit2_T blackedge)
{
        int h = Bit2_height(bitmap);
        int w = Bit2_width(bitmap);

        Bit2_T visited = Bit2_new(w, h);  /* for classic depth-first search  */
        int nvisited   = 0;               /* number of pixels in visited map */
        Seq_T iq = Seq_new((2 * w) + (2 * h));
        Seq_T jq = Seq_new((2 * w) + (2 * h));

        ⟨visit left and right columns; set black bits in blackedge and enqueue the adjacent bit 21a⟩;
        ⟨visit top and bottom rows; set black bits in blackedge and enqueue the adjacent bit 21b⟩;

        int maxlen = Seq_length(iq);  /* maximum length of queue */
        int blackened = 0;            /* number of black edge pixels detected */
        int tried = 2 * (w + h - 1);  /* number of touches of pixels */
        ⟨empty the queue, marking blackedge and visited, and tracking statistics 22a⟩;
        Bit2_free(&visited);
        Seq_free(&iq);
        Seq_free(&jq);
        if (getenv("UNBLACKVERBOSE"))
                fprintf(stderr, "Tried %d%% (%.1f%% distinct) and unblacked "
                        "%.1f%% of %dx%d == %d pixels; max q = %d\n",
                        (100 * tried + (w * h / 2)) / (w * h),
                        (100.0 * (double)nvisited)  / (double) (w * h),
                        (100.0 * (double)blackened) / (double) (w * h),
                        w, h, w * h, maxlen);
}
```

In violation of the specification, I print statistics to show how much work is done.

The initial visits are straightforward. Here I mark every pixel in the leftmost and rightmost columns as visited; for each such pixel that is black, I mark it as such and possibly enqueue its neighboring pixel. The algorithm is much like visiting a root set in a garbage collector.

21a ⟨*visit left and right columns; set black bits in* `blackedge` *and enqueue the adjacent bit* 21a⟩≡ (20)

```
for (int row = 0; row < h; row++) {
        if (Bit2_get(bitmap, 0, row)) {
                Bit2_put(blackedge, 0, row, 1);
                maybe_enqueue(bitmap, 1, row, iq, jq);
        }
        Bit2_put(visited, 0, row, 1);
        if (Bit2_get(bitmap, w - 1, row)) {
                Bit2_put(blackedge, w - 1, row, 1);
                maybe_enqueue(bitmap, w - 2, row, iq, jq);
        }
        Bit2_put(visited, w - 1, row, 1);
}
nvisited += 2 * h;
```

The same, for for the top and bottom rows.

21b ⟨*visit top and bottom rows; set black bits in* `blackedge` *and enqueue the adjacent bit* 21b⟩≡ (20)

```
for (int col = 0; col < w; col++) {
        if (Bit2_get(bitmap, col, 0)) {
                Bit2_put(blackedge, col, 0, 1);
                maybe_enqueue(bitmap, col, 1, iq, jq);
        }
        Bit2_put(visited, col, 0, 1);
        if (Bit2_get(bitmap, col, h - 1)) {
                Bit2_put(blackedge, col, h - 1, 1);
                maybe_enqueue(bitmap, col, h - 2, iq, jq);
        }
        Bit2_put(visited, col, h - 1, 1);
}
nvisited += 2 * (w - 2);  /* do not double-count corner pixels */
```

Here is the interesting bit: dequeue (i, j), and if it isn't visited already, mark it as visited. Then, if it is black, mark it as a black edge pixel and enqueue each of its black neighbors.

Enqueuing a pixel that is already visited is a bit of a waste of work, since the pixel will take up space until dequeued, at which point it is just thrown away again. But I think the algorithm with the extra enqueueing is easier to understand and get right. Profiling will show if the extra work matters.

22a ⟨*empty the queue, marking* `blackedge` *and* `visited`*, and tracking statistics* 22a⟩≡ (20)

```
while (Seq_length(iq) > 0) {
        tried++;
        if (Seq_length(iq) > maxlen) maxlen = Seq_length(iq);
        int i = (int)(uintptr_t) Seq_remlo(iq);
        int j = (int)(uintptr_t) Seq_remlo(jq);
        if (!Bit2_get(visited, i, j)) {
                Bit2_put(visited, i, j, 1);
                nvisited++;
                if (Bit2_get(bitmap, i, j)) {
                        blackened++;
                        Bit2_put(blackedge, i, j, 1);
                        maybe_enqueue(bitmap, i + 1, j    , iq, jq);
                        maybe_enqueue(bitmap, i - 1, j    , iq, jq);
                        maybe_enqueue(bitmap, i    , j + 1, iq, jq);
                        maybe_enqueue(bitmap, i    , j - 1, iq, jq);
                }
        }
}
assert(Seq_length(jq) == 0);
```

The `remove_black_edges` function combines all the components I've built. If the input bitmap is only one pixel high or wide, I simply clear all pixels.

22b ⟨*functions to detect and remove black edge pixels* 19a⟩+≡ (18b) ◁20

```
static void clear_pixel(int i, int j, Bit2_T bitmap, int bit, void *cl)
{
        (void)bit; (void)cl; /* stop compiler bleats */
        Bit2_put(bitmap, i, j, 0);
}

void remove_black_edges(Bit2_T bitmap)
{
        int h = Bit2_height(bitmap);
        int w = Bit2_width(bitmap);
        if (w == 1 || h == 1)
                Bit2_map_row_major(bitmap, clear_pixel, NULL);
        else {
                assert(w >= 2 && h >= 2);
                Bit2_T blackedge = Bit2_new(w, h);
                set_blackedge_bits(bitmap, blackedge);
                clear_blackedge_bits(bitmap, blackedge);
                Bit2_free(&blackedge);
        }
}
```

5 Removal of black edges using an efficient idea from John Dias

This is what we promise:

23a ⟨*unblackedges.h* 23a⟩≡

```
#include "bit2.h"

extern void remove_black_edges(Bit2_T bitmap);
  /* remove black edge pixels from the bitmap given */
```

Here's the structure of the implementation:

23b ⟨*unblackgc.c* 23b⟩≡

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "pnmrdr.h"
#include "assert.h"
#include "unblackedges.h"
#include "seq.h"
#include "pbmio.h"

⟨private auxiliary functions 26a⟩
⟨functions to detect and remove black edge pixels 25a⟩
```

5.1 Black-edge detection

A pixel is considered a black edge pixel if it is black and lies on an edge, or if it is next to a black edge pixel. Finding all of the black edge pixels requires treating the bitmap as an undirected graph in which pixels are vertices and there is a graph edge connecting each pair of adjacent black pixels. The black edge pixels are all those reachable from the periphery of the image.[2] To find reachable pixels, I follow a suggestion of John Dias and use the same algorithm that a garbage collector uses to find reachable objects on a heap:

1. A given set of initial objects is known to be reachable and is placed on a queue.
2. The algorithm loops until the queue is empty. At each step it removes an object from the queue and "visits" it. When an object is removed from the queue, every object that it points to is checked, and if not already visited, it is added to the queue.

   The algorithm needs to keep a "visited set."
3. When the queue is empty, the set of objects visited is exactly the same set that is reachable by following pointers from the initial objects.

To adapt this algorithm to detect black edge pixels, I use the the loop invariant that every black edge pixel from the original image is in one of these two states:

- It is white in the current image.
- It can be reached from a pixel in the queue by following only currently black neighbors.

I establish the invariant in one step that traverses the periphery:

- Every black pixel on the periphery is turned white and placed in a queue.

Then, at each step of the loop, I maintain the invariant as follows:

- When a pixel is removed from the queue, all of its black neighbors are turned white and are added to the queue (by function `maybe_enqueue`).

When the queue is empty, the invariant guarantees that every black edge pixel is now white. (To prove that the program whitens *only* black edge pixels we need the additional invariant that every pixel in the queue is a black edge pixel.)

In mathematical terms, the way this solution works is that I generalize the inductive definition in the problem to replace the base case. The base case in the original problem is "black pixels on the periphery." The base case in my algorithm is "pixels in the queue."

- I establish acceptable initial conditions by placing the black peripheral pixels in the queue.
- The algorithm is guaranteed to terminate because adding to the queue requires turning a black pixel white, and this step can be done only finitely many times.
- At termination the queue is empty, which guarantees that all black edge pixels are now white.

[2] From here on, I use the word "periphery" to refer to the boundaries of the image and the word "edge" to refer to the implicit graph.

I need only one auxiliary data structure: the queue. Hanson's CII provides `Seq_T`, an abstraction that is ideal for implementing both stacks and queues very efficiently. Unfortunately a `Seq_T` stores only pointers, or things the size of a pointer, and I need to store (i, j) coordinate pairs. So I use two sequences `iq` and `jq` in tandem.[3]

25a ⟨*functions to detect and remove black edge pixels* 25a⟩≡ (23b)

```
void remove_black_edges(Bit2_T bitmap) {
  int h = Bit2_height(bitmap);
  int w = Bit2_width(bitmap);
  Seq_T iq = Seq_new(2*w+2*h); // queue of column indices
  Seq_T jq = Seq_new(2*w+2*h); // queue of row indices

  ⟨enqueue black pixels from the periphery 26b⟩

  int maxlen = Seq_length(iq);  // maximum length of queue
  int nvisited = 0;             // total number of pixels visited
  ⟨empty the queue, tracking maxlen 25b⟩
  Seq_free(&iq);
  Seq_free(&jq);
  if (getenv("UNBLACKVERBOSE"))
    fprintf(stderr, "Unblacked %.1f%% of %dx%d == %d pixels; max q = %d\n",
            (double) (100 * nvisited + (w * h / 2)) / (w * h),
            w, h, w*h, maxlen);
}
```

Here's the core of the algorithm:

25b ⟨*empty the queue, tracking* `maxlen` 25b⟩≡ (25a)

```
while (Seq_length(iq) > 0) {
  if (Seq_length(iq) > maxlen) maxlen = Seq_length(iq);
  int i = (int)(uintptr_t) Seq_remlo(iq);
  int j = (int)(uintptr_t) Seq_remlo(jq);
  assert(Bit2_get(bitmap, i, j) == 0); // invariant
  nvisited++;
  if (i+1<w) maybe_enqueue(bitmap, i+1, j, iq, jq);
  if (i>0)   maybe_enqueue(bitmap, i-1, j, iq, jq);
  if (j+1<h) maybe_enqueue(bitmap, i, j+1, iq, jq);
  if (j>0)   maybe_enqueue(bitmap, i, j-1, iq, jq);
}
assert(Seq_length(jq) == 0);
```

By using a more complex invariant that places only interior pixels in the queue, I could avoid the conditional tests. But this code already spends over 90% of its time reading and writing bitmaps, not finding black pixels, so the game is not worth the candle.

[3] I did an experiment using a stack instead of a queue—equivalent to depth-first search instead of bread-first search—and the stack can use up to forty times as much space as the queue.

The fundamental operation on which others are built is the static inline function `maybe_enqueue`, which enqueues a pixel's coordinates if the pixel is black. Crucial to the algorithm is that as it goes on the queue, the pixel is turned white.

26a ⟨*private auxiliary functions* 26a⟩≡ (23b)

```
static inline void maybe_enqueue(Bit2_T bitmap, int i, int j, Seq_T iq, Seq_T jq) {
  if (Bit2_get(bitmap, i, j)) {
    Bit2_put(bitmap, i, j, 0);
    Seq_addhi(iq, (void *)(uintptr_t)i);
    Seq_addhi(jq, (void *)(uintptr_t)j);
  }
}
```

If I wanted to try a stack-based algorithm (depth-first search instead of breadth-first search), I would simply replace `addhi` with `addlo`.

All that's left to implement is the initial enqueuing, which is straightforward.

26b ⟨*enqueue black pixels from the periphery* 26b⟩≡ (25a)

```
for (int row = 0; row < h; row++) {
  maybe_enqueue(bitmap, 0,   row, iq, jq);
  maybe_enqueue(bitmap, w-1, row, iq, jq);
}
for (int col = 0; col < w; col++) {
  maybe_enqueue(bitmap, col, 0,   iq, jq);
  maybe_enqueue(bitmap, col, h-1, iq, jq);
}
```