麻雀点数計算プログラム 注釈書

14EC004 飯田頌平

2016 年 3 月 17 日

1　はじめに

本稿は吉田さんの麻雀点数計算プログラム [1] の注釈書にあたる。

このプログラムはカメラで麻雀の手牌を撮影すると点数を自動で計算するというものであり、大まかにサーバ、画像認識、点数計算の三つのモジュールに分かれている。

このうち本稿では画像認識の部分について触れる。画像認識の実装ファイル TemplateMatching.scala およびユーザ定義関数の実装ファイル package.scala を付録に掲載するので、参考にしながら読むこと。

2　プログラムの実行方法

プログラムは github 上に上げられているが、それを clone するだけでは動作しない。sbt をインストールした上で、opencv を使用できるようにせねばならない。

Ubuntu であれば、sbt は以下のコマンドでインストールできる。

ソースコード 1　sbt

```
1 $ sudo apt-get install sbt
```

opencv は各自自力でコンパイルする必要がある。公式ドキュメントを参考にして、以下のコマンドを実行して.so ファイルを生成する。

ソースコード 2　opencv

```
1 $ git clone git://github.com/Itseez/opencv.
    git
2 $ cd opencv
3 $ mkdir build
4 $ cd build
5 $ cmake -DBUILD_SHARED_LIBS=OFF
6 $ make -j8
```

.so ファイルは opencv/build/lib に生成されるはずである。もし生成されていない場合は、cmake の出力結果を確認すること。To be built の項目に java とない場合は、java-8-oracle の環境変数を設定できていないので、

ソースコード 3　JAVAHOME

```
1 $ JAVA_HOME=/usr/lib/jvm/java-8-oracle
```

を実行すること。

生成された.so ファイルを、clone してきた mahjongs ディレクトリの下の mahjongs/recognisser/lib に置けば sbt 環境下で opencv を使用することができるようになる。

3　画像認識のフロー

画像認識はパターンマッチングによって行われている。よって、最初にテンプレートを生成し、その後テストデータとテンプレートを照合させて結果を求める。まずはテンプレートの生成手順を示す。

1. 雀牌を黒と白で二値化する
2. 雀牌の輪郭を判別し、雀牌だけを切り取る
3. 雀牌から牌ひとつあたりの縦幅と横幅を求める
4. 黒と判別された誤差（牌の隅）を白く塗りつぶす

5. 牌の絵柄部分の輪郭情報を得る
6. 輪郭情報と雀牌のグレースケール画像からテンプレートを生成する

詳細な説明については後述するとして、次に照合の手順を示す。

1. 手牌の輪郭を判別し、手牌（純手牌＋鳴牌）だけを切り取る
2. テンプレートと同じサイズにリサイズする
3. 取得した輪郭の頂点数を求める（四角形なら純手牌である）
4. テンプレートから各牌を回転させたイメージを得る
5. 手牌とテンプレート（及び回転イメージ）をテンプレートマッチングで照合する
6. 照合結果に牌種のシーケンスを与え、結果の最大スコアを取り出す
7. 最大スコアを基に矩形を作成し、手牌を矩形領域で塗りつぶす
8. ５．６．７．の手順を再帰的に繰り返すと、手牌が黒く塗りつぶされてゆく。完全に塗りつぶされたら、再帰を終了する
9. 純手牌を第一要素、鳴牌を第二要素のタプルを作り、関数の結果として返す

こうして手牌を認識することができた。次章以降ではこれらの流れの詳細について述べる。

4　テンプレートの生成

テンプレートの生成にあたって、まずは元画像を用意する。以下にその要件を示す。

- 雀牌を 4*9 の矩形上に隙間なく並べる。矩形は平行になるようにする
- 牌の位置は固定である。一萬からはじまる SEQ が予め定められている
- 雀牌と背景以外のものを除外する
- 撮影時に、各牌が等しい高さと幅を持つよう注意する
- 二値化からの境界値探索を阻害しないよう背景を選ぶ

背景には麻雀マットなどを用いれば良い。また斜めから撮影された画像からは正しいテンプレートが得られないことに気を付ける。

実際に使用したテンプレート画像を図 1 に示す。

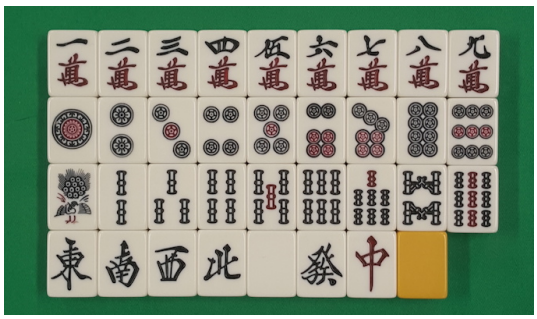

図 1　テンプレート

この画像の画素値は幅 680x 高さ 396 ピクセルであり、ビットの深さは 32 である。このビット数はグレースケール変換後（図 2）には 8 ビットまで圧縮される。

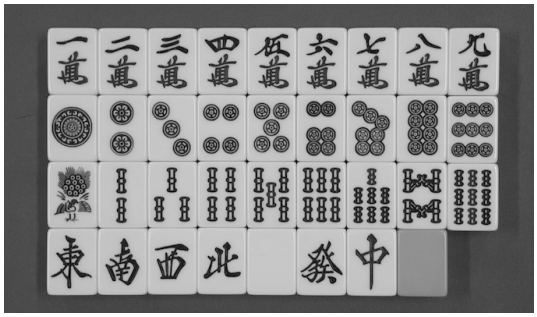

図 2　テンプレート グレースケール

元画像は図 2 の状態に変換された後にプログラムへ Mat 型で渡される。Mat 型は opencv[*1]のライブラリが提供する行列の型であり、opencv ではこの型で画像データを扱う。Mat 型の内部構造は行列、画素値、画素のデータ型である。createTemplate の引

[*1] https://github.com/bytedeco/javacv

数 mat が元画像のデータを示す変数である。

mat に対して最初に行われる処理は、定数 MaxResolution のサイズに近づくよう resize で画像サイズを修正する。大きい画像であると処理時間が増えてしまうため、これによって効率化する。

次に雀牌と背景を識別し、雀牌だけを切り取る crop を実行する。crop は元画像を引数に取り、Buffer 型のシーケンスで識別結果の画像を返す。（なお Buffer はタプルであり、画像だけでなく輪郭の二次元配列も返しているが、この値は createTemplate では参照しない。）

crop の実装は、まず findContours によって元画像を輪郭で分離し、minAreaRect で矩形を得て、warpAffine で矩形の傾きを底面と垂直になるよう修正し、getRectSubPix で矩形領域のピクセル値を得ることで、入力画像を輪郭毎に分離した上で補正をかけた画像を得るようになっている。

輪郭を切り取る関数 findContours の理論を示す。

1. 二値化を行う
2. 画素の左上から右下にかけて白の値を走査する
3. 白の値が見つかれば、そこを基点と定める
4. 基点に隣接する白の値を探し、見つかればそこを新たな基点として再帰的に輪郭を得る。見つからなければ孤立点または孤立線として捉え、輪郭を抽出できたとする
5. 最初の基点と現在の基点が重なったときに、輪郭を抽出出来たとする
6. すべての画素の走査が終了するまで２．～５．を繰り返す

当プログラムでは最も高いヒエラルキーの輪郭のみを抽出しているため、処理が高速である。ヒエラルキーとは opencv の概念であり、輪郭の内部にまた別の輪郭がある場合、後者は前者の子だとして階層関係を定義しているのだが、ここでは子となるあらゆる輪郭を無視している。

findContours の実装についてはドキュメントに数式が散見されず、またソースコード（contours.cpp）が 1700 行にも上る [5] ため、解析は困難であった。代わりに上述の理論に基づいたアルゴリズムを自分で実装したので、そのようなアルゴリズムが使われていたのだと仮定することにする。実装したアルゴリズムを以下に示す。

ソースコード 4　輪郭抽出アルゴリズム

```
import cv2

class boader:

  base = {"row": 0, "col": 0}
  isolate = []

  def threshold(self, img):
    res = img
    for y in range(len(img)):
      for x in range(len(img[y])):
        if img[y][x] >= 128:
          buf = 255
        else:
          buf = 0
        res[y][x] = buf
    return res

  def directions(self, n):
```

```
    """
    When recursiving, send index of direction.
    Right-back direction is gotten by (n+4+1) % 8
    """
    if n==0:
      direction = [[−1,−1], [0,−1], [1,−1], [1,0], [1,1], [0,1], [−1,1], [−1,0]]
    elif n==1:
      direction = [[0,−1], [1,−1], [1,0], [1,1], [0,1], [−1,1], [−1,0], [−1,−1]]
    elif n==2:
      direction = [[1,−1], [1,0], [1,1], [0,1], [−1,1], [−1,0], [−1,−1], [0,−1]]
    elif n==3:
      direction = [[1,0], [1,1], [0,1], [−1,1], [−1,0], [−1,−1], [0,−1], [1,−1]]
    elif n==4:
      direction = [[1,1], [0,1], [−1,1], [−1,0], [−1,−1], [0,−1], [1,−1], [1,0]]
    elif n==5:
      direction = [[0,1], [−1,1], [−1,0], [−1,−1], [0,−1], [1,−1], [1,0], [1,1]]
    elif n==6:
      direction = [[−1,1], [−1,0], [−1,−1], [0,−1], [1,−1], [1,0], [1,1], [0,1]]
    elif n==7:
      direction = [[−1,0], [−1,−1], [0,−1], [1,−1], [1,0], [1,1], [0,1], [−1,0]]
    else:
      print("directions catch undefined number as parameter:" + str(n))
    return direction

  def directionDecoder(self, y, x):
    # getting forward direction
    n = 0
    if x==−1:
      if y==−1: n = 0
      elif y==0: n = 1
      elif y==1: n = 2
    elif x==0:
      if y==−1: n = 7
      elif y==1: n = 3
    elif x==1:
      if y==−1: n = 6
      elif y==0: n = 5
      elif y==1: n = 4
    res = (n+5)%8
    print("res:" + str(res) + ", y:" + str(y) + ", x:" + str(x))
    return res

  def recursiveSearch(self, img, row, col, contour, directionNum):
    """
    1. base-point is passed as parameter
    2. search black-point using counterclockwise rotation.
       start direction is right-back
    3. if find black-point, deciding direction and recursiving and fill new black-point
       marker number "1". marker number is reseted on going to new base-point
```

```
␣␣␣␣␣␣␣␣if␣not␣find␣black-point,␣back␣previous␣point␣and␣search␣another␣point
␣␣␣␣4.␣if␣reach␣base-point,␣contour␣extraction␣is␣finished
␣␣␣␣"""

    # 8 direction (Northwest -> West -> ... -> North)
    direction = self.directions(directionNum)
    print("-" * 30)
    print("row:" + str(row) + ",␣col:" + str(col))
    print("directionNum:" + str(directionNum))
    print("direction:")
    print direction

    for i in range(len(direction)-1):
      y = row + direction[i][0]
      x = col + direction[i][1]

      # only permit number can be in index
      if (y >= 0 and y < len(img)) and (x >= 0 and y < len(img[y])):
        if len(contour) > 30: #forcely finish(for debug)
            b += 0
        # if find black-point
        if img[y][x]==0:
          # if black-point is base-point
          if y==self.base["row"] and x==self.base["col"]:
            img[y][x] = 1 # fill
            return (contour, True) # back to recursiving-parent
          else:
            newDirectionNum = self.directionDecoder(direction[i][0], direction[i][1])
            img[y][x] = newDirectionNum+10 # fill (for duplication)
            c = contour
            c.append([y, x])
            print("direction=>[y,␣x]:", str(direction[i][0]) + ",␣" + str(direction[i][1]))
            res = self.recursiveSearch(img, y, x, c, newDirectionNum)
            if res[1] == True:
              self.isolate = [] # reset
              return res # back to searchBoader

    # if not found new black-point
    """
␣␣␣␣1.␣traceback␣previous␣point
␣␣␣␣2.␣check␣new␣black-point(without␣deadend␣route)
␣␣␣␣␣␣␣␣deadend␣route␣is␣ignored␣by␣fill
␣␣␣␣3.␣if␣traceback␣reach␣base-point,␣this␣case␣is␣getting␣curve
␣␣␣␣4.␣in␣other␣case,␣when␣isolate␣curve␣is␣detected,␣use␣global␣variable␣"isolate"␣and␣
    assign␣deadend-contour.␣"isolate"␣is␣final␣variable,␣re-assignment␣is␣only␣permitted␣in
    ␣end␣of␣recursiving␣for␣reset
␣␣␣␣"""
    if len(self.isolate) == 0:
      self.isolate = contour
```

```
115     if len(self.isolate)==1 or row==self.base["row"] and col==self.base["col"] :
116       img[y][x] = 1 # fill
117       isl = self.isolate
118       self.isolate = []
119       return (isl, True) # back to recursiving-parent
120     c = contour[:-1]
121     res = self.recursiveSearch(img, contour[len(contour)-1][0], contour[len(contour)-1][1], c, 0) # in this
            case, black-point duplication is not found. then, directionNum is any value.
122     if res[1] == True:
123       self.isolate = []
124       return res
125
126     print("contour::False")
127     self.isolate = []
128     return (contour, False)
129
130   def searchBoader(self, img):
131     """search␣most␣external␣contour."""
132     res = []
133     img = b.threshold(img)
134     for y in range(len(img)):
135       for x in range(len(img[y])):
136         if img[y][x] == 0:
137           self.base["row"] = y
138           self.base["col"] = x
139           buf = self.recursiveSearch(img, y, x, [[y, x]], 0)
140           if buf[1]==True:
141             res.append(buf[0])
142             img = self.fill(img, buf[0])
143
144     self.show(img)
145     return res
146
147   def show(self, img):
148     print img
149
150   def fill(self, img, contour):
151     """
152 ␣␣␣␣fill␣inner␣contour.
153 ␣␣␣␣in␣case,␣fill␣make␣black␣pixel␣inner␣contoutr␣assign␣value␣of␣2.
154 ␣␣␣␣because␣black␣pixel␣referenced␣by␣searchBoader␣is␣only␣value␣of␣0.
155 ␣␣␣␣"""
156     fillVal = 2
157     whiteVal = 255
158     flag = 0
159     for c in contour:
160       img[c[0]][c[1]] = fillVal
161     for y in range(len(img)):
162       prev = whiteVal # prev is used for detecting edge
```

```
        for x in range(len(img[y])):
          if not img[y][x] == prev:
            if prev > img[y][x]: flag = 1
            else: flag = 0
          if flag == 1: img[y][x] = fillVal
          prev = img[y][x]

    return img


b = boader()
img = cv2.imread("zero.jpg", 0)
res = b.searchBoader(cv2.resize(img, (10, 10)))
print res
```

このアルゴリズムでは次の基準点を探すときに方向を取得しており、移動方向から右斜め後ろにあたる点から反時計回りの順で画素の輝度を探索するようにしている。

このアルゴリズムについて図 3 で実験を行ったところ、図 4 のような結果が得られた。出力されたリストを見れば、輪郭を抽出できていることがわかる。

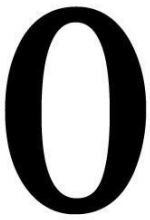

図 3　輪郭抽出実験 元画像

```
[[255 255 255 255 255 255 255 255 255 255]
 [255 255 255   2   2   2   2 255 255 255]
 [255 255   2   2 255 255   2 255 255 255]
 [255 255   2 255 255 255   2   2 255 255]
 [255 255   2 255 255 255   2   2 255 255]
 [255 255   2 255 255 255   2   2 255 255]
 [255 255   2 255 255 255   2   2 255 255]
 [255 255   2   2 255 255   2 255 255 255]
 [255 255 255   2   2   2 255 255 255 255]
 [255 255 255 255 255 255 255 255 255 255]]
[[[1, 3], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2], [7, 2], [8, 3], [8, 4], [8, 5], [7,
6], [6, 7], [5, 7], [4, 7], [3, 7], [2, 6], [1, 6], [1, 5], [1, 4]]]
```

図 4　輪郭抽出実験 結果

麻雀認識プログラムの輪郭抽出の話に戻るが、最終的に、図 5 のように輪郭が取得される。

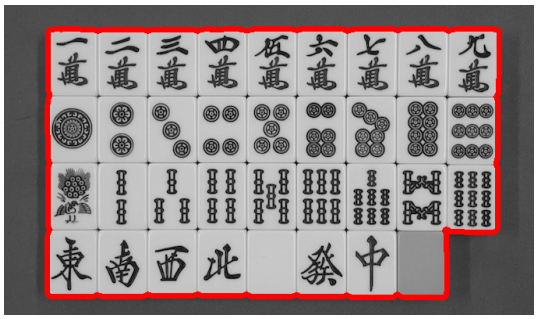

図 5　テンプレート 輪郭

opencv の輪郭抽出メソッド findCountours から得た輪郭 countour のメソッド toArray を用いれば、minAreaRect で矩形を引くべき座標を得られる。

こうして得られた分離後の画像のシーケンス（返り値は Buffer 型であることに注意）から headOption を使うことで先頭要素を取り出せる。findContours を呼び出した際に大きさ順に画像をソートしているので、先頭要素には一番大きな画像が入る。一番大きな画像が雀牌であるという前提であるなら、この先頭要素は雀牌であるため、雀牌のテンプレートは引数 template に束縛される。findCounters は複数の輪郭を検出できたとき、それらをまとめて返すため、このようにして雀牌を取り出す。

以上の処理を行い輪郭抽出を終えた画像を図 6 に示す。

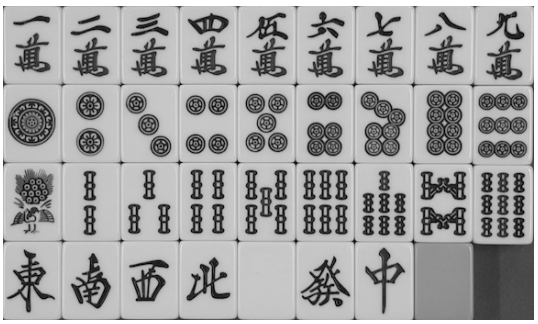

図 6　テンプレート 輪郭抽出済

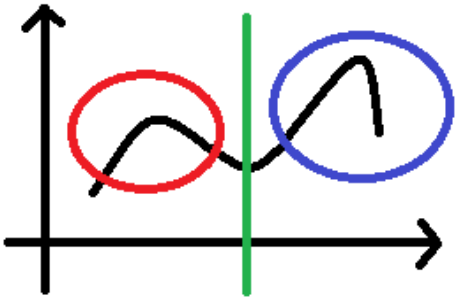

図 8　輝度－画素数のヒストグラム

得られた雀牌画像を threshold によって黒と白で二値化すると、図 7 の様な画像になる。

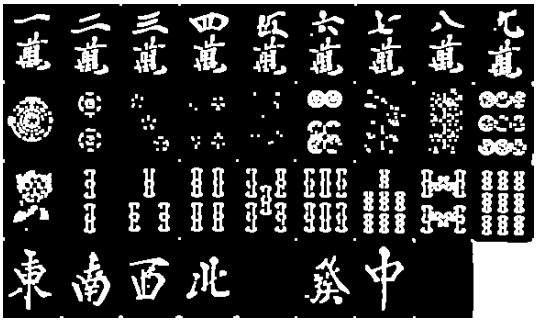

図 7　テンプレート 二値化

ここでは模様が白、牌の白地が黒に分離されている。二値化処理では白地が白となるのが普通だが、ここでは二値化の後に反転をかけてあえて白地を黒、模様を白としている。これは opencv の輪郭抽出処理が黒を背景として切り捨てるため [6] であり、この後模様の輪郭抽出を行う際に模様を白くして抽出できるようにする。

二値化の閾値は大津の二値化と呼ばれるアルゴリズムを用い、統計的に自動で算出する。大津の二値化について説明する。まず横軸に輝度、縦軸に画素数を取ったヒストグラム（図 8）を画像データから生成する。

図中の赤い領域と青い領域をうまく分離できるような閾値（緑線）を探索することがこのアルゴリズムの目的である。

閾値を評価する指数として、分離度という値を導入する。分離度はクラス内分散 σ_w に反比例し、クラス間分散 σ_b に比例する。σ_w、σ_b はそれぞれ以下の式で表される。

$$\sigma_w = \frac{w_1{\sigma_1}^2 + w_2{\sigma_2}^2}{w_1 + w_2} \tag{1}$$

$$\sigma_b = \frac{w_1(m_1 - m_t)^2 + w_2(m_2 - m_t)^2}{w_1 + w_2} \tag{2}$$

クラス内分散とは各クラス（赤領域と青領域）についての分散を出し、各クラスのデータ数で重み付けをして求められる、各クラス毎のまとまりを包括的に示す指標である。

一方クラス間分散とは全体の平均値と各クラスの平均値を使って求められる分散であり、これは各クラスがどれだけ散らばっているのかを示す指標である。

最終的に分離度は、定数を切り離して

$$\sigma = w_1 w_2 (m_1 - m_2)^2 \tag{3}$$

によって示される。式 3 が最大となる閾値を求めればよい。輝度が 8bit ならば試行回数は 256 で済むので、総当りで探索しても差し支え無いだろう。

こうして得られた二値化後の画像（図 7）について見てみると、牌の模様が白、牌の背景が黒で表さ

れているものの、牌と牌の間なども白く判定されてしまっている。牌の模様以外が白と認識されてはパターンマッチに失敗しやすくなるため、次はこの誤判定部分を黒く塗りつぶす。それには floodFill を使う。floodFill は座標を指定し、連結部分を指定した色で塗りつぶす関数である。これを牌と牌の間にすべてに対して実行することで、誤判定が塗りつぶされる。

さらに floodFill の補完として、dilate を使用する。dilate は膨張処理によってノイズを消す効果があり、ターゲットの周辺に 1 ピクセルでも白があれば、ターゲットを白に置き換える処理を行う。これによって牌の模様をはっきりとさせる。

floodFill と dilate をかけた結果の画像を図 9 に示す。図 7 に比べると、牌と牌の間が塗りつぶされていることと、模様の部分が膨張していることがわかる。

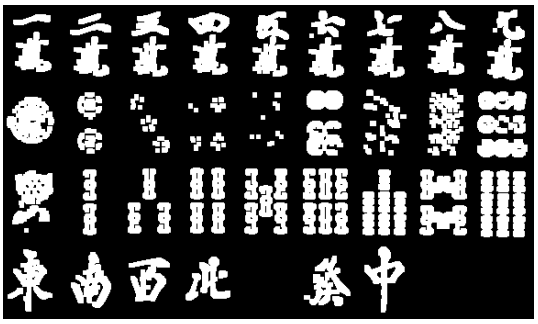

図 9　塗りつぶし・膨張処理後

そして牌の模様だけを抽出する。二値化した値に対して findContours によって再び輪郭ごとの分離をかければ、模様だけを抽出することができる。四筒のように模様が複数に分離されてしまうケース（二筒ふたつとして認識される）の場合は、convexHull によって結合する。convexHull は点群の凸包を得た後に外接する矩形で近似する。

凸包の近似のアルゴリズムを示す。まずは基準点を決定する。この際基準点となりうる点は、凸包を形成する点でなくてはならないので、点集合の最も上部にある点を基準点とする。この点を求めるには、画像の左上から右下にかけて走査し、最初に当たった点を極地にある点だと見做せばよい。

次に基準点を始点とする半直線（基準線）を置く。左上から右下への走査を行った場合は、画像に対し真上の方向か、それに左へ直角な基準線を考える。

そして基準点からあらゆる他の点集合に対して直線を引き、その直線と基準線との角度の差が最も小さい点が、凸包を形成する点であると見なせる。

後は再帰的にこれまでの処理を繰り返し、多角形が形成されれば凸包を求めることができる。

凸包から話を戻す。得られた模様の中で、もっとも面積の大きなものを size に代入する。マッチングを行う際にはこのサイズを基準にするためである。

ここまでで二値化を通して牌の模様の輪郭情報を抽出する処理を行った。今度はこれをグレースケール画像に適用し、牌一種類ごとのテンプレートを得る。

それには grid を用いる。grid は submat を実装に含み、元画像から矩形範囲を抽出できる関数である。grid で牌を 4*9 に区切り、zip で牌ごとに模様の輪郭の情報を付与し、それらに flatMap を挟んで getRectSubPix をかけることによって、輪郭だけが抽出されたすべての牌のテンプレートを得ることができる。

以上でテンプレートの生成が完了した。

5　テストデータとの照合

照合は関数 recognize によって実装されている。この引数 mat に手牌の画像データを、templates にテンプレートの画像データを渡す。

実際に使用した手牌データを図 10 に示す。

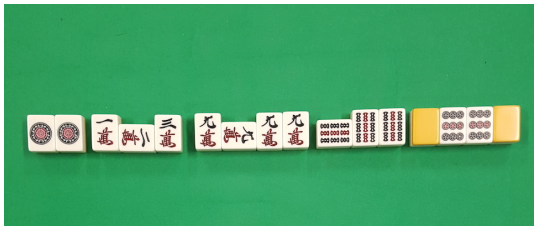

図 10　手牌

この画像の画素値は幅 959x 高さ 400 ピクセルであり、ビットの深さは 32 ビットである。手牌の場合も二値化を施すと 8 ビットまで圧縮される。

手牌のデータは crop によって背景から切り離される。しかしテンプレートを crop で切り離したときと違い、この場合は牌のデータが複数に分かれる。手牌には純手牌と鳴牌があり、鳴牌は純手牌と離して置くためだ。この時の出力結果は以下のようになる。

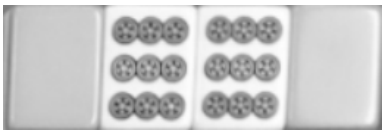

図 11　手牌 分離後 1

図 12　手牌 分離後 2

図 13　手牌 分離後 3

図 14　手牌 分離後 4

図 15　手牌 分離後 5

図のデータの順は、配列変数内の順に一致する。その順は画像データの面積によって決められており、分離時に得た輪郭を利用して領域内の面積を求めている。

面積を求めるには、まず輪郭情報から四方向（上下左右）の極地点を取得し、次に各極値が持つ座標情報から輪郭に外接する矩形を求め、その長辺と短辺を掛けあわせれば良い。

手牌のデータは collect によって hand に束縛される。まずは hand をテンプレートのサイズに合わせてリサイズする。それから頂点数を求め純手牌か否かの情報を edge に代入して、テンプレートに向きを回転させた牌を加えて tiles に代入すると、いよいよ go でパターンマッチを行う。

go の内部では、まず各テンプレートに対して opencv のテンプレートマッチング関数たる matchTemplate を呼ぶ。その結果が result へと代入されるので、minMaxLoc を通して最高スコアと最低スコアの情報を取り出し、使用したテンプレートとスコア情報のタプルを locs に加えてゆく。この手順をすべてのテンプレートに対して実行する。minMaxLoc の返り値はドキュメントによると [2] Core.MinMaxLocResult 型である。座標を返す maxLoc フィールドと値を返す maxVal フィールドを参照することで最高スコアの情報を得ることができる。

マッチングの結果はは図 16 のような形で得られる。結果の画素はテンプレートと比較対象の画素の差であり、白い部分ほどスコアが高いことを意味している。

図 16　マッチング結果

図 17　テンプレート

図 16 の結果は図 12 に対して九萬のテンプレート（図 17）でマッチングを行った結果である。左端と右端、それから「マッチング結果」の「果」の上にあたる部分で白い点のようなものが見えるが、この三点こそ縦向きの九萬が発見された点を示している。

すべてのテンプレート（及びその回転）に対してのマッチングの結果 locs から、loc 値が最大となるものを maxBy によって取得し、変数 ((tile,loc),i) に代

入する。スコアが最大となる位置は loc.maxLoc で参照でき、それにテンプレートの大きさ tile.size を組み合わせることで、マッチした手牌の輪郭 rect を得られる。rect の実装では、最大スコアの場所とサイズから、基準点と縦横のサイズ情報を持つ矩形を作成している。

ここで go の引数を見てみる。go の引数は rects であり、それは rect と、そのインデックス番号のタプルのシーケンスである。go は再帰を前提としており、このシーケンス rects は再帰のたびに認識結果を積み重ねてゆく。

rectangle によってマッチングした牌の輪郭部分に矩形を描き、矩形内部の領域を塗りつぶすと、go を再帰する。このときシーケンスである引数 rects にマッチした牌 rect を加えており、rects はすべての牌についてのマッチングが終了した時には関数 go の返り値となる。

矩形を引いた結果を図 18 に、塗りつぶした結果を図 19 に示す。

図 18　矩形描画

図 19　塗りつぶし後

このように塗りつぶされた箇所は、以後の再帰でのマッチングにおいて、テンプレートにマッチしなくなる。

矩形領域がすべて塗りつぶされたあとは、適当な地点が最大スコアとして返り、そこからそのループでの rect が得られるが、intersects によって現在の rect と過去の rect（rects に保存）の矩形領域が重なっているか否かの判別を行い、重なっていた場合にマッチング終了という判定を下す。

マッチングを終えたら結果を手牌の位置順にソートし、シーケンス情報だけを取り出し indices に代入する。最後に純手牌か否かの判別を改めて行い、その結果と indices のタプルを result に代入していく。

result は純手牌および鳴牌のまとまり毎に要素数がひとつずつ増える構造であり、最終的に、純手牌と鳴牌のシーケンスのタプルが recognize の返り値となる。

6　matchTemplate について

opencv におけるテンプレートマッチングを行うメソッド matchTemplate は、以下の引数を取る。

1. image, テンプレートの探索対象となる画像。8 ビットまたは 32 ビットの浮動小数点型
2. templ, 探索されるテンプレート。image と同じデータ型である必要がある
3. result, 比較結果のマップ
4. method, 比較方法の指定

プログラムでは method に TM_CCORR_NORMED を指定している。これは数式で表すと

$$R(x,y)=\frac{\sum_{x'=0}^{w(T)}\sum_{y'=0}^{h(T)}(T(x',y')\cdot I(x'+x,y'+y))}{\sqrt{\sum_{x'=0}^{w(T)}\sum_{y'=0}^{h(T)}T(x',y')^2}\sqrt{\sum_{x'=0}^{w(T)}\sum_{y'=0}^{h(T)}I(x'+x,y'+y)^2}}$$

図 20　相互相関関数

であり、一般に相互相関関数と呼ばれる式である。なお w(T) は T の横の画素数を、h(T) は T の縦の画素数を指す。

この式の意味するところは、テンプレートと対象画像の畳込み演算である。ふたつのベクトル（または関数）に対しての畳込み演算は、積の総和によって求められる。これを相関と言うが、この値が大きければ大きいほど、正の相関度が高いと言える。

相関について別の見方をすると、ベクトルの内積である。これを逆手に取って、各ベクトルの大きさの積で割ることによって、cos θ だけを得られるようになる。$-1 \leq cos\theta \leq 1$ より、相関結果にスカラ値

の割り算を施すと、-1 から 1 の範囲で正規化される。相互相関関数の左辺は内積の角度 cos θ を示しており、cos θ が 1 に近づけば近づくほど、θ は小さく、テンプレートと対象画像の領域が近づいていることを意味する。

各ピクセルに対して内積を求める操作は行われ、その総和を以ってスコアとするので、スコアが大きければ大きいほど、テンプレートとマッチしているという結果になる。この点は最小誤差によってマッチングを行う methodSQDIFF とは逆の意味になっているので、関数 matchTemplate の結果 result には注意が必要である。

matchTemplate の結果である result については、スコアのマップ形式で返される。マップについて説明する。そもそもテンプレートマッチングとは、テンプレート画像と対象画像を重ね合わせて画素の差を比較するという行為を、各ピクセルに対して行うものである。つまり重ねあわせが行われる範囲でマッチングの結果が得られるのであって、それは対象画像のサイズとテンプレート画像のサイズの差の範囲である。この範囲の行列をマップと呼ぶ。（この呼び名は opencv のドキュメント [3][4] に即する）そしてマップの値にはテンプレートマッチングのスコア（相互相関関数の値）が入るのである。

差の範囲でのみ比較を行うという性質上、テンプレートと対象画像のサイズ差が小さい場合は、非常に小さい計算量で済むという特徴がある。テンプレートマッチングの結果である図 16 を見ると、横長の帯状の結果が得られているが、これは図 12 の対象画像に対して、図 16 の範囲のみマッチングを行ったという証左である。

参考文献

[1] Sanshiro Yoshida. mahjongs.
https://github.com/halcat0x15a/mahjongs

[2] OpenCV 2. 4. 2 Java API. Core. MinMaxLocResult.
http://docs.opencv.org/java/2.4.2/index.html?org/opencv/core/Core.html

[3] 物体検出− opencv2.2 documentation.
http://opencv.jp/opencv-2svn/cpp/object_detection.html

[4] Object Detection − opencv2.4.10.
http://docs.opencv.org/2.4/modules/imgproc/doc/object_detection.html

[5] opencv/contours.cpp − Itseez/opencv.
https://github.com/Itseez/opencv/blob/d8c352d20d2dba96298c26b92b7919e98e47c4c3/modules/imgproc/src/contou

[6] contours − OpenCV.
http://docs.opencv.org/3.1.0/d4/d73/tutorial_py_contours_begin.html

7　付録

7.1　画像認識

ソースコード 5　TemplateMatching.scala

```
1  package mahjongs.recognizer
2  import org.opencv.core._
3  import org.opencv.imgproc.Imgproc
4
5  object TemplateMatching {
6
7    val MaxResolution: Int = 1024 * 1024
8
9    def createTemplate(mat: Mat): Option[(IndexedSeq[Mat], Size)] = {
10     crop(resize(mat, MaxResolution)).headOption.map {
11       case (template, _) =>
12         val mask = threshold(template.clone, true)
13         val width = template.cols / 9
14         val height = template.rows / 4
15         floodFill(mask, (0 until 4).map(_ * height) :+ (mask.rows - 1), 0 until mask.cols)
16         floodFill(mask, 0 until mask.rows, (0 until 9).map(_ * width) :+ (mask.cols - 1))
17         Imgproc.dilate(mask, mask, new Mat)
18         val rects = for (tiles <- grid(mask, 4, 9)) yield {
19           for (tile <- tiles) yield {
20             val contours = findContours(tile.clone).flatMap(_.toArray)
21             if (contours.length > 0)
22               Some(convexHull(contours))
23             else
24               None
25           }
26         }
27         val size = rects.flatten.flatten.map(_.size).maxBy(_.area)
28         val tiles = grid(template, 4, 9).zip(rects).flatMap {
29           case (tiles, rects) =>
30             tiles.zip(rects).map {
31               case (tile, rect) =>
32                 Imgproc.getRectSubPix(tile, size, rect.fold(center(tile))(center), tile)
33                 tile
34             }
35         }.take(34)
36         (tiles, new Size(width, height))
37     }
38   }
39
40   def recognize(mat: Mat, templates: Seq[Mat], width: Int, height: Int): (Seq[Int], Seq[Seq[Int]]) = {
41     val result = crop(resize(mat, MaxResolution)).collect {
42       case (hand, contour) if hand.size.area > 0 =>
```

```
43          Imgproc.resize(hand, hand, new Size(hand.size.width * height / hand.size.height, height))
44          val edge = approxPoly(new MatOfPoint2f(contour.toArray: _*)).rows
45          val tiles = templates ++ templates.map(m => flip(m.t, 0)) ++ templates.map(flip(_, -1)) ++
                templates.map(m => flip(m.t, 1))
46          def go(rects: List[(Int, Rect)]): List[(Int, Rect)] = {
47            val locs = for (tile <- tiles) yield {
48              val result = new Mat
49              Imgproc.matchTemplate(hand, tile, result, Imgproc.TM_CCORR_NORMED)
50              (tile, Core.minMaxLoc(result))
51            }
52            val ((tile, loc), i) = locs.zipWithIndex.maxBy(__1._2.maxVal)
53            val rect = new Rect(loc.maxLoc, tile.size)
54            if (rects.forall(pair => !intersects(rect, pair._2))) {
55              Imgproc.rectangle(hand, rect.tl, rect.br, new Scalar(0), -1)
56              go((i % 34, rect) :: rects)
57            } else {
58              rects
59            }
60          }
61          val indices = go(Nil).sortBy(__2.x).map(__1)
62          (edge == 4 && indices.size != 4, indices)
63      }.groupBy(__1).mapValues(_.map(__2))
64      (result(true)(0), result.get(false).toList.flatten)
65    }
66
67  }
```

7.2 ユーザ定義関数

ソースコード 6　package.scala

```
1  package mahjongs
2  import scala.collection.JavaConverters._
3  import scala.collection.mutable.Buffer
4  import org.opencv.core._
5  import org.opencv.imgcodecs.Imgcodecs
6  import org.opencv.imgproc.Imgproc
7
8  package object recognizer {
9
10   for {
11     ext <- sys.props("os.name").toLowerCase match {
12       case name if name.contains("nix") => Some("so")
13       case name if name.contains("mac") => Some("dylib")
14       case _ => None
15     }
16   } System.load(getClass.getResource(s"/libopencv_java300.$ext").getPath)
17
```

```
18  def findContours(mat: Mat): Buffer[MatOfPoint] = {
19    val contours = Buffer.empty[MatOfPoint]
20    Imgproc.findContours(mat, contours.asJava, new Mat, Imgproc.RETR_EXTERNAL, Imgproc.
          CHAIN_APPROX_TC89_KCOS)
21    contours.sortBy(Imgproc.contourArea)(Ordering.Double.reverse)
22  }
23
24  def floodFill(mat: Mat, rows: Seq[Int], cols: Seq[Int]): Mat = {
25    val mask = new Mat
26    val color = new Scalar(0)
27    for (row <- rows; col <- cols if mat.get(row, col)(0) > 0)
28      Imgproc.floodFill(mat, mask, new Point(col, row), color)
29    mat
30  }
31
32  def grid(mat: Mat, rows: Int, cols: Int): IndexedSeq[IndexedSeq[Mat]] = {
33    val height = mat.rows / rows
34    val width = mat.cols / cols
35    for (row <- 0 until rows) yield {
36      val rowRange = new Range(row * height, (row + 1) * height)
37      for (col <- 0 until cols) yield
38        mat.submat(rowRange, new Range(col * width, (col + 1) * width))
39    }
40  }
41
42  def approxPoly(contour: MatOfPoint2f, epsilon: Double = 0.01): MatOfPoint2f = {
43    val curve = new MatOfPoint2f
44    Imgproc.approxPolyDP(contour, curve, Imgproc.arcLength(contour, true) * epsilon, true)
45    if (curve.rows % 2 == 0)
46      curve
47    else
48      approxPoly(contour, epsilon + 0.01)
49  }
50
51  def threshold(mat: Mat, inv: Boolean): Mat = {
52    val (tpe, op) = if (inv) (Imgproc.THRESH_BINARY_INV, Imgproc.MORPH_OPEN) else (Imgproc.
          THRESH_BINARY, Imgproc.MORPH_CLOSE)
53    Imgproc.threshold(mat, mat, 0, 255, tpe | Imgproc.THRESH_OTSU)
54    Imgproc.morphologyEx(mat, mat, op, new Mat)
55    mat
56  }
57
58  def crop(mat: Mat): Buffer[(Mat, MatOfPoint)] = {
59    for (contour <- findContours(threshold(mat.clone, false))) yield {
60      val patch = new Mat
61      val rect = Imgproc.minAreaRect(new MatOfPoint2f(contour.toArray: _*))
62      Imgproc.warpAffine(mat, patch, Imgproc.getRotationMatrix2D(rect.center, rect.angle, 1), mat.size)
63      Imgproc.getRectSubPix(patch, rect.size, rect.center, patch)
64      if (rect.angle <= -45) Core.flip(patch.t, patch, 0)
```

```
65        (patch, contour)
66      }
67    }
68
69    def convexHull(contours: Seq[Point]): Rect = {
70      val hull = new MatOfInt
71      Imgproc.convexHull(new MatOfPoint(contours: _*), hull, false)
72      Imgproc.boundingRect(new MatOfPoint(hull.toArray.map(contours): _*))
73    }
74
75    def flip(mat: Mat, code: Int): Mat = {
76      val m = new Mat
77      Core.flip(mat, m, code)
78      m
79    }
80
81    def resize(mat: Mat, max: Int): Mat = {
82      val r = math.sqrt(mat.rows * mat.cols / max.toDouble)
83      if (r > 1) Imgproc.resize(mat, mat, new Size(mat.size.width / r, mat.size.height / r))
84      mat
85    }
86
87    def center(mat: Mat): Point =
88      new Point(mat.size.width / 2, mat.size.height / 2)
89
90    def center(rect: Rect): Point =
91      new Point(rect.x + rect.width / 2, rect.y + rect.height / 2)
92
93    def toMat(bytes: Array[Byte], gray: Boolean): Mat =
94      Imgcodecs.imdecode(new MatOfByte(bytes: _*), if (gray) Imgcodecs.
            CV_LOAD_IMAGE_GRAYSCALE else Imgcodecs.CV_LOAD_IMAGE_COLOR)
95
96    def fromMat(mat: Mat): Array[Byte] = {
97      val buf = new MatOfByte
98      Imgcodecs.imencode(".png", mat, buf)
99      buf.toArray
100   }
101
102   def intersects(a: Rect, b: Rect): Boolean =
103     math.max(a.x, b.x) < math.min(a.x + a.width, b.x + b.width) && math.max(a.y, b.y) < math.min(
            a.y + a.height, b.y + b.height)
104
105   def read(filename: String, gray: Boolean): Mat =
106     Imgcodecs.imread(filename, if (gray) Imgcodecs.IMREAD_GRAYSCALE else Imgcodecs.
            IMREAD_COLOR)
107
108 }
```