ДОДАТОК А

Код програми

```
#include "precomp.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include <limits>

namespace cv
{
   using std::vector;
   class MSER_Impl CV_FINAL : public MSER
   {
   public:
      struct Params
      {
         Params(int _delta = 5, int _min_area = 60, int _max_area = 14400,
            double _max_variation = 0.25, double _min_diversity = .2,
            int _max_evolution = 200, double _area_threshold = 1.01,
            double _min_margin = 0.003, int _edge_blur_size = 5)
         {
            delta = _delta;
            minArea = _min_area;
            maxArea = _max_area;
            maxVariation = _max_variation;
            minDiversity = _min_diversity;
            maxEvolution = _max_evolution;
            areaThreshold = _area_threshold;
            minMargin = _min_margin;
            edgeBlurSize = _edge_blur_size;
            pass2Only = false;
```

```
        }

        int delta;
        int minArea;
        int maxArea;
        double maxVariation;
        double minDiversity;
        bool pass2Only;

        int maxEvolution;
        double areaThreshold;
        double minMargin;
        int edgeBlurSize;
    };

    explicit MSER_Impl(const Params& _params) : params(_params) {}

    virtual ~MSER_Impl() CV_OVERRIDE {}

    void setDelta(int delta) CV_OVERRIDE { params.delta = delta; }
    int getDelta() const CV_OVERRIDE { return params.delta; }

    void setMinArea(int minArea) CV_OVERRIDE { params.minArea = minArea; }
    int getMinArea() const CV_OVERRIDE { return params.minArea; }

    void setMaxArea(int maxArea) CV_OVERRIDE { params.maxArea = maxArea; }
    int getMaxArea() const CV_OVERRIDE { return params.maxArea; }
```

```
        void setPass2Only(bool f) CV_OVERRIDE { params.pass2Only = f; }
        bool getPass2Only() const CV_OVERRIDE { return params.pass2Only; }

        enum { DIR_SHIFT = 29, NEXT_MASK = ((1 << DIR_SHIFT) - 1) };

        struct Pixel
        {
            Pixel() : val(0) {}
            Pixel(int _val) : val(_val) {}

            int getGray(const Pixel* ptr0, const uchar* imgptr0, int mask) const
            {
                return imgptr0[this - ptr0] ^ mask;
            }
            int getNext() const { return (val & NEXT_MASK); }
            void setNext(int next) { val = (val & ~NEXT_MASK) | next; }

            int getDir() const { return (int)((unsigned)val >> DIR_SHIFT); }
            void setDir(int dir) { val = (val & NEXT_MASK) | (dir << DIR_SHIFT); }
            bool isVisited() const { return (val & ~NEXT_MASK) != 0; }

            int val;
        };
        typedef int PPixel;

        struct WParams
        {
            Params p;
            vector<vector<Point> >* msers;
```

```
    vector<Rect>* bboxvec;
    Pixel* pix0;
    int step;
};

// the history of region grown
struct CompHistory
{
    CompHistory()
    {
        parent_ = child_ = next_ = 0;
        val = size = 0;
        var = -1.f;
        head = 0;
        checked = false;
    }
    void updateTree(WParams& wp, CompHistory** _h0, CompHistory**
_h1, bool final)
    {
        if (var >= 0.f)
            return;
        int delta = wp.p.delta;

        CompHistory* h0_ = 0, * h1_ = 0;
        CompHistory* c = child_;
        if (size >= wp.p.minArea)
        {
            for (; c != 0; c = c->next_)
            {
                if (c->var < 0.f)
```

```
                c->updateTree(wp, c == child_ ? &h0_ : 0, c == child_ ? &h1_ : 0, final);
                if (c->var < 0.f)
                    return;
            }
        }

        // find h0 and h1 such that:
        //     h0->val >= h->val - delta and (h0->parent == 0 or h0->parent->val < h->val - delta)
        //    h1->val <= h->val + delta and (h1->child == 0 or h1->child->val < h->val + delta)
        // then we will adjust h0 and h1 as h moves towards latest
        CompHistory* h0 = this, * h1 = h1_ && h1_->size > size ? h1_ : this;
        if (h0_)
        {
            for (h0 = h0_; h0 != this && h0->val < val - delta; h0 = h0->parent_)
                ;
        }
        else
        {
            for (; h0->child_ && h0->child_->val >= val - delta; h0 = h0->child_)
                ;
        }

        for (; h1->parent_ && h1->parent_->val <= val + delta; h1 = h1->parent_)
```

```
            ;

        if (_h0) *_h0 = h0;
        if (_h1) *_h1 = h1;

        // when we do not well-defined ER(h->val + delta), we stop
        // the process of computing variances unless we are at the final step
        if (!final && !h1->parent_ && h1->val < val + delta)
            return;

        var = (float)(h1->size - h0->size) / size;
        c = child_;
        for (; c != 0; c = c->next_)
            c->checkAndCapture(wp);
        if (final && !parent_)
            checkAndCapture(wp);
    }

    void checkAndCapture(WParams& wp)
    {
        if (checked)
            return;
        checked = true;
        if (size < wp.p.minArea || size > wp.p.maxArea || var < 0.f || var >
wp.p.maxVariation)
            return;
        if (child_)
        {
            CompHistory* c = child_;
            for (; c != 0; c = c->next_)
```

```
                {
                    if (c->var >= 0.f && var > c->var)
                        return;
                }
            }
            if (var > 0.f && parent_ && parent_->var >= 0.f && var >=
parent_->var)
                return;
            int xmin = INT_MAX, ymin = INT_MAX, xmax = INT_MIN, ymax
= INT_MIN, j = 0;
            wp.msers->push_back(vector<Point>());
            vector<Point>& region = wp.msers->back();
            region.resize(size);
            const Pixel* pix0 = wp.pix0;
            int step = wp.step;

            for (PPixel pix = head; j < size; j++, pix = pix0[pix].getNext())
            {
                int y = pix / step;
                int x = pix - y * step;

                xmin = std::min(xmin, x);
                xmax = std::max(xmax, x);
                ymin = std::min(ymin, y);
                ymax = std::max(ymax, y);

                region[j] = Point(x, y);
            }

            wp.bboxvec->push_back(Rect(xmin, ymin, xmax - xmin + 1, ymax -
```

```
ymin + 1));
        }

        CompHistory* child_;
        CompHistory* parent_;
        CompHistory* next_;
        int val;
        int size;
        float var;
        PPixel head;
        bool checked;
    };

    struct ConnectedComp
    {
        ConnectedComp()
        {
            init(0);
        }

        void init(int gray)
        {
            head = tail = 0;
            history = 0;
            size = 0;
            gray_level = gray;
        }

        // add history chunk to a connected component
        void growHistory(CompHistory*& hptr, WParams& wp, int
```

```
new_gray_level, bool final)
                {
                   if (new_gray_level < gray_level)
                      new_gray_level = gray_level;

                   CompHistory* h;
                   if (history && history->val == gray_level)
                   {
                      h = history;
                   }
                   else
                   {
                      h = hptr++;
                      h->parent_ = 0;
                      h->child_ = history;
                      h->next_ = 0;

                      if (history)
                      {
                         history->parent_ = h;
                      }
                   }
                   CV_Assert(h != NULL);
                   h->val = gray_level;
                   h->size = size;
                   h->head = head;
                   h->var = FLT_MAX;
                   h->checked = true;
                   if (h->size >= wp.p.minArea)
                   {
```

```
        h->var = -1.f;
        h->checked = false;
    }

    gray_level = new_gray_level;
    history = h;
    if (history && history->val != gray_level)
    {
        history->updateTree(wp, 0, 0, final);
    }
}

// merging two connected components
void merge(ConnectedComp* comp1, ConnectedComp* comp2,
    CompHistory*& hptr, WParams& wp)
{
    if (comp1->gray_level < comp2->gray_level)
        std::swap(comp1, comp2);

    gray_level = comp1->gray_level;
    comp1->growHistory(hptr, wp, gray_level, false);
    comp2->growHistory(hptr, wp, gray_level, false);

    if (comp1->size == 0)
    {
        head = comp2->head;
        tail = comp2->tail;
    }
    else
    {
```

```
    head = comp1->head;
    wp.pix0[comp1->tail].setNext(comp2->head);
    tail = comp2->tail;
}

size = comp1->size + comp2->size;
history = comp1->history;

CompHistory* h1 = history->child_;
CompHistory* h2 = comp2->history;
// the child_'s size should be the large one
if (h1 && h1->size > h2->size)
{
    // add h2 as a child only if its size is large enough
    if (h2->size >= wp.p.minArea)
    {
        h2->next_ = h1->next_;
        h1->next_ = h2;
        h2->parent_ = history;
    }
}
else
{
    history->child_ = h2;
    h2->parent_ = history;
    // reserve h1 as a child only if its size is large enough
    if (h1 && h1->size >= wp.p.minArea)
    {
        h2->next_ = h1;
    }
```

```
        }
    }

    PPixel head;
    PPixel tail;
    CompHistory* history;
    int gray_level;
    int size;
};

void detectRegions(InputArray image,
    std::vector<std::vector<Point> >& msers,
    std::vector<Rect>& bboxes) CV_OVERRIDE;
void detect(InputArray _src, vector<KeyPoint>& keypoints, InputArray
_mask) CV_OVERRIDE;

void preprocess1(const Mat& img, int* level_size)
{
    memset(level_size, 0, 256 * sizeof(level_size[0]));

    int i, j, cols = img.cols, rows = img.rows;
    int step = cols;
    pixbuf.resize(step * rows);
    heapbuf.resize(cols * rows + 256);
    histbuf.resize(cols * rows);
    Pixel borderpix;
    borderpix.setDir(5);

    for (j = 0; j < step; j++)
    {
```

```
        pixbuf[j] = pixbuf[j + (rows - 1) * step] = borderpix;
    }

    for (i = 1; i < rows - 1; i++)
    {
        const uchar* imgptr = img.ptr(i);
        Pixel* pptr = &pixbuf[i * step];
        pptr[0] = pptr[cols - 1] = borderpix;
        for (j = 1; j < cols - 1; j++)
        {
            int val = imgptr[j];
            level_size[val]++;
            pptr[j].val = 0;
        }
    }
}

void preprocess2(const Mat& img, int* level_size)
{
    int i;

    for (i = 0; i < 128; i++)
        std::swap(level_size[i], level_size[255 - i]);

    if (!params.pass2Only)
    {
        int j, cols = img.cols, rows = img.rows;
        int step = cols;
        for (i = 1; i < rows - 1; i++)
        {
```

```
            Pixel* pptr = &pixbuf[i * step];
            for (j = 1; j < cols - 1; j++)
            {
               pptr[j].val = 0;
            }
         }
      }
   }

   void pass(const Mat& img, vector<vector<Point> >& msers, vector<Rect>& bboxvec,
      Size size, const int* level_size, int mask)
   {
      CompHistory* histptr = &histbuf[0];
      int step = size.width;
      Pixel* ptr0 = &pixbuf[0], * ptr = &ptr0[step + 1];
      const uchar* imgptr0 = img.ptr();
      Pixel** heap[256];
      ConnectedComp comp[257];
      ConnectedComp* comptr = &comp[0];
      WParams wp;
      wp.p = params;
      wp.msers = &msers;
      wp.bboxvec = &bboxvec;
      wp.pix0 = ptr0;
      wp.step = step;

      heap[0] = &heapbuf[0];
      heap[0][0] = 0;
```

```
for (int i = 1; i < 256; i++)
{
   heap[i] = heap[i - 1] + level_size[i - 1] + 1;
   heap[i][0] = 0;
}

comptr->gray_level = 256;
comptr++;
comptr->gray_level = ptr->getGray(ptr0, imgptr0, mask);
ptr->setDir(1);
int dir[] = { 0, 1, step, -1, -step };
for (;; )
{
   int curr_gray = ptr->getGray(ptr0, imgptr0, mask);
   int nbr_idx = ptr->getDir();
   // take tour of all the 4 directions
   for (; nbr_idx <= 4; nbr_idx++)
   {
      // get the neighbor
      Pixel* ptr_nbr = ptr + dir[nbr_idx];
      if (!ptr_nbr->isVisited())
      {
         // set dir=1, next=0
         ptr_nbr->val = 1 << DIR_SHIFT;
         int nbr_gray = ptr_nbr->getGray(ptr0, imgptr0, mask);
         if (nbr_gray < curr_gray)
         {
            // when the value of neighbor smaller than current
            // push current to boundary heap and make the neighbor to be
the current one
```

```
            // create an empty comp
            *(++heap[curr_gray]) = ptr;
            ptr->val = (nbr_idx + 1) << DIR_SHIFT;
            ptr = ptr_nbr;
            comptr++;
            comptr->init(nbr_gray);
            curr_gray = nbr_gray;
            nbr_idx = 0;
            continue;
        }
        // otherwise, push the neighbor to boundary heap
        *(++heap[nbr_gray]) = ptr_nbr;
    }
}

// set dir = nbr_idx, next = 0
ptr->val = nbr_idx << DIR_SHIFT;
int ptrofs = (int)(ptr - ptr0);
CV_Assert(ptrofs != 0);

// add a pixel to the pixel list
if (comptr->tail)
    ptr0[comptr->tail].setNext(ptrofs);
else
    comptr->head = ptrofs;
comptr->tail = ptrofs;
comptr->size++;
// get the next pixel from boundary heap
if (*heap[curr_gray])
{
```

```
            ptr = *heap[curr_gray];
            heap[curr_gray]--;
        }
        else
        {
            for (curr_gray++; curr_gray < 256; curr_gray++)
            {
                if (*heap[curr_gray])
                    break;
            }
            if (curr_gray >= 256)
                break;

            ptr = *heap[curr_gray];
            heap[curr_gray]--;

            if (curr_gray < comptr[-1].gray_level)
            {
                comptr->growHistory(histptr, wp, curr_gray, false);
                CV_DbgAssert(comptr->size == comptr->history->size);
            }
            else
            {
                // there must one pixel with the second component's gray level in the heap,
                // so curr_gray is not large than the second component's gray level
                comptr--;
                CV_DbgAssert(curr_gray == comptr->gray_level);
                comptr->merge(comptr, comptr + 1, histptr, wp);
```

```
                    CV_DbgAssert(curr_gray == comptr->gray_level);
                }
            }
        }

        for (; comptr->gray_level != 256; comptr--)
        {
            comptr->growHistory(histptr, wp, 256, true);
        }
    }

    Mat tempsrc;
    vector<Pixel> pixbuf;
    vector<Pixel*> heapbuf;
    vector<CompHistory> histbuf;

    Params params;
};

/*
TODO:
the color MSER has not been completely refactored yet. We leave it mostly as-is,
with just enough changes to convert C structures to C++ ones and
add support for color images into MSER_Impl::detectAndLabel.
*/

const int TABLE_SIZE = 400;

static const float chitab3[] =
```

```
{
    0.f,  0.0150057f,  0.0239478f,  0.0315227f,
    0.0383427f,  0.0446605f,  0.0506115f,  0.0562786f,
    0.0617174f,  0.0669672f,  0.0720573f,  0.0770099f,
    0.081843f,  0.0865705f,  0.0912043f,  0.0957541f,
    0.100228f,  0.104633f,  0.108976f,  0.113261f,
    0.117493f,  0.121676f,  0.125814f,  0.12991f,
    0.133967f,  0.137987f,  0.141974f,  0.145929f,
    0.149853f,  0.15375f,  0.15762f,  0.161466f,
    0.165287f,  0.169087f,  0.172866f,  0.176625f,
    0.180365f,  0.184088f,  0.187794f,  0.191483f,
    0.195158f,  0.198819f,  0.202466f,  0.2061f,
    0.209722f,  0.213332f,  0.216932f,  0.220521f,
    0.2241f,  0.22767f,  0.231231f,  0.234783f,
    0.238328f,  0.241865f,  0.245395f,  0.248918f,
    0.252435f,  0.255947f,  0.259452f,  0.262952f,
    0.266448f,  0.269939f,  0.273425f,  0.276908f,
    0.280386f,  0.283862f,  0.287334f,  0.290803f,
    0.29427f,  0.297734f,  0.301197f,  0.304657f,
    0.308115f,  0.311573f,  0.315028f,  0.318483f,
    0.321937f,  0.32539f,  0.328843f,  0.332296f,
    0.335749f,  0.339201f,  0.342654f,  0.346108f,
    0.349562f,  0.353017f,  0.356473f,  0.35993f,
    0.363389f,  0.366849f,  0.37031f,  0.373774f,
    0.377239f,  0.380706f,  0.384176f,  0.387648f,
    0.391123f,  0.3946f,  0.39808f,  0.401563f,
    0.405049f,  0.408539f,  0.412032f,  0.415528f,
    0.419028f,  0.422531f,  0.426039f,  0.429551f,
    0.433066f,  0.436586f,  0.440111f,  0.44364f,
    0.447173f,  0.450712f,  0.454255f,  0.457803f,
```

0.461356f, 0.464915f, 0.468479f, 0.472049f,
0.475624f, 0.479205f, 0.482792f, 0.486384f,
0.489983f, 0.493588f, 0.4972f, 0.500818f,
0.504442f, 0.508073f, 0.511711f, 0.515356f,
0.519008f, 0.522667f, 0.526334f, 0.530008f,
0.533689f, 0.537378f, 0.541075f, 0.54478f,
0.548492f, 0.552213f, 0.555942f, 0.55968f,
0.563425f, 0.56718f, 0.570943f, 0.574715f,
0.578497f, 0.582287f, 0.586086f, 0.589895f,
0.593713f, 0.597541f, 0.601379f, 0.605227f,
0.609084f, 0.612952f, 0.61683f, 0.620718f,
0.624617f, 0.628526f, 0.632447f, 0.636378f,
0.64032f, 0.644274f, 0.648239f, 0.652215f,
0.656203f, 0.660203f, 0.664215f, 0.668238f,
0.672274f, 0.676323f, 0.680384f, 0.684457f,
0.688543f, 0.692643f, 0.696755f, 0.700881f,
0.70502f, 0.709172f, 0.713339f, 0.717519f,
0.721714f, 0.725922f, 0.730145f, 0.734383f,
0.738636f, 0.742903f, 0.747185f, 0.751483f,
0.755796f, 0.760125f, 0.76447f, 0.768831f,
0.773208f, 0.777601f, 0.782011f, 0.786438f,
0.790882f, 0.795343f, 0.799821f, 0.804318f,
0.808831f, 0.813363f, 0.817913f, 0.822482f,
0.827069f, 0.831676f, 0.836301f, 0.840946f,
0.84561f, 0.850295f, 0.854999f, 0.859724f,
0.864469f, 0.869235f, 0.874022f, 0.878831f,
0.883661f, 0.888513f, 0.893387f, 0.898284f,
0.903204f, 0.908146f, 0.913112f, 0.918101f,
0.923114f, 0.928152f, 0.933214f, 0.938301f,
0.943413f, 0.94855f, 0.953713f, 0.958903f,

0.964119f, 0.969361f, 0.974631f, 0.979929f,
0.985254f, 0.990608f, 0.99599f, 1.0014f,
1.00684f, 1.01231f, 1.01781f, 1.02335f,
1.02891f, 1.0345f, 1.04013f, 1.04579f,
1.05148f, 1.05721f, 1.06296f, 1.06876f,
1.07459f, 1.08045f, 1.08635f, 1.09228f,
1.09826f, 1.10427f, 1.11032f, 1.1164f,
1.12253f, 1.1287f, 1.1349f, 1.14115f,
1.14744f, 1.15377f, 1.16015f, 1.16656f,
1.17303f, 1.17954f, 1.18609f, 1.19269f,
1.19934f, 1.20603f, 1.21278f, 1.21958f,
1.22642f, 1.23332f, 1.24027f, 1.24727f,
1.25433f, 1.26144f, 1.26861f, 1.27584f,
1.28312f, 1.29047f, 1.29787f, 1.30534f,
1.31287f, 1.32046f, 1.32812f, 1.33585f,
1.34364f, 1.3515f, 1.35943f, 1.36744f,
1.37551f, 1.38367f, 1.39189f, 1.4002f,
1.40859f, 1.41705f, 1.42561f, 1.43424f,
1.44296f, 1.45177f, 1.46068f, 1.46967f,
1.47876f, 1.48795f, 1.49723f, 1.50662f,
1.51611f, 1.52571f, 1.53541f, 1.54523f,
1.55517f, 1.56522f, 1.57539f, 1.58568f,
1.59611f, 1.60666f, 1.61735f, 1.62817f,
1.63914f, 1.65025f, 1.66152f, 1.67293f,
1.68451f, 1.69625f, 1.70815f, 1.72023f,
1.73249f, 1.74494f, 1.75757f, 1.77041f,
1.78344f, 1.79669f, 1.81016f, 1.82385f,
1.83777f, 1.85194f, 1.86635f, 1.88103f,
1.89598f, 1.91121f, 1.92674f, 1.94257f,
1.95871f, 1.97519f, 1.99201f, 2.0092f,

```
    2.02676f,  2.04471f,  2.06309f,  2.08189f,
    2.10115f,  2.12089f,  2.14114f,  2.16192f,
    2.18326f,  2.2052f,  2.22777f,  2.25101f,
    2.27496f,  2.29966f,  2.32518f,  2.35156f,
    2.37886f,  2.40717f,  2.43655f,  2.46709f,
    2.49889f,  2.53206f,  2.56673f,  2.60305f,
    2.64117f,  2.6813f,  2.72367f,  2.76854f,
    2.81623f,  2.86714f,  2.92173f,  2.98059f,
    3.04446f,  3.1143f,  3.19135f,  3.27731f,
    3.37455f,  3.48653f,  3.61862f,  3.77982f,
    3.98692f,  4.2776f,  4.77167f,  133.333f
};

struct MSCRNode;

struct TempMSCR
{
    MSCRNode* head;
    MSCRNode* tail;
    double m; // the margin used to prune area later
    int size;
};

struct MSCRNode
{
    MSCRNode* shortcut;
    // to make the finding of root less painful
    MSCRNode* prev;
    MSCRNode* next;
    // a point double-linked list
```

```
    TempMSCR* tmsr;
    // the temporary msr (set to NULL at every re-initialise)
    TempMSCR* gmsr;
    // the global msr (once set, never to NULL)
    int index;
    // the index of the node, at this point, it should be x at the first 16-bits, and y at the last 16-bits.
    int rank;
    int reinit;
    int size, sizei;
    double dt, di;
    double s;
};

struct MSCREdge
{
    double chi;
    MSCRNode* left;
    MSCRNode* right;
};

static double ChiSquaredDistance(const uchar* x, const uchar* y)
{
    return (double)((x[0] - y[0]) * (x[0] - y[0])) / (double)(x[0] + y[0] + 1e-10) +
        (double)((x[1] - y[1]) * (x[1] - y[1])) / (double)(x[1] + y[1] + 1e-10) +
        (double)((x[2] - y[2]) * (x[2] - y[2])) / (double)(x[2] + y[2] + 1e-10);
}

static void initMSCRNode(MSCRNode* node)
```

```
{
    node->gmsr = node->tmsr = NULL;
    node->reinit = 0xffff;
    node->rank = 0;
    node->sizei = node->size = 1;
    node->prev = node->next = node->shortcut = node;
}

// the preprocess to get the edge list with proper gaussian blur
static int preprocessMSER_8uC3(MSCRNode* node,
    MSCREdge* edge,
    double* total,
    const Mat& src,
    Mat& dx,
    Mat& dy,
    int Ne,
    int edgeBlurSize)
{
    int srccpt = (int)(src.step - src.cols * 3);
    const uchar* srcptr = src.ptr();
    const uchar* lastptr = srcptr + 3;
    double* dxptr = dx.ptr<double>();
    for (int i = 0; i < src.rows; i++)
    {
        for (int j = 0; j < src.cols - 1; j++)
        {
            *dxptr = ChiSquaredDistance(srcptr, lastptr);
            dxptr++;
            srcptr += 3;
            lastptr += 3;
```

```
    }
    srcptr += srccpt + 3;
    lastptr += srccpt + 3;
}
srcptr = src.ptr();
lastptr = srcptr + src.step;
double* dyptr = dy.ptr<double>();
for (int i = 0; i < src.rows - 1; i++)
{
    for (int j = 0; j < src.cols; j++)
    {
        *dyptr = ChiSquaredDistance(srcptr, lastptr);
        dyptr++;
        srcptr += 3;
        lastptr += 3;
    }
    srcptr += srccpt;
    lastptr += srccpt;
}
// get dx and dy and blur it
if (edgeBlurSize >= 1)
{
    GaussianBlur(dx, dx, Size(edgeBlurSize, edgeBlurSize), 0);
    GaussianBlur(dy, dy, Size(edgeBlurSize, edgeBlurSize), 0);
}
dxptr = dx.ptr<double>();
dyptr = dy.ptr<double>();
// assian dx, dy to proper edge list and initialize mscr node
// the nasty code here intended to avoid extra loops
MSCRNode* nodeptr = node;
```

```
initMSCRNode(nodeptr);
nodeptr->index = 0;
*total += edge->chi = *dxptr;
dxptr++;
edge->left = nodeptr;
edge->right = nodeptr + 1;
edge++;
nodeptr++;
for (int i = 1; i < src.cols - 1; i++)
{
   initMSCRNode(nodeptr);
   nodeptr->index = i;
   *total += edge->chi = *dxptr;
   dxptr++;
   edge->left = nodeptr;
   edge->right = nodeptr + 1;
   edge++;
   nodeptr++;
}
initMSCRNode(nodeptr);
nodeptr->index = src.cols - 1;
nodeptr++;
for (int i = 1; i < src.rows - 1; i++)
{
   initMSCRNode(nodeptr);
   nodeptr->index = i << 16;
   *total += edge->chi = *dyptr;
   dyptr++;
   edge->left = nodeptr - src.cols;
   edge->right = nodeptr;
```

```
edge++;
*total += edge->chi = *dxptr;
dxptr++;
edge->left = nodeptr;
edge->right = nodeptr + 1;
edge++;
nodeptr++;
for (int j = 1; j < src.cols - 1; j++)
{
   initMSCRNode(nodeptr);
   nodeptr->index = (i << 16) | j;
   *total += edge->chi = *dyptr;
   dyptr++;
   edge->left = nodeptr - src.cols;
   edge->right = nodeptr;
   edge++;
   *total += edge->chi = *dxptr;
   dxptr++;
   edge->left = nodeptr;
   edge->right = nodeptr + 1;
   edge++;
   nodeptr++;
}
initMSCRNode(nodeptr);
nodeptr->index = (i << 16) | (src.cols - 1);
*total += edge->chi = *dyptr;
dyptr++;
edge->left = nodeptr - src.cols;
edge->right = nodeptr;
edge++;
```

```
    nodeptr++;
}
initMSCRNode(nodeptr);
nodeptr->index = (src.rows - 1) << 16;
*total += edge->chi = *dxptr;
dxptr++;
edge->left = nodeptr;
edge->right = nodeptr + 1;
edge++;
*total += edge->chi = *dyptr;
dyptr++;
edge->left = nodeptr - src.cols;
edge->right = nodeptr;
edge++;
nodeptr++;
for (int i = 1; i < src.cols - 1; i++)
{
    initMSCRNode(nodeptr);
    nodeptr->index = ((src.rows - 1) << 16) | i;
    *total += edge->chi = *dxptr;
    dxptr++;
    edge->left = nodeptr;
    edge->right = nodeptr + 1;
    edge++;
    *total += edge->chi = *dyptr;
    dyptr++;
    edge->left = nodeptr - src.cols;
    edge->right = nodeptr;
    edge++;
    nodeptr++;
```

```
            }
            initMSCRNode(nodeptr);
            nodeptr->index = ((src.rows - 1) << 16) | (src.cols - 1);
            *total += edge->chi = *dyptr;
            edge->left = nodeptr - src.cols;
            edge->right = nodeptr;

            return Ne;
        }

        class LessThanEdge
        {
        public:
            bool operator()(const MSCREdge& a, const MSCREdge& b) const { return a.chi < b.chi; }
        };

        // to find the root of one region
        static MSCRNode* findMSCR(MSCRNode* x)
        {
            MSCRNode* prev = x;
            MSCRNode* next;
            for (; ; )
            {
                next = x->shortcut;
                x->shortcut = prev;
                if (next == x) break;
                prev = x;
                x = next;
            }
```

```
        MSCRNode* root = x;
        for (; ; )
        {
            prev = x->shortcut;
            x->shortcut = root;
            if (prev == x) break;
            x = prev;
        }
        return root;
    }

    // the stable mscr should be:
    // bigger than minArea and smaller than maxArea
    // differ from its ancestor more than minDiversity
    static bool MSCRStableCheck(MSCRNode* x, const MSER_Impl::Params& params)
    {
        if (x->size <= params.minArea || x->size >= params.maxArea)
            return false;
        if (x->gmsr == NULL)
            return true;
        double div = (double)(x->size - x->gmsr->size) / (double)x->size;
        return div > params.minDiversity;
    }

    static void
        extractMSER_8uC3(const Mat& src,
            vector<vector<Point> >& msers,
            vector<Rect>& bboxvec,
            const MSER_Impl::Params& params)
```

```
        {
            bboxvec.clear();
            MSCRNode* map = (MSCRNode*)cvAlloc(src.cols * src.rows * sizeof(map[0]));
            int Ne = src.cols * src.rows * 2 - src.cols - src.rows;
            MSCREdge* edge = (MSCREdge*)cvAlloc(Ne * sizeof(edge[0]));
            TempMSCR* mscr = (TempMSCR*)cvAlloc(src.cols * src.rows * sizeof(mscr[0]));
            double emean = 0;
            Mat dx(src.rows, src.cols - 1, CV_64FC1);
            Mat dy(src.rows - 1, src.cols, CV_64FC1);
            Ne = preprocessMSER_8uC3(map, edge, &emean, src, dx, dy, Ne, params.edgeBlurSize);
            emean = emean / (double)Ne;
            std::sort(edge, edge + Ne, LessThanEdge());
            MSCREdge* edge_ub = edge + Ne;
            MSCREdge* edgeptr = edge;
            TempMSCR* mscrptr = mscr;
            // the evolution process
            for (int i = 0; i < params.maxEvolution; i++)
            {
                double k = (double)i / (double)params.maxEvolution * (TABLE_SIZE - 1);
                int ti = cvFloor(k);
                double reminder = k - ti;
                double thres = emean * (chitab3[ti] * (1 - reminder) + chitab3[ti + 1] * reminder);
                // to process all the edges in the list that chi < thres
                while (edgeptr < edge_ub && edgeptr->chi < thres)
                {
```

```
MSCRNode* lr = findMSCR(edgeptr->left);
MSCRNode* rr = findMSCR(edgeptr->right);
// get the region root (who is responsible)
if (lr != rr)
{
   // rank idea take from: N-tree Disjoint-Set Forests for Maximally Stable Extremal Regions
   if (rr->rank > lr->rank)
   {
      MSCRNode* tmp;
      CV_SWAP(lr, rr, tmp);
   }
   else if (lr->rank == rr->rank) {
      // at the same rank, we will compare the size
      if (lr->size > rr->size)
      {
         MSCRNode* tmp;
         CV_SWAP(lr, rr, tmp);
      }
      lr->rank++;
   }
   rr->shortcut = lr;
   lr->size += rr->size;
   // join rr to the end of list lr (lr is a endless double-linked list)
   lr->prev->next = rr;
   lr->prev = rr->prev;
   rr->prev->next = lr;
   rr->prev = lr;
   // area threshold force to reinitialize
   if (lr->size > (lr->size - rr->size) * params.areaThreshold)
```

```
{
    lr->sizei = lr->size;
    lr->reinit = i;
    if (lr->tmsr != NULL)
    {
        lr->tmsr->m = lr->dt - lr->di;
        lr->tmsr = NULL;
    }
    lr->di = edgeptr->chi;
    lr->s = 1e10;
}
lr->dt = edgeptr->chi;
if (i > lr->reinit)
{
    double s = (double)(lr->size - lr->sizei) / (lr->dt - lr->di);
    if (s < lr->s)
    {
        // skip the first one and check stability
        if (i > lr->reinit + 1 && MSCRStableCheck(lr, params))
        {
            if (lr->tmsr == NULL)
            {
                lr->gmsr = lr->tmsr = mscrptr;
                mscrptr++;
            }
            lr->tmsr->size = lr->size;
            lr->tmsr->head = lr;
            lr->tmsr->tail = lr->prev;
            lr->tmsr->m = 0;
        }
```

```
                        lr->s = s;
                    }
                }
            }
            edgeptr++;
        }
        if (edgeptr >= edge_ub)
            break;
    }
    for (TempMSCR* ptr = mscr; ptr < mscrptr; ptr++)
        // to prune area with margin less than minMargin
        if (ptr->m > params.minMargin)
        {
            MSCRNode* lpt = ptr->head;
            int xmin = INT_MAX, ymin = INT_MAX, xmax = INT_MIN, ymax
= INT_MIN;
            msers.push_back(vector<Point>());
            vector<Point>& mser = msers.back();

            for (int i = 0; i < ptr->size; i++)
            {
                Point pt;
                pt.x = (lpt->index) & 0xffff;
                pt.y = (lpt->index) >> 16;
                xmin = std::min(xmin, pt.x);
                xmax = std::max(xmax, pt.x);
                ymin = std::min(ymin, pt.y);
                ymax = std::max(ymax, pt.y);

                lpt = lpt->next;
```

```
                    mser.push_back(pt);
                }
                bboxvec.push_back(Rect(xmin, ymin, xmax - xmin + 1, ymax - ymin
+ 1));
            }
        cvFree(&mscr);
        cvFree(&edge);
        cvFree(&map);
    }

    void MSER_Impl::detectRegions(InputArray _src, vector<vector<Point>
>& msers, vector<Rect>& bboxes)
    {
        CV_INSTRUMENT_REGION();

        Mat src = _src.getMat();

        msers.clear();
        bboxes.clear();

        if (src.rows < 3 || src.cols < 3)
            CV_Error(Error::StsBadArg, "Input image is too small. Expected at
least 3x3");

        Size size = src.size();

        if (src.type() == CV_8U)
        {
            int level_size[256];
            if (!src.isContinuous())
```

```
            {
                src.copyTo(tempsrc);
                src = tempsrc;
            }

            // darker to brighter (MSER+)
            preprocess1(src, level_size);
            if (!params.pass2Only)
                pass(src, msers, bboxes, size, level_size, 0);
            // brighter to darker (MSER-)
            preprocess2(src, level_size);
            pass(src, msers, bboxes, size, level_size, 255);
        }
        else
        {
            CV_Assert(src.type() == CV_8UC3 || src.type() == CV_8UC4);
            extractMSER_8uC3(src, msers, bboxes, params);
        }
    }

    void MSER_Impl::detect(InputArray _image, vector<KeyPoint>&
keypoints, InputArray _mask)
    {
      CV_INSTRUMENT_REGION();
        vector<Rect> bboxes;
        vector<vector<Point> > msers;
        Mat mask = _mask.getMat();
        detectRegions(_image, msers, bboxes);
        int i, ncomps = (int)msers.size();
        keypoints.clear();
```

```
            for (i = 0; i < ncomps; i++)
            {
                Rect r = bboxes[i];
                // TODO check transformation from MSER region to KeyPoint
                RotatedRect rect = fitEllipse(Mat(msers[i]));
                float diam = std::sqrt(rect.size.height * rect.size.width);
                if (diam > std::numeric_limits<float>::epsilon() &&
r.contains(rect.center) &&
                    (mask.empty() || mask.at<uchar>(cvRound(rect.center.y),
cvRound(rect.center.x)) != 0))
                    keypoints.push_back(KeyPoint(rect.center, diam));
            }
        }
        Ptr<MSER> MSER::create(int _delta, int _min_area, int _max_area,
            double _max_variation, double _min_diversity,
            int _max_evolution, double _area_threshold,
            double _min_margin, int _edge_blur_size)
        {
            return makePtr<MSER_Impl>(
                MSER_Impl::Params(_delta, _min_area, _max_area,
                    _max_variation, _min_diversity,
                    _max_evolution, _area_threshold,
                    _min_margin, _edge_blur_size));
        }
        String MSER::getDefaultName() const
        {
            return (Feature2D::getDefaultName() + ".MSER");
        }

    }
```

ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ

Позначення	Найменування	Дод. відомості
	Текстові документи	
1	Пояснювальна записка	106 с.
2	Презентаційний матеріал	23 с.
	Інші документи	
3	Роздруківки програм	36 с.
4	Рецензія	2 с.
5	Відгук керівника	1 с.

					Математичні моделі та методи визначення стану бетонних колекторів за цифровим зображенням			
Змін	**Арк.**	**Номер докум.**	**Підп.**	**Дата**				
Розроб.		**Фенько Д.С.**			**(Тема роботи) Відомість атестаційної роботи**		**Аркуш**	**Аркушів**
Перевір.		**Єсілевський В.С.**						
Н. контр.		**Сидоров М.В.**				**ХНУРЕ кафедра ПМ**		
Затв.		**Тевяшев А.Д.**						