

Harbin Institute of Technology

数字图像处理拼图实验报告

院 系：电子与信息工程学院

班 级： 1505204班

姓 名： 高天立

学 号： 1150520423

哈尔滨工业大学

1. 简介

一个依据拼图块自身邻接性的智能拼图算法

可以通过调整程序的特征值与相关系数算法及启发式算法中的遍历优先值评估算法来改善调整程序的行为,并可选输入相关的先验知识提升性能

* 1. 关键词

**图像拼接 启发式算法**

1. 背景

图像碎片的拼接在图像复原领域有着重要应用,比如破损的图像碎片自动拼接成一幅相对完整的图像。

基于此我决定提出一种不需要原始图像对比,对外边界没有要求,对拼图块完整性没有要求的一个误差容忍度高,适应性好,鲁棒性强并且易于拓展可根据不同场合快速调整的算法。

1. 算法
   1. 图像预处理
      1. 阀值法二值化

src为读入的原图像,img\_bw为二值化后的图像

src = im2double(imread('mona\_puzzle.jpg'));

img\_bw = ~imbinarize(rgb2gray(src), 0.99);

* + 1. 滤除干扰块

滤除检测出来的长宽值过小的区域,暂限制为10个像素点

img\_reg=regionprops(img\_bw>0,img\_bw,'all');

for i=1:size(img\_reg)

if img\_reg(i).BoundingBox(3)<10||img\_reg(i).BoundingBox(4)<10

for j=1:size(img\_reg(i).PixelList,1)

img\_bw(img\_reg(i).PixelList(j,2),img\_reg(i).PixelList(j,1))=0;

end

end

end

* + 1. 腐蚀边缘

针对与此次作业的拼图块边缘值明显偏高的情况,如果不去除不但影响检测出的边缘的特征值,还会导致拼接出来的图像邻接边缘有白边.

* + - * 1. 使用了bwperim来提取边缘

img\_bw=img\_bw-bwperim(img\_bw);

* + 1. 给各拼图块编号

使用了bwlabeln函数对各个连通域(拼图块)进行编号,其比bwlabel速度略快

使用了regionprops来计算各个区域的各种属性,后面大量的计算均基于此

lable=bwlabeln(img\_bw);

img\_reg = regionprops(lable,img\_gray,'all');

* + 1. 提取边缘

使用bwperim函数来提取一个像素宽度的边缘,并计算其属性值

img\_edge\_mask=bwperim(img\_bw);

img\_edge\_rgb=img\_edge\_mask(:,:,[1,1,1]).\*src;

img\_edge\_gray=rgb2gray(img\_edge\_rgb);

img\_edge\_lable=lable.\*img\_edge\_mask;

img\_edge\_reg = regionprops(img\_edge\_lable,img\_edge\_gray,'all')

* + 1. 绘制

绘制提取出来的拼图块及其编号,以及后续计算出来的四个角点,方便校验及后续编程.其中的findCorner函数为另外边写的寻找角点的函数,会在后续提到.

rects = cat(1, img\_reg.BoundingBox);

centroids=cat(1,img\_reg.Centroid);

corners=findCorner(img\_reg);

figure();imshow(img\_bw);

hold on;

% plot(centroids(:,1), centroids(:,2), 'r\*'),

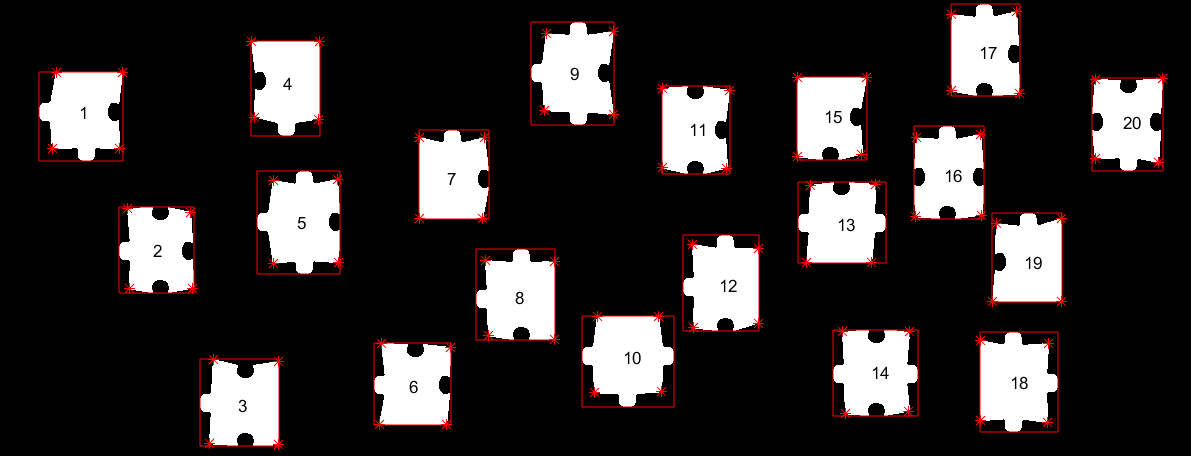
plot(corners(:,:,1), corners(:,:,2), 'r\*'),

for i = 1:size(rects, 1)

rectangle('position', rects(i, :), 'EdgeColor', 'r');

text(centroids(i,1)-5, centroids(i,2),num2str(i));

end



* 1. 计算边缘代表的特征值

这是一个由用户根据其采用的算法而编写的函数, 可对每个像素点的领域进行运算从而得到这一点的特征值,程序之负责查询,可方便快速地修改以提升程序性能,为了体现此算法的适普性,我们直接使用了像素点对应的灰度值作为特征值.

* 1. 寻找四个角点
     1. 描述

计算并返回所有拼图块的四个角点的坐标矩阵

* + 1. 算法

这里采用的是检测坐标的形式,例如一点的横纵坐标之和最大,那可认定其为偏离原点最远的点,即为右下方的角点.

先计算出其特征值矩阵(坐标值的加减来衡量),再用max提取出最优解的下标位置记为index,提取后存入corners矩阵

* + 1. 输出

因为这些是内核函数,可能会被多次调用,故牺牲可读性未使用结构体来存储,而使用了3维矩阵,分别记录着拼图块编号,边角的序号,坐标值

在此程序中有20个拼图块,每个拼图块有四个边角,故为20x4x2的矩阵

* + 1. 程序

%

% from the top-left and go through clockwise

% 1%%%%2

% %%%%%

% %%%%%

% %%%%%

% 4%%%%3

% note :

% the return Value is coordinates not index!

%

function corners = findCorner(img\_reg)

corners=zeros(size(img\_reg,1),4,2);

for i=1:size(img\_reg,1)

% left-top

[~,index]=max(-img\_reg(i).PixelList(:,1)-img\_reg(i).PixelList(:,2));

corners(i,1,1)=img\_reg(i).PixelList(index,1);

corners(i,1,2)=img\_reg(i).PixelList(index,2);

% top-right

[~,index]=max(img\_reg(i).PixelList(:,1)-img\_reg(i).PixelList(:,2));

corners(i,2,1)=img\_reg(i).PixelList(index,1);

corners(i,2,2)=img\_reg(i).PixelList(index,2);

% right-buttom

[~,index]=max(img\_reg(i).PixelList(:,1)+img\_reg(i).PixelList(:,2));

corners(i,3,1)=img\_reg(i).PixelList(index,1);

corners(i,3,2)=img\_reg(i).PixelList(index,2);

% buttom-left

[~,index]=max(-img\_reg(i).PixelList(:,1)+img\_reg(i).PixelList(:,2));

corners(i,4,1)=img\_reg(i).PixelList(index,1);

corners(i,4,2)=img\_reg(i).PixelList(index,2);

end

end

* 1. 依据角点把边缘像素点分成四块
     1. 描述

输入每块拼图块的边缘,并用边角点分割为上下左右四块边界

* + 1. 算法

采用了检测每个像素点到上下左右四个方向的每两个对应的边角点距离之和的方法来评定其归属,例如一点到左上和右上角两个顶点的距离之和最小,则认定其属于上边界

同样采用了向量化编程思想,用distance记录了每个像素点到各个角点的距离,用distance2记录了每个角点到每条边的特征值(对应两边角点的距离之和),再用min找出最优解,确定其归属,最后存入对应边界的矩阵中

* + 1. 输出

因为每个拼图块每条边的像素个数均不同,故使用了cell结构体来存储

共2维的cell,分别为拼图块序号,边缘序号,以及每个cell里包含的2维矩阵,分别为像素点索引号与像素点横纵坐标值及此像素点存储的(特征)值

* + 1. 程序

% the mapping num of the four direction

%from the top and go through clockwise

%

% %%1 %%

% %%%%%

% 4 %%% 2

% %%%%%

% %%3 %%

%

%Description :

% From the img\_reg(calcated by fun regionprops)

% we divide each region's pixel into four direction groups by their

% distance to corners

%

% example:

% if this pixel has min sum distance to corner1(top-left) and 2(top-right)

% then it's classified into the upper group

%

% Struct :

% pixelGroups contain pixels of each direction within each region

% and the pixel elements contain the coordinates and the Property Value

% To extract :

% pixelGroups{regionNum}{directionNum}[x,y,value]

%

function pixelGroups = divisionByCorner(img\_reg)

%from the top and go through clockwise

corners = findCorner(img\_reg);

pixelGroups = cell(size(img\_reg,1),1);

pixelValues = {img\_reg.PixelValues};

pixelLists = {img\_reg.PixelList};

for i=1:size(img\_reg,1)

distance2=zeros(0);

distance=(pixelLists{i}(:,1)-corners(i,:,1)).^2+(pixelLists{i}(:,2)-corners(i,:,2)).^2;

distance2(:,1)=distance(:,1)+distance(:,2);

distance2(:,2)=distance(:,2)+distance(:,3);

distance2(:,3)=distance(:,3)+distance(:,4);

distance2(:,4)=distance(:,4)+distance(:,1);

[~,groupIndex]=min(distance2,[],2);

upPixels=ones(0);

rightPixels=ones(0);

bottomPixels=ones(0);

leftPixels=ones(0);

for j=1:size(img\_reg(i).PixelList,1)

switch(groupIndex(j))

case 1

upPixels=[upPixels;[pixelLists{i}(j,1),pixelLists{i}(j,2)],pixelValues{i}(j)];

case 2

rightPixels=[rightPixels;[pixelLists{i}(j,1),pixelLists{i}(j,2)],pixelValues{i}(j)];

case 3

bottomPixels=[bottomPixels;[pixelLists{i}(j,1),pixelLists{i}(j,2)],pixelValues{i}(j)];

case 4

leftPixels=[leftPixels;[pixelLists{i}(j,1),pixelLists{i}(j,2)],pixelValues{i}(j)];

end

end

pixelGroups{i}={upPixels,rightPixels,bottomPixels,leftPixels};

end

end

* 1. 计算每个拼图块的每块边缘与其它拼图块的对应边缘的关系
     1. 描述

例如一块拼图块的上边缘,计算出其与其它所有拼图块的下边缘的相关性,(如考虑旋转,则计算所有边缘)最终的结果保存在矩阵中输出.

日后考虑根据不同的先验知识提供不同的模式,如已知只有外边缘为直线则检测出来为直线的边与其他所有边的相关度均置零

* + 1. 算法

由前面计算出的pixelGroups中提取出选定的边缘(的所有像素点的坐标)以及其他拼图块的对应边缘,通过角点计算偏移量,平移过去之后遍历一条边缘的像素点,通过子函数findNearbyPixel寻找附近的有值点,并将两点对应的特征值送入用户定义的calcRelevance函数,得到相关度.

* + 1. 结构

3维矩阵,分别为拼图块的序号,边缘的序号,与其他的拼图块的序号,

* + 1. 程序

因长度过长显示效果太差故不在此全部展示,详情可见github

% mappingdata = calcMappingdata(img\_reg)

% mappingdata = calcMappingdata(img\_reg,img\_property)

%mappingdata=calcMappingdata(img\_reg,img\_property,calcRelevanceMode)

%mappingdata=calcMappingdata(img\_reg,img\_property,calcRelevanceMode,searchRange)

%核心循环:

dx=corners(m,mod(j-2,4)+1,1)-corners(i,j,1);

dy=corners(m,mod(j-2,4)+1,2)-corners(i,j,2);

[correspondingPoints,ismatchedMatrix]=findNearbyPixel([edgePixelGroups{i,j}(:,1)+dx,edgePixelGroups{i,j}(:,2)+dy],edgePixelGroups{m,n},searchRange);

for k=1:size(edgePixelGroups{i,j},1)

propertyValue1(k,:)=img\_property(edgePixelGroups{i,j}(k,2),edgePixelGroups{i,j}(k,1),:);

propertyValue2(k,:)=img\_property(correspondingPoints(k,2),correspondingPoints(k,1),:);

end

mappingdata(i,j,m)=calcRelevance(propertyValue1,propertyValue2,ismatchedMatrix,calcRelevanceMode);

* 1. 计算相关性
     1. 描述

面向用户的一个函数,输入为两组相同大小的像素点对应的(归一化的)特征值矩阵,以及一个边缘各点是否匹配成功的矩阵,可快速修相关性的计算方式以调整程序行为,内置了平衡模式和重边缘模式

* + 1. 程序

%

% Description :

% v1,v2 are supposed to be scaled value

% given two (list of) Property Value and calc the relevanceValue

% Matrix input are supported

%

% could be modified to achieve better performance

function relevanceValue = calcRelevance(v1,v2,ismatchedMatrix,mode)

if nargin<4

mode='balance';

elseif nargin<3

ismatchedMatrix=[];

end

if ~ismember(mode,{'balance','edgeDominant'})

disp('invalid mode option, balance mode was used automatically');

mode='balance';

end

switch mode

case 'balance'

% relevanceValueArray=1-(v1-v2).^2;

relevanceValueArray=1-(v1-v2).^2./((v1+0.0001).^2+v2.^2); %in case v1==v1==0

relevanceValue=mean(mean(relevanceValueArray));

case 'edgeDominant'

relevanceValueArray=1-(v1-v2).^2;

if ~isempty(ismatchedMatrix)

ismatchedMatrix=ismatchedMatrix(:,ones(size(v1,2),1));

relevanceValueArray=relevanceValueArray.\*ismatchedMatrix;

else

disp('waring: ismatchedMatrix is empty edge detection enhance aborted');

end

relevanceValue=mean(mean(relevanceValueArray));

otherwise

end

end

* 1. 计算拼图块位置
     1. 描述

由前面得到的每个拼图块每条边与其他所有拼图块的对应边的相关性,我们可以找到一个最优的位置分配矩阵,使整幅拼图的平均相关性最大

* + 1. 算法

因为直接遍历所有情况的计算量非常巨大,故采用了启发式算法来改进,该算法对所有的待搜索状态做一个评定,其中越接近目标状态的评级越高,搜索的优先级也越高

所以我们的主循环就是每次取出待搜索队列中优先级最高的一种情况,并在其基础上做延拓(在边缘增加一个拼图块),得出所有可能的下一步拼接矩阵,把无重复的计算优先级后加入待搜索矩阵,同时删除以搜索的这一项,完成一次循环

循环终止条件为取出的优先级最高的矩阵已拥有所有的拼图块,此时结束循环并返回此最优解矩阵

由于此算法优先拓展当前的最优解,故当搜索优先值评定合理时,其可快速逼近最终的最优解,相比于广度优先算法减少了大量的搜索次数,同时也保证了在最坏情况下其也能遍历所有状态,具有完备性

* + - 1. 关于评定状态并返回搜索优先值

这是这个环节的启发式算法的灵魂,如何评定哪种矩阵是更接近最优解的矩阵决定了程序的搜索行为,所谓的相对于广度优先算法的启发量即为此,此函数可由用户自行修改调整参数,增加限制条件以提升性能

这里我采用了三个量来衡量这种状态的好坏,一种是整个矩阵各个已存在的拼图块的平均相关性,很容易理解其平均相关性越好,其状态越有价值,一种是遍历深度,即此位置矩阵使用的拼图块的数目,越小值越大,还有一种是位置矩阵中的有效长宽,长宽和越小状态越优,调整其相互之间的权重系数可以调整算法的行为

相关性的系数越高,即额外的启发量越大,算法就越趋向于最短时间找到解,然而容易受干扰,比如左上角的左边缘与右上角的右边缘其形状基本相同,颜色也非常相似,两边缘的相关性非常大,故算法便会依据此情况进行下一步的拓展.当边缘形状或颜色有较大区别及图像噪声较小时,可加大此项的权重以最快逼近最优解

而拼图块使用的数目的权重代表的遍历的深度,增加其权重可限制拓展深度,有效避免因单条边缘的相关性极大而造成的误导,使算法趋向于整体相关性最优解,当其值远大于其他值时,算法退化为广度优先算法.

而矩阵中的有效长宽的和则限制了其延横/纵单方向发展的情况,控制了其拓展的顺序,使其趋向于横纵同时拓展,各拼图块位置分布紧挨,此时若一新拼图块的周围不止一个拼图块,则有多条边需要校验,可有效避免单条边由各种因素造成的干扰,提升平均相关性的效果.当拼图块有较多残缺时,应适当减小该值

此外为了提升对边缘干扰的抗干扰能力,加快搜索速度,增加了个可选的完成图的宽高的输入量,当现有的位置分配矩阵超维时,相关性直接置零.

* + - 1. 关于提取出最高优先值的方法

在一组非常大的数组中寻找到一个最大值,最容易想到的就是使用matlab自带的max函数,其需要遍历整个队列来寻找,即出队列的时间复杂度为N,而每次把新元素直接附在队列末尾,入队列的时间复杂度为1

第二种方法是使用有序数组来存储,每次入队操作时都对数组重新排序,而提取时只需要提取出第一位的元素即可,但其时间复杂度相对于上面的无需数组并未有提升

第三种方法是使用完全二叉树来存储,(也称为霍夫曼编码树),其保证了每个父节点的值都大于子节点,故根节点必为最大值,其入队列和出队列的时间复杂度均为logN,在N较大时有可观的提升

这里由于其优化效果有限优化优先级不高,我们暂时使用了第一种方法,待后续更新

* + - 1. 关于查重算法

为了避免做重复的延拓,在把一种状态加入待搜索队列前对其进行查重是很有必要的,而如果直接用”==”号去比较这个位置分配矩阵是否和待搜索队列里的每个位置分配矩阵相等的话性能开销非常严重,因为每个位置分配矩阵都包含有较多的元素,而之前那样是把对应的元素提取出来一一对比.

为了优化此过程,我们采用了哈希表的思想,即为每一个矩阵计算出一个对应的哈希值,这个值应满足单映射的关系,所以我们只要对比新矩阵的哈希值是否已存在即可,就像是我们判断两个人是否为同一个人不对其每个元素作比较而依据其指纹来判断,在这里,我们采用了平均相关性来作为其哈希值.

而想上面采用完全二叉树来优化寻找最大值的过程一样,我们也可以采用排序二叉树来判断新矩阵的哈希值是否已存在,待后续更新

* + 1. 接口

% allocateMatrix = calcRegionAllocation(mappingdata)

% allocateMatrix = calcRegionAllocation(mappingdata,allocationSize)

* + 1. 程序

仅展示计算搜索优先值的核心部分, allocationSize是可选的输入量,代表完成图的宽高,当前的个因素权重为临时赋予的

if ~isempty(allocationSize)

if width<=allocationSize(1)&&hight<=allocationSize(2)

priority=averageRelevance;

else

priority=0;

end

else

priority=666\*averageRelevance-regionCounts-(width+hight).^2;

end

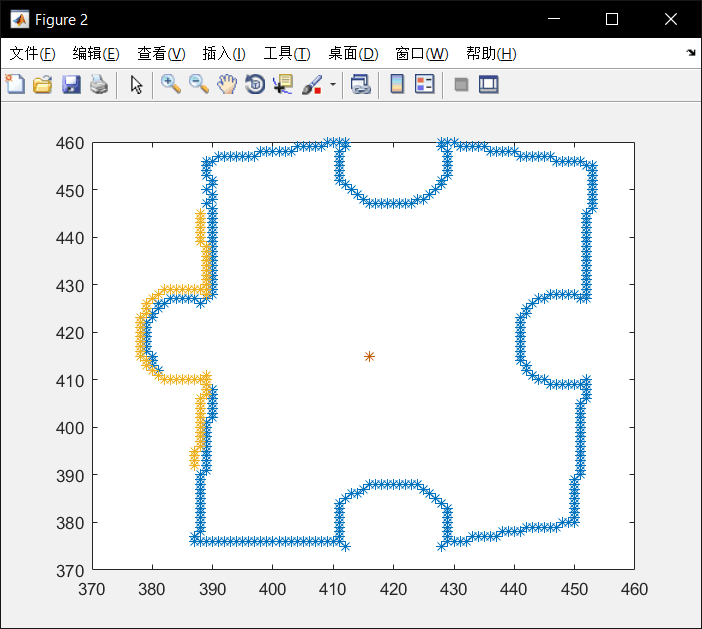
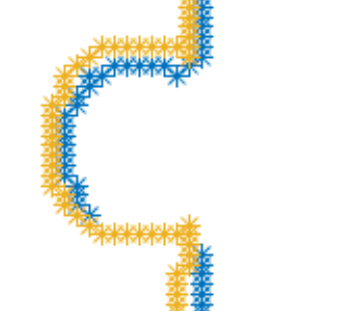
* 1. 拼接
     1. 描述

由上个函数返回的allocateMatrix,我们知道了各个拼图块的邻接对象,通过其角点我们可以得到一个大致的位移量,同样本着寻找最优解的思想,我们在由边角得到的位移量的基础上进行小范围的遍历,并对每种状态做衡量得到最优解.

* + 1. 算法

当前是提取待拼入的拼图块的对应的边缘与当前已拼上的拼图的边缘作比较,比如下图的黄色为提取出的待拼入的边缘,蓝色为已存在的图像的边缘,我们将黄色的所有像素点坐标往右平移一个单位,记录其与蓝色边缘像素点重合的数目,将其对黄色边缘的长度做归一化后得到对当前状态的衡量值

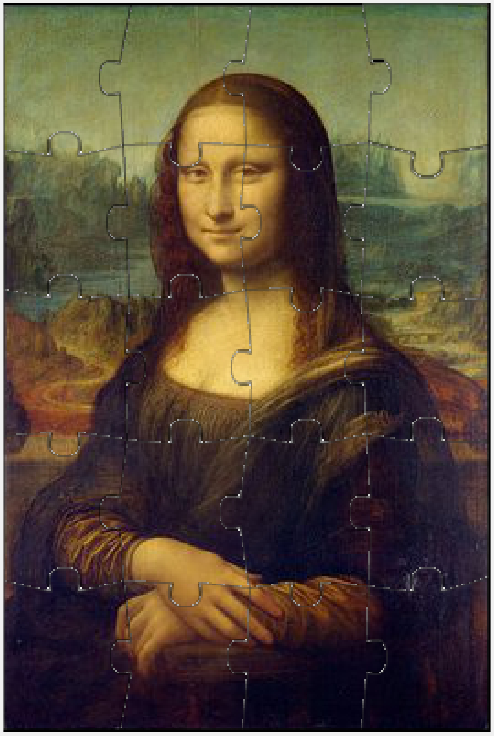
但当前的算法效果并不如直接用角点确定位移量的拼接算法,因为次图像拼图块与矩形相似度高,角点提取准确干扰小,及已经为最优解,而且按照之前的衡量算法,即黄色右边的像素点与蓝色的重合程度,则会造成上下的偏移,即其衡量出来的最优解并非为我们需要的,待后续改进(当前把搜索范围退化为0即用角点也能取得不错的效果)

* + 1. 程序

同以上的其他未展示的程序,可以在我的github里找到完整的最新的源代码与效果图.

1. 优点
   1. 提供了三个面向用户可令其快速调调整程序行为适应不同情况需求的函数,分别为计算每个点特征值的方式,由两边缘各点的特征值计算两边缘相关性的方式,以及搜索时当前状态的评定方式.
   2. 不依赖拼图块之间的边缘形状;不依赖外边缘的形状; 不依赖拼图块的完整性
   3. 如有先验知识,比如知道完成时拼图块横竖的块数,以及外边缘为直线而内边缘非直线等情况可以输入额外参数,加速运算
   4. 编程较为规范,内核函数均使用了向量化编程以适应matlab的特点,函数名变量名取名直观且采用了驼峰命名法,函数接口多样,可根据需要指定已有的合适参数
2. 缺点
   1. 计算量还是过大
   2. 基于拼图块是矩形的假设,依赖四个边角点,以及把边缘分成了四条边来检测
   3. 目前由于精力有限仍存在大量待优化项目
      1. calcMappingData那里增加模式的选择,可选直线边缘模式,圆形边缘模式等,以及当前的普通模式,以提高对不同边缘的适应性
      2. 确定拼图块位置的calcAllocationMatrix函数使用C++编写,编译后链接到matlab里
      3. 在最后一步的generateSplicingImg中采用效果更好的评估边缘重合度的算法
      4. 最终计算拼接图时因遍历多种状态而造成内存开销过大,在拼接32位图像时发生过因内存不足而终止运行的现象
3. 展示
   1. 完整效果图



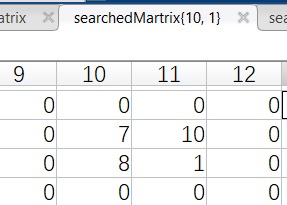
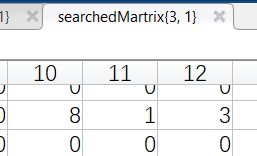
* 1. 残缺拼图块效果图
     1. 简介

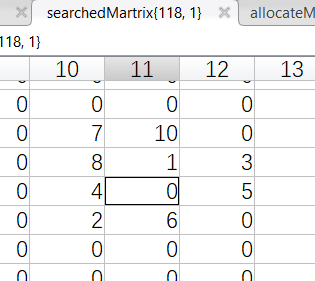
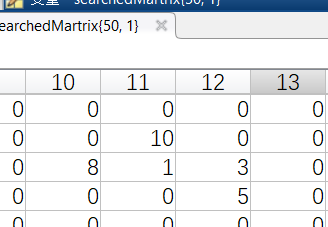
此算法对不完整的边缘与空缺的拼图块有着良好的适应性,以当前情况举例,若已知最终尺寸是限制在3x4的矩形框内,而有10块拼图块,若采用广度优先算法遍历则共约为2.4亿种可能性,而我们的算法只提取搜索了118种状态,产生了1799种拓展态即完成了最优解的搜索(整个程序合计60s,而如果边缘等干扰严重,搜索状态与时间将指数增加)



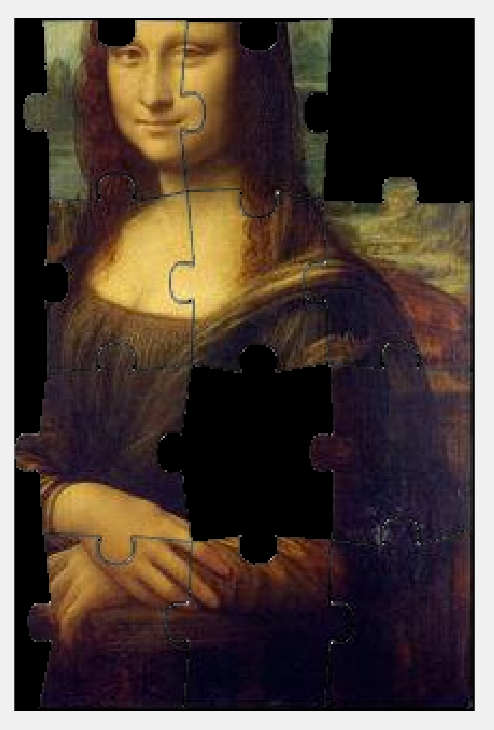
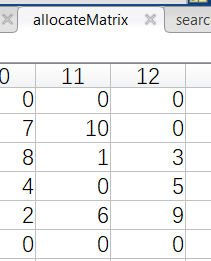
* + 1. 详情:

因为有十块拼图块,故产生了21x21的矩阵来容纳,其中心点放置着拼图块1

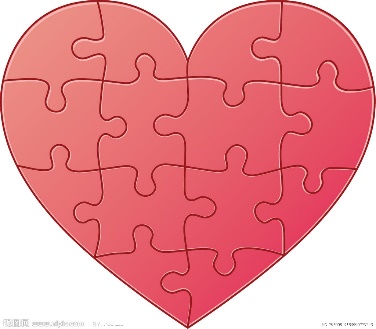




上图分别为第3,10,50,118次从待搜索队列中提取出来的位置分配矩阵,其中第50次的左上角并未赋值,说明其并非由状态10拓展而来,(最终解从searchMatrix中提取出来后因无需拓展,并未记入searchedMatrix,故第118个位置矩阵右下角残缺,并非最终解)最终输出的位置分配矩阵与拼接完成图如下

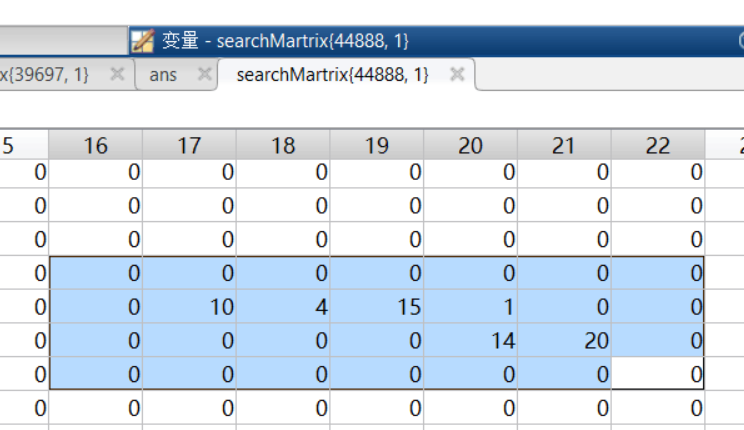


* 1. 其它拼图

这代表了几种当前难以有效处理的拼图类型,分别为外边缘形状相似,内边缘颜色差异大不平滑造成较严重的干扰,拼图块不类似矩形等情况,理论上应能存在并能得到最优解,但因干扰严重,运算量极大

比如在未限制矩阵有效大小时,左右边缘容易相互拼接造成大量的无效运算



当前处理效果较好运算较快的还是边缘差异大,拼图块形状类似矩形有较为明显的四个边角点可以提取的图像碎片

1. 参考资料
   1. [拼图游戏和它的AI算法](https://www.jianshu.com/p/10a6d02616ad)
   2. [完全二叉树实现优先队列与堆排序](https://www.jianshu.com/p/9a456d1b59b5)
2. 备注
   1. 故若需试运行分析程序,建议运行残缺拼图块的拼图(puzzle\_partVersion.m)
   2. [程序源代码及后续更新地址:](https://github.com/G-T-L/Puzzle)

https://github.com/G-T-L/Puzzle