**109590004 呂育瑋** Machine Vision HW#2

這次作業實作影像label處理，對圖像做灰階化與二值化，其code與上次作業相同

這次我以class來建構整個過程：

得到二值化圖像後，對其進行噪點去除與填補空洞，各圖像皆有不同的處理方式，基於Dilation與Erosion組合與次數，我以卷積3x3方式判斷，最終得出以物件邊緣擴增或縮減的二值化圖片。

// 二值化填洞與去雜訊處理的四種方法

void ReconstructBinaryImage(const string& restructMode, const int& restructArg1, const int& restructArg2) {

if (restructMode == "opening") {

ErosionBinaryImage(restructArg1);

DilationBinaryImage(restructArg2);

}

else if (restructMode == "closing") {

DilationBinaryImage(restructArg1);

ErosionBinaryImage(restructArg2);

}

else if (restructMode == "dilation") {

DilationBinaryImage(restructArg1);

}

else if (restructMode == "erosion") {

ErosionBinaryImage(restructArg1);

}

imshow(\_name + " binary", \_binaryImage);

imwrite(BINARY\_FOLDER + \_name, \_binaryImage);

}

// 對二值化圖像侵蝕 iteration 次

void ErosionBinaryImage(int iteration) {

uchar\* binaryPtr;

for (int repeat = 0; repeat < iteration; ++repeat) {

InitLabelingData();

for (int row = 0; row < \_binaryImage.rows; ++row) {

binaryPtr = \_binaryImage.ptr<uchar>(row);

for (int col = 0; col < \_binaryImage.cols; ++col) {

if (\_labelVector.at(LabelVectorIndex(row, col)) == MARK\_WHITE) {

ConvolutionReconstruct(row, col, 255);

}

}

}

}

}

// 對二值化圖像膨脹 iteration 次

void DilationBinaryImage(int iteration) {

uchar\* binaryPtr;

for (int repeat = 0; repeat < iteration; ++repeat) {

InitLabelingData();

for (int row = 0; row < \_binaryImage.rows; ++row) {

binaryPtr = \_binaryImage.ptr<uchar>(row);

for (int col = 0; col < \_binaryImage.cols; ++col) {

if (\_labelVector.at(LabelVectorIndex(row, col)) == MARK\_BLACK) {

ConvolutionReconstruct(row, col, 0);

}

}

}

}

}

// 用3x3卷積判斷是否修改鄰近的像素點

void ConvolutionReconstruct(int row, int col, int value) {

int labelMark = (value == 0) ? MARK\_BLACK : MARK\_WHITE;

for (int neighborRow = -1; neighborRow <= 1; ++neighborRow) {

for (int neighborCol = -1; neighborCol <= 1; ++neighborCol) {

int nRow = row + neighborRow, nCol = col + neighborCol;

if (nRow >= 0 && nRow < \_binaryImage.rows && nCol >= 0 && nCol < \_binaryImage.cols) {

if (\_labelVector.at(LabelVectorIndex(nRow, nCol)) != labelMark) {

\_binaryImage.at<uchar>(nRow, nCol) = value;

\_labelVector.at(LabelVectorIndex(nRow, nCol)) == EXPAND\_MARK;

}

}

}

}

}

得到較為理想的二值化圖片後，再來要進行連通判斷：

在四連通方法實作上，我以給定的像素點，往上與左查看這兩值的關係，進而做出對應label處理方式。

如果上與左都沒有label：給予新的label值

其中之一有label：給予有label的值  
都有label並且label相同：給予左邊的label值

都有label但label不相同：給予左邊的label值，並更新label vector所有label為上方的值更新為左邊的值，更新方式為遍歷所有值做更新。

// 依 4 連通規則設定label相關資料

void LabelPixelBy4Neighbor(const int& row, const int& col, int& labelNumber) {

int labelTop, labelLeft;

labelTop = (row > 0) ? \_labelVector.at(LabelVectorIndex(row - 1, col)) : 0;

labelLeft = (col > 0) ? \_labelVector.at(LabelVectorIndex(row, col - 1)) : 0;

if (labelTop == 0 && labelLeft == 0) {

\_labelVector.at(LabelVectorIndex(row, col)) = labelNumber;

\_labelSet.insert(labelNumber);

++labelNumber;

}

else if (labelTop == 0 && labelLeft > 0) \_labelVector.at(LabelVectorIndex(row, col)) = labelLeft;

else if (labelTop > 0 && labelLeft == 0) \_labelVector.at(LabelVectorIndex(row, col)) = labelTop;

else {

\_labelVector.at(LabelVectorIndex(row, col)) = labelLeft;

if (labelTop != labelLeft) {

for (int labelIndex = 0; labelIndex < \_binaryImage.rows \* \_binaryImage.cols; ++labelIndex) {

if (\_labelVector.at(labelIndex) == labelTop) \_labelVector.at(labelIndex) = labelLeft;

}

\_labelSet.erase(labelTop);

}

}

}

在八連通方法實作上，與四連通類似，以該像素點的左上、上、右上、左，這四個點做如四連通的判斷：

如果全都是0：給予新的label值

只有1種label：給予唯一1種label的值

超過1種label：將所有label併成同一種label，更新方式為遍歷所有值做更新。

// 依 8 連通規則設定label相關資料

void LabelPixelBy8Neighbor(const int& row, const int& col, int& labelNumber) {

vector<int> neighborLabelSet = GetNeighborLabelBy8(row, col);

if (neighborLabelSet.size() == 0) {

\_labelVector.at(LabelVectorIndex(row, col)) = labelNumber;

\_labelSet.insert(labelNumber);

labelNumber++;

}

else {

\_labelVector.at(LabelVectorIndex(row, col)) = neighborLabelSet.at(0);

int mergeLabel = neighborLabelSet.at(0);

for (int maskIndex = 1; maskIndex < neighborLabelSet.size(); ++maskIndex) {

int combineLabel = neighborLabelSet.at(maskIndex);

for (int labelIndex = 0; labelIndex < \_binaryImage.rows \* \_binaryImage.cols; ++labelIndex) {

if (\_labelVector.at(labelIndex) == combineLabel) {

\_labelVector.at(labelIndex) = mergeLabel;

}

}

\_labelSet.erase(combineLabel);

}

}

}

在以上功能皆處理完後，我得到這張圖的label集合，以map索引方式隨機生成不重複顏色，方便接下來建構最終輸出圖像時做label-color的功能。

// 依照 \_labelSet 做出 label-color map

void SetColorLabelMap() {

set<vector<uchar>> colorSet;

int max = 255, min = 0;

srand(time(0));

for (int labelNumber : \_labelSet) {

bool isInColorSet = false;

vector<uchar> bufferColor;

do {

bufferColor.clear();

for (int color = 0; color < 3; ++color) {

bufferColor.push\_back(rand() % (max - min + 1) + min);

}

isInColorSet = colorSet.find(bufferColor) != colorSet.end();

} while (isInColorSet);

colorSet.insert(bufferColor);

\_labelColorMap[labelNumber] = bufferColor;

}

}

最後以label-color map對輸出圖像著色

// 依照 label-color map 將輸出圖著色

void DrawLabel() {

SetColorLabelMap();

uchar\* labelImagePtr;

for (int row = 0; row < \_labelImage.rows; ++row) {

labelImagePtr = \_labelImage.ptr<uchar>(row);

for (int col = 0; col < \_labelImage.cols; ++col) {

int labelCode = \_labelVector.at(LabelVectorIndex(row, col));

vector<uchar> colorDecode = \_labelColorMap[labelCode];

for (int bgr = 0; bgr < 3; ++bgr) {

\*labelImagePtr++ = (labelCode != 0) ? colorDecode.at(bgr) : 0;

}

}

}

}

因為4連通與8連通在步驟上僅在判斷修改區域上有差別，因此我利用函式指標，經由判斷得出要做n連通的處理，並指向n連通的判斷修改的函式，減少了duplicate code。

// 在class宣告中定義

void (LabelImage::\*\_labelPixelFunc)(const int&, const int&, int&) = nullptr;

// 依 neighborRule 輸出 label 圖像、數量

void LabelingByNeighbor(int neighborRule) {

if (neighborRule == 4) \_labelPixelFunc = &LabelImage::LabelPixelBy4Neighbor;

else if (neighborRule == 8) \_labelPixelFunc = &LabelImage::LabelPixelBy8Neighbor;

else {

cout << "Only support 4 or 8 neighbor." << endl;

return;

}

InitLabelingData();

int labelNumber = 1;

for (int row = 0; row < \_labelImage.rows; ++row) {

for (int col = 0; col < \_labelImage.cols; ++col) {

if (\_labelVector.at(LabelVectorIndex(row, col)) != 0) {

(this->\*\_labelPixelFunc)(row, col, labelNumber);

}

}

}

\_component = \_labelSet.size();

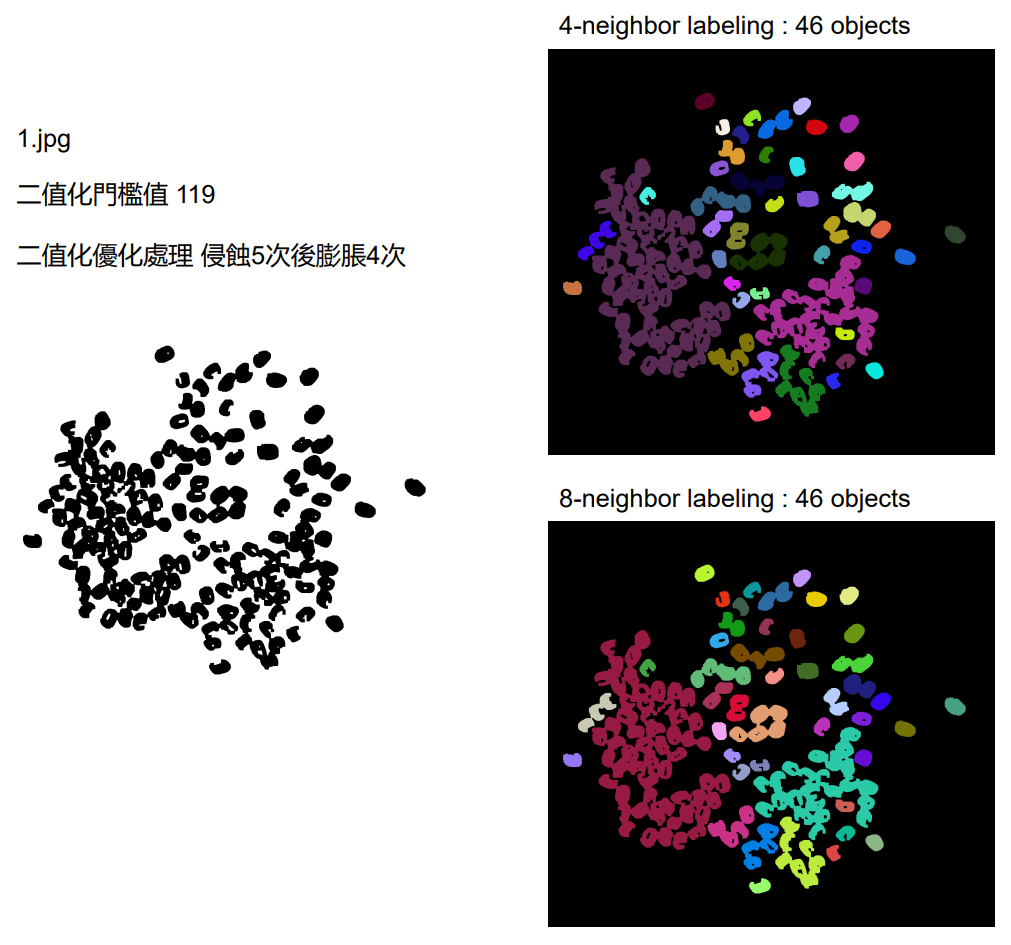
DrawLabel();

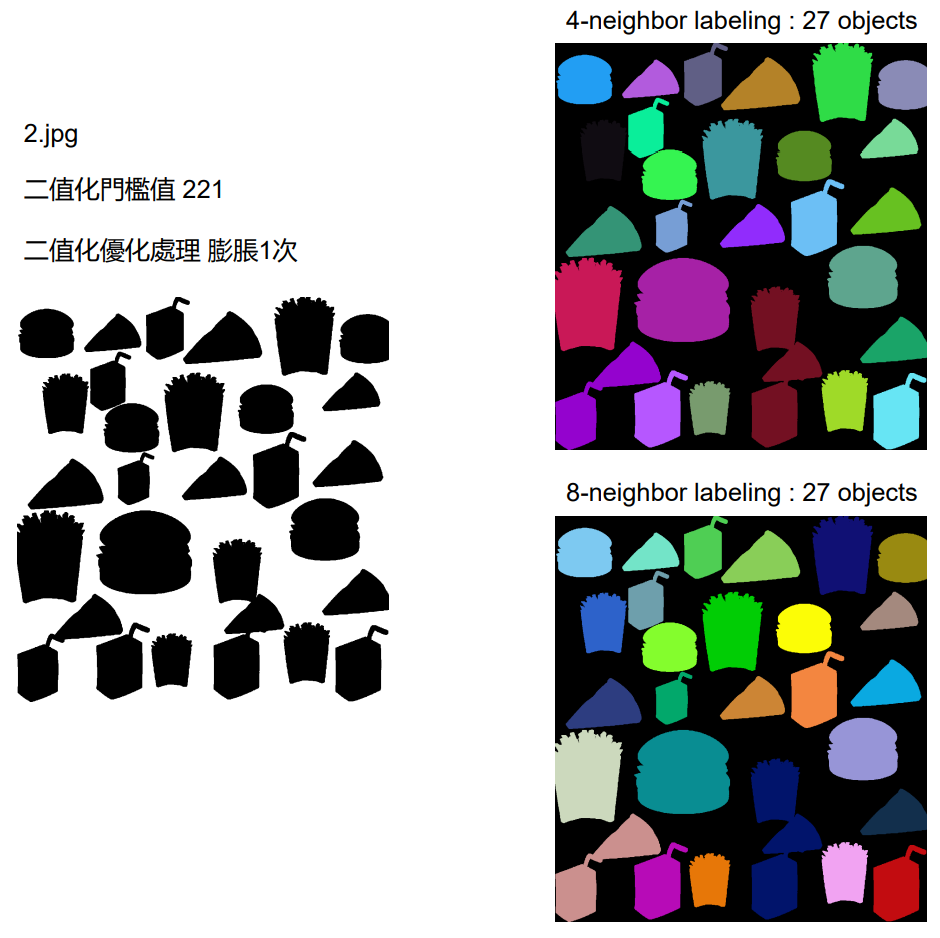
cout << \_name << " with " << neighborRule << "-neighbor has " << \_component << " objects" << endl;

imshow(\_name + " " + to\_string(neighborRule) + "-neighbor labeled", \_labelImage);

imwrite(LABELED\_FOLDER + to\_string(neighborRule) + "-neighbor\_" + \_name, \_labelImage);

}





一張含有 圖表 的圖片

自動產生的描述

