ラベリングクラスについて

井村 誠孝

2005 年 7 月 19 日

1 この文書

この文書は、画像の連続領域抽出 (ラベリング処理) を行うクラステンプレート Labeling について解説したものである。

2 動作環境

C++。STL(Standard Template Library) が使えること。
g++ (GCC) 3.3.2 で動作確認済みである。

3 使い方

3.1 入出力の型

Labeling はクラステンプレートであり、インスタンスの生成時に入力する画像のピクセルのクラス `SrcT` と出力する画像のピクセルのクラス `DstT` を指定する必要がある。

```
SrcT: unsigned char, shoft
DstT: short
```

で動作確認済みである。
よく使用されると思われる型は typedef してある。

1. 入力が unsigned char, 出力が short

```
typedef Labeling<unsigned char,short> LabelingBS;
typedef Labeling<unsigned char,short>::RegionInfo RegionInfoBS;
```

2. 入力が short, 出力も short

```
typedef Labeling<short,short> LabelingSS;
typedef Labeling<short,short>::RegionInfo RegionInfoSS;
```

3.2 入出力フォーマット

入力バッファの各画素が保持する情報は以下の意味を持つ。

- 0: ラベリング対象外画素
- 0 以外: ラベリング対象画素

  同一の値が連続している領域が、1 つの領域とみなされる。

出力バッファの各画素が保持する情報は以下の意味を持つ。

- 0: ラベリング対象外画素
- 0 以外: ラベリング結果 (連続領域の識別番号)

補足事項

- 一般的な使用形態としては、

  1. 画像をキャプチャ
  2. 適当な前処理 (背景差分など)
  3. 適当な閾値処理

  を経過した段階で、unsigned char や short のバッファに各ピクセルの状態が保持されていると想定している。

- 入力バッファと出力バッファが同じアドレスの場合、出力バッファのゼロクリアが省略できるため、より高速に実行される。

  省略可能な理由は、

  – ラベリング対象外の画素はラベリング前後で値が変化しない (0 のまま)
  – ラベリング対象画素に入っている値は連続領域識別番号で上書きされる

  ためである。

  入力バッファが保持している情報が以降の処理で必要ない場合には有用である。

- 出力バッファには領域番号が格納されるが、出力バッファの型が unsigned char の場合は最大 255 領域までしか保持できないため心もとない。出力バッファには short を使うことを推奨する。

  領域数が保持できる最大数を超えた場合の挙動は現状では不定である。オーバーフローした場合には残りの領域はすべて 0 とする処理を入れ、オーバーフローしたことをユーザに通知するべきである。

3.3 ラベリングの実行

ラベリングの実行は、メンバ関数 Exec の呼び出しにより行われる。入力バッファの先頭アドレス、出力バッファの先頭アドレス、バッファの横サイズ、バッファの縦サイズを指定する必要がある。また、以下の機能を持つ。

- 入力バッファと出力バッファが同一アドレスでも動作する (型も同一の場合)。

- ラベリング結果を領域の面積順にソートして認識番号を付加できる。
- 連続領域の最小面積 (これより小さい領域は削除する) を設定できる。

詳しくは関数一覧の節を参照のこと。

3.4 ラベリング結果

ラベリング結果は、出力バッファの各画素に連続領域の番号が格納される。

またクラス `Labeling<SrcT,DstT>::RegionInfo` にはより高次の情報が格納される。保持している情報は以下の通り。

- 領域に属する画素数
- 領域の中心の座標 (重心ではなく、あくまで最大と最小の中間)
- 領域の縦横サイズ
- 領域の最小座標、最大座標
- 領域の入力バッファでの値
- 領域の出力バッファでの値
- 領域の重心 (選択可能)
- 領域を構成しているピクセルの座標 (未実装)

4 ラベリングアルゴリズム

ラベリング処理の詳細は以下の通りである。

1. RasterSegment の抽出

   各ラスタ (水平方向直線) 毎に、0 以外の同一の値が連続している区間を探索し、RasterSegment クラスとして取り出す。

   RasterSegment クラスは、各ラスタ毎の list コンテナに保持される。

2. 連続領域を構成する RasterSegment の探索

   連続領域番号が未確定の RasterSegment を 1 つ取り出す。

   RasterSegment が属しているラスタの上下のラスタの list コンテナの中から、RasterSegment と連続しているものを取り出す。

   この手順を、取り出す RasterSegment がなくなるまで続ける。

3. 領域サイズ順の識別番号の付加

   RasterSegment の長さを合計することで各連続領域の面積を算出し、面積の降順に 1 から連続領域の識別番号を付ける。

4. 出力バッファへの書き込み

   各連続領域ごとに、その連続領域を構成している RasterSegment が占めている領域に領域番号を付加する。

5 機能の取捨選択

Labeling.h の先頭部分の #define で、以下の機能の有効/無効を選択可能である。

- 出力バッファをクリアする/しない

```
#define CLEAR_DST_BUFFER       1
```

- 出力バッファをクリアする際に、前処理段階で全画素を 0 にする/しない

```
#define CLEAR_ALL_DST_BUFFER   1
```

- 重心を計算する/しない

```
#define CALC_CENTER_OF_GRAVITY 1
```

6 関数一覧

6.1 クラス Labeling<SrcT,DstT>

- コンストラクタ

  `Labeling<SrcT,DstT>( void )`

- ラベリングを実行する

```
int Exec( SrcT *target, DstT *result, int target_width, int target_height,
          const bool is_sort_region, const int region_size_min )
```

  引数:

SrcT	*target	入力バッファ
DstT	*result	出力バッファ
int	target_width	バッファの横サイズ
int	target_height	バッファの縦サイズ
bool	is_sort_region	連続領域を大きさの降順にソートするか
int	region_size_min	最小の領域サイズ (これ未満は無視する)

  返値:

  0: 正常終了

- 領域数を返す

  `inline int GetNumOfRegions( void ) const`

- 領域数 (面積が指定の画素数以上のもの) を返す

  `inline int GetNumOfResultRegions( void ) const`

- num 番目の領域 (大きさ順にソートしている場合は連続領域番号=num+1) の情報を保持している RegionInfo クラスへのポインタを返す

  `inline RegionInfo * GetResultRegionInfo( const int num ) const`

6.2 クラス Labeling<SrcT,DstT>::RegionInfo

- 画素数を返す

  `inline int GetNumOfPixels( void ) const`

- 領域中央の座標 (!=重心) を返す

  `inline void GetCenter( float& x, float& y ) const`

- 領域のサイズを返す

  `inline void GetSize( int& x, int& y ) const`

- 領域の左上座標を返す

  `inline void GetMin( int& x, int& y ) const`

- 領域の右下座標を返す

  `inline void GetMax( int& x, int& y ) const`

- 領域の重心を返す

  `inline void GetCenterOfGravity( float& x, float& y ) const`

- 領域の入力バッファにおける値を返す

  `inline SrcT GetSourceValue( void ) const`

- 領域の連続領域番号を返す

  `inline DstT GetResult( void ) const`

- 領域の情報を出力する

  `friend std::ostream& operator<<( std::ostream& s, RegionInfo& ri )`

7 To-Do

- よりいっそうの高速化
- 領域数が想定より多かった場合の処理
- RegionInfo クラスに各領域を構成する画素の座標を格納

8 使用例

```
// Main.cpp

#include    <iostream>

#include    "Bmp.h"
#include    "ColorConv.h"
#include    "Labeling.h"
```

```
using namespace std;

// short 型のバッファに格納された結果を bmp ファイルに出力する

void
Output( short *s, int width, int height, char *filename )
{
    ColorConv   conv;

    unsigned char   *output = new unsigned char[ width * height * 3 ];

    unsigned char   *p = output;
    for ( int i = 0; i < width * height; i++ ) {
        if ( *s == 0 ) {
            *p++ = 0;
            *p++ = 0;
            *p++ = 0;
        } else {
            double  r, g, b;
            conv.HsvToRgb( *s / 16.0, 1.0, 1.0, r, g, b );
            *p++ = static_cast<unsigned char>( r * 255 );
            *p++ = static_cast<unsigned char>( g * 255 );
            *p++ = static_cast<unsigned char>( b * 255 );
        }
        s++;
    }

    Bmp bmp_manip;
    bmp_manip.Write( filename, width, height, output );

    delete [] output;
}

// メイン関数

int
main( int argc, char **argv )
{
    unsigned char   *image;
    int width, height;

    // bmp ファイル読み込み
```

```
Bmp bmp_manip;
image = bmp_manip.Read( argv[ 1 ], width, height );

// バッファの確保

short   *src_buf = new short[ width * height ];
short   *dst_buf = new short[ width * height ];

// 画像の赤成分をラベリングソースとして使用する

while ( 1 ) {
unsigned char   *p = image;
short   *q = src_buf;
for ( int j = 0; j < width * height; j++ ) {
    *q++ = *p;
    p += 3;
}

// ラベリングの実行

LabelingSS  labeling;   // = Labeling<short,short>
labeling.Exec( src_buf, dst_buf, width, height, true, 10 );

// 画像をファイルに出力

Output( dst_buf, width, height, argv[ 2 ] );

// 領域情報をテキストで出力

cout << "number of regions: " << labeling.GetNumOfRegions() << endl;
cout << "number of result regions: "
     << labeling.GetNumOfResultRegions() << endl;

for ( int i = 0; i < labeling.GetNumOfResultRegions(); i++ ) {
    RegionInfoSS    *ri = labeling.GetResultRegionInfo( i );
    cout << "--------------------------------" << endl;
    cout << "number: " << ri->GetResult() << endl;
    cout << "number of pixels: " << ri->GetNumOfPixels() << endl;
    float   fx, fy;
    ri->GetCenter( fx, fy );
    cout << "center: (" << fx << "," << fy << ")" << endl;
    int x, y;
    ri->GetSize( x, y );
    cout << "size: (" << x << "," << y << ")" << endl;
```

```
        ri->GetMin( x, y );
        cout << "min: (" << x << "," << y << ")" << endl;
        ri->GetMax( x, y );
        cout << "max: (" << x << "," << y << ")" << endl;
        ri->GetCenterOfGravity( fx, fy );
        cout << "center of gravity: (" << fx << "," << fy << ")" << endl;
        cout << "source value: " << ri->GetSourceValue() << endl;
    }

    delete [] src_buf;
    delete [] dst_buf;

    return 0;
}
```

9 連絡先

バグ報告・要望は井村 (imura@is.naist.jp) まで。