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。

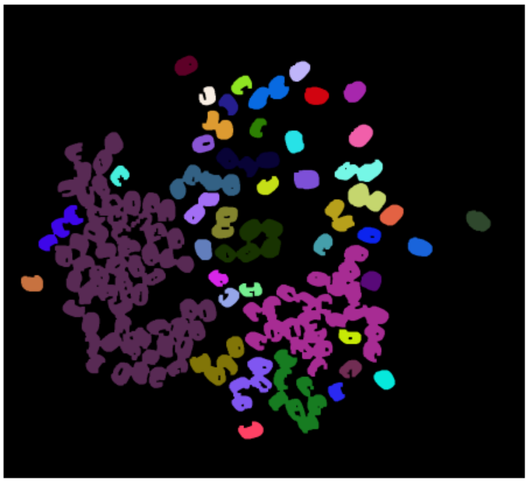
```
// 在 class 宣告中定義
void (LabelImage::*_labelPixelFunc)(const int&, const int&, int&) = nullptr;
// 依 neighborRule 輸出 label 圖像、數量
void LabelingByNeighbor(int neighborRule) {
    if (neighborRule == 4) _labelPixelFunc = &LabelImage::LabelPixelBy4Neighbor;
    else if (neighborRule == 8) _labelPixelFunc = &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);
}
```

1.jpg

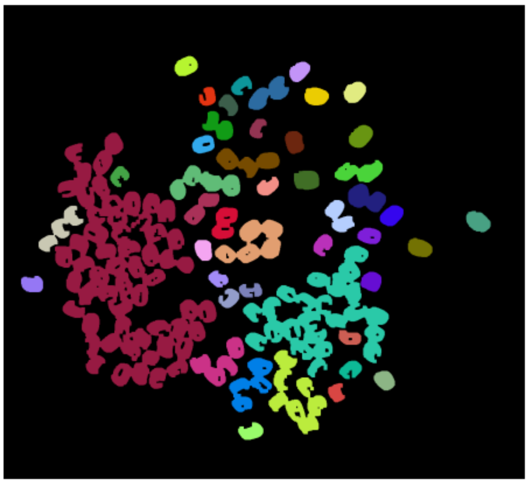
二值化門檻值 119

二值化優化處理 侵蝕5次後膨脹4次

4-neighbor labeling : 46 objects



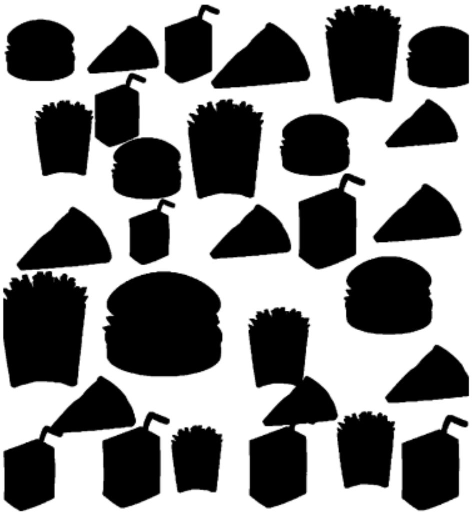
8-neighbor labeling : 46 objects



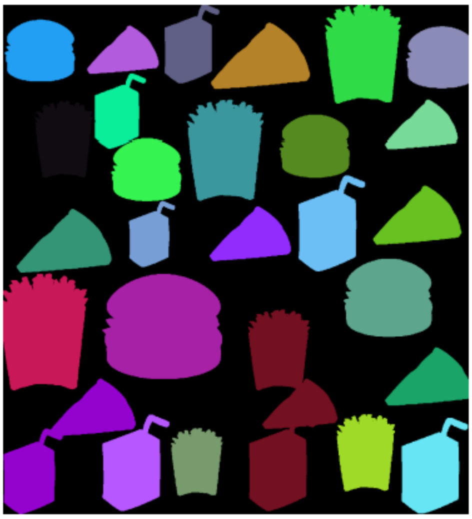
2.jpg

二值化門檻值 221

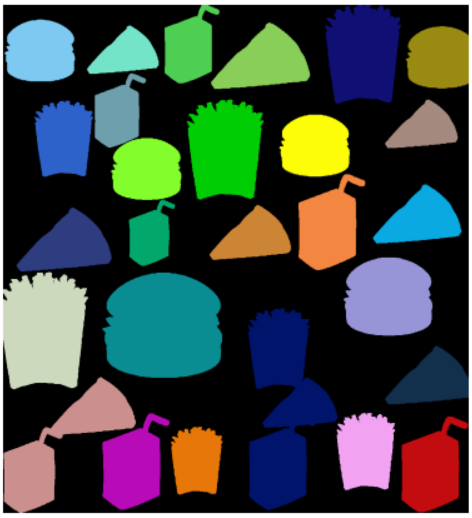
二值化優化處理 膨脹1次



4-neighbor labeling : 27 objects



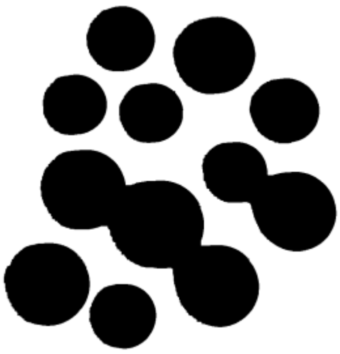
8-neighbor labeling : 27 objects



3.jpg

二值化門檻值 85

二值化優化處理 膨脹4次後侵蝕5次



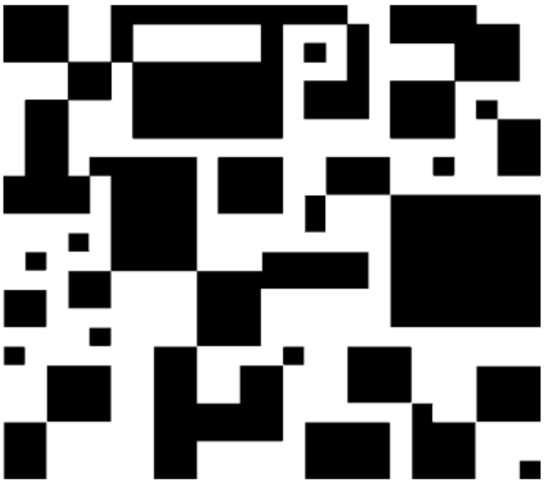
4-neighbor labeling : 9 objects

8-neighbor labeling : 9 objects

4.jpg

二值化門檻值 228

二值化優化處理 無



4-neighbor labeling : 26 objects

8-neighbor labeling : 23 objects