The Zhang Suen Thinning Algorithm

Gagarine Yaikhom

I. Introduction

We implement[1] the *Zhang-Suen Thinning Algorithm* for skeletonisation of binary images [T. Y. Zhang and C. Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Communications of the ACM*, 27(3):236–239, 1984].

⟨*Functions*⟩≡
```
#define V(p,r,c) *(p + r * width + c)
#define has_bpat(n,p) (((n) & (p)) == (p))
void zhang_suen_thinning(
    uint8_t *skeleton, uint8_t *image, uint8_t *mask,
    uint16_t width, uint16_t height)
{
    size_t num_pixels = width * height;
    memcpy(skeleton, image, num_pixels);
    memset(mask, 1, num_pixels);
    ⟨Define constant variables⟩
    int not_done = 0;
    do {
        ⟨Prepare and apply thinning mask⟩
    } while(not_done);
}
#undef V
#undef has_bpat
```

⟨*Prepare and apply thinning mask*⟩≡
```
for (int step = 0; step < 2; ++step) {
    not_done = 0;
    ⟨Process pixels using neighbours⟩
    ⟨Update skeleton using thinning mask⟩
}
```

Since the border pixels are not processed, we define `H` and `W` so that the indexing of the row and column within the loops in ⟨*Process pixels using neighbours*⟩ lie in the intervals $[1, H]$ and $[1, W]$ respectively.

⟨*Define constant variables*⟩≡
```
const uint16_t H = height - 1;
const uint16_t W = width - 1;
```

⟨*Process pixels using neighbours*⟩≡
```
for (uint16_t r = 1; r < H; ++r) {
    for (uint16_t c = 1; c < W; ++c) {
        if (V(skeleton, r, c) == 0) {
            continue;
        }
        ⟨Determine pixel mask condition⟩
    }
}
```

⟨*Determine pixel mask condition*⟩≡
⟨*Treat first neighbour as special*⟩
⟨*Count bright pixels and 0-1 transitions*⟩
⟨*Check bright pixels count condition*⟩
⟨*Check 0-1 transition condition*⟩
⟨*Check shape conditions*⟩
⟨*Update mask if pixel satisfies conditions*⟩

The two-dimensional array `ncoords` stores the neighbourhood pixel index offsets, starting with the neighbour above the pixel, and going clockwise for all of the eight neighbours.

⟨*Define constant variables*⟩+≡
```
const int ncoords[8][2] = {
    {0, -1}, {1, -1}, {1, 0}, {1, 1},
    {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}
};
```

⟨*Treat first neighbour as special*⟩≡
```
int x = c + ncoords[0][0];
int y = r + ncoords[0][1];
uint8_t v = V(skeleton, y, x);
uint8_t neighs = v;
```

[1] 2020 January 15, Wednesday, 21:17:27

⟨*Count bright pixels and 0-1 transitions*⟩≡
```
uint8_t prev = v;
int sum = v, num_0_to_1 = 0;
for (int i = 1; i < 8; ++i) {
    x = c + ncoords[i][0];
    y = r + ncoords[i][1];
    v = V(skeleton, y, x);
    if (v) {
        neighs |= 0x1 « i;
        ++sum;
        if (prev == 0) {
            ++num_0_to_1;
        }
    }
    prev = v;
}
```

⟨*Check bright pixels count condition*⟩≡
```
if (sum < 2 || sum > 6) {
    continue;
}
```

⟨*Check 0-1 transition condition*⟩≡
```
if (prev == 0 && (neighs & 0x1)) {
    ++num_0_to_1;
}
if (num_0_to_1 != 1) {
    continue;
}
```

The two-dimensional array `bpat` stores the neighbour pixel patterns required by the algorithm. Since the algorithm works with binary images, we can store the entire neighbourhood in a byte, each bit storing the state of the neighbours: `0` if pixel is black, and `1` if white. Hence, to check a specific pattern, we simply check if the neighbourhood bitmap matches one of the patterns in `bpat`.

⟨*Define constant variables*⟩+≡
```
const uint8_t bpat[2][2] = {
    {0x15 /* 0 & 2 & 4 */, 0x54 /* 2 & 4 & 6 */ },
    {0x45 /* 0 & 2 & 6 */, 0x51 /* 0 & 4 & 6 */ }
};
```

Depending on the step of the loop (there are two steps for each thinning iteration), we check the pattern made by the neighbourhood pixels against the preset bitmaps. Note that there are two patterns to check for each step.

⟨*Check shape conditions*⟩≡
```
if (has_bpat(neighs, bpat[step][0]) ||
    has_bpat(neighs, bpat[step][1])) {
    continue;
}
```

⟨*Update mask if pixel satisfies conditions*⟩≡
```
V(mask, r, c) = 0;
not_done = 1;
```

⟨*Update skeleton using thinning mask*⟩≡
```
if (not_done) {
    uint8_t *s = skeleton;
    uint8_t *m = mask;
    for (size_t i = num_pixels; i; −i) {
        uint8_t v = *s;
        *s++ = v & *m++;
    }
}
```

For simplicity, we provide an ASCII text file that contains the binary image data. The pixel values (either 0 or 1) are listed in row-major form. For instance, in the following, the `binary.txt` file must contain $10,000$ binary values, representing a binary image with 100×100 pixels. The pixel data for the resulting thinned skeleton image is generated the same way.

⟨*Test functions*⟩≡
```
void test_zhang_suen_thinning()
{
    printf("test_zhang_suen_thinning()\n");
    uint8_t binary[100][100];
    uint8_t skeleton[100][100];
    uint8_t mask[397][100];
    uint16_t width = 100;
    uint16_t height = 100;

    FILE *f = fopen("data/binary.txt", "r");
    if (f == NULL) {
       printf("Failed to open binary image file.");
       return;
    }
    for (uint16_t r = 0; r < height; ++r) {
        for (uint16_t c = 0; c < width; ++c) {
            int v = fgetc(f);
            binary[r][c] = v == '0' ? 0 : 1;
        }
    }
    fclose(f);

    zhang_suen_thinning(&skeleton[0][0],
        &binary[0][0], &mask[0][0], width, height);

    f = fopen("result/result.txt", "w");
    for (uint16_t r = 0; r < height; ++r) {
        for (uint16_t c = 0; c < width; ++c) {
            fprintf(f, "%d", skeleton[r][c]);
        }
    }
    fclose(f);
}
```

II. Main program

Main program for testing the functions.

⟨*Main*⟩≡

```
int main(void) {
    test_zhang_suen_thinning();
    return 0;
}
```

⟨*Standard libraries*⟩≡

```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
```

⟨*algorithm.c*⟩≡

⟨*Standard libraries*⟩
⟨*Functions*⟩
⟨*Test functions*⟩
⟨*Main*⟩