“人工智能基础”课程大作业

项目报告

项目名称：QTangram 七巧板求解

姓名：刘国子

学号：2017011883

班级：自 81

日期：2019.10.19

目　录

1 平台功能设计

1.1 任务描述与需求分析

七巧板是我国古代劳动人民智慧的结晶，也是锻炼手脑和丰富想象的益智玩具，它的关键玩法和有趣之处在于用碎片的元件来拼接出整体的图案，这种图形想象和实践的能力是人类所具有的一种“高级智能”，这次作业则需要我们创造出模仿人类这种能力的人工智能，具有很强的挑战性和趣味性。

总体任务可以高度概括为，输入一张黑白二值的图像，利用搜索的方法，输出怎样用七巧板拼出该图形的解。选做任务要求能够搜索更复杂的巧板并能够拼接出非巧板构成的图形。图像的输入方式可以从图库选择、自己定义或打开文件，将图像存入内存后进行搜索，最后展示搜索结果。

另外，该问题隐含的一些要求在于：

①搜索的时间或空间成本不能太高

②能够自适应输入图像中不同的七巧板尺寸

③能够对输入图像的躁点、图形缺陷（如缺角或少尺寸）有一定鲁棒性

1.2 总体功能描述

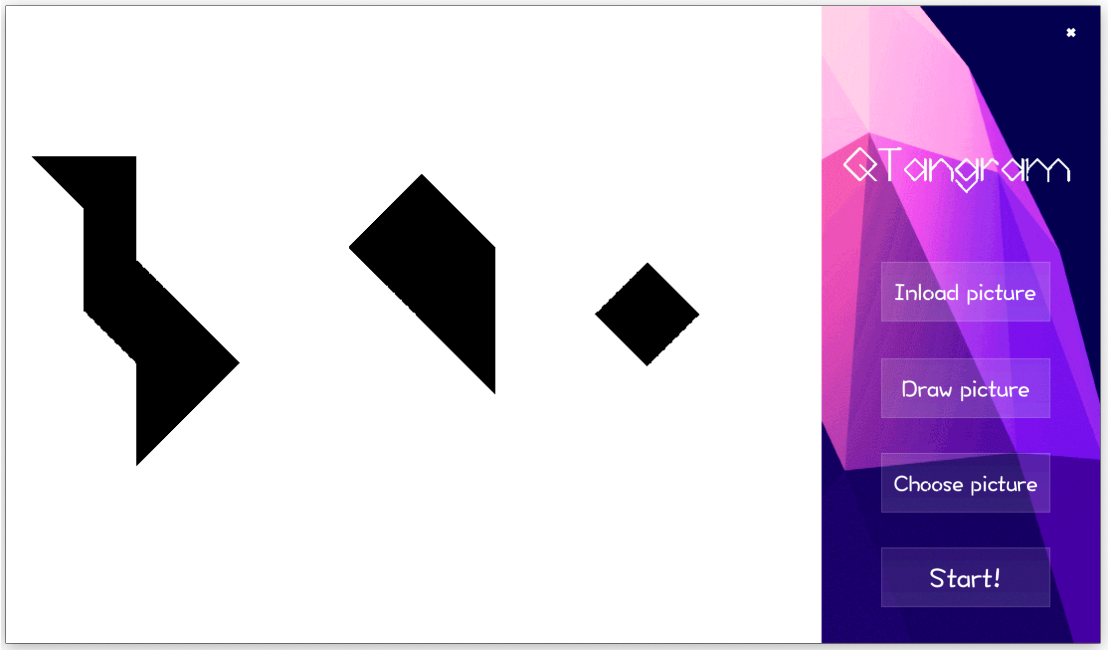
笔者开发的 QTangram 程序实现了搜索七巧板解法的智能，允许用户输入图像文件进行求解、从图库中选择图像进行求解、并实现了模拟现实中七巧板拼接的功能，用户可以自己拖动、旋转元件创造出图形，并交给程序进行搜索求解，用户还能将自己的作品保存为图像文件存储在硬盘上，程序还可以将它探索某个图的搜索过程生成图像组进行存储，用户根据这些图像可以直观地看到人工智能在求解搜索问题时的思路。

由于能力所限，笔者并没有完成选做任务，程序只能搜索七巧板拼接出的图形。但笔者在必做的基础上加入了很多创新功能，如用户可以自己拼接图形并保存、程序支持多种图像格式输入、保存搜索过程的图片等等。

1.3 ***使用说明与功能流程

本着“用户体验至上”的原则，笔者设计的 QTangram 程序秉承人机交互极简主义的原则，只留下最核心的功能模块，界面完全“傻瓜式”，尽最大可能保证用户“一看到，就会用”，但在一些细节方面笔者有必要做出一些使用说明如下介绍：

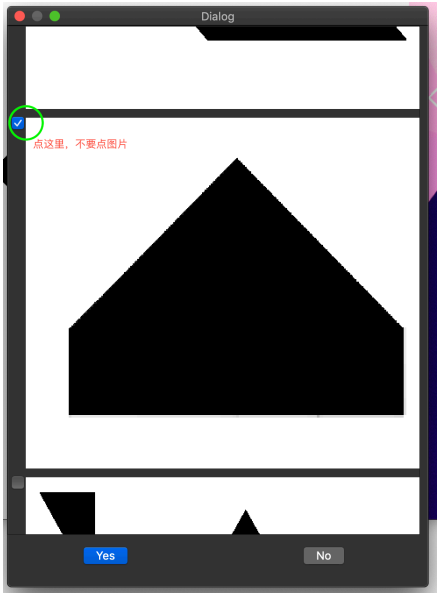
1.3.1 主界面介绍



如图，主程序的设计十分简洁，界面左边占大部分的白色区域用来显示图像和拼接结果，并且用户也可以在该区域内创作图形；右边紫色区域为功能按钮，共有四个 button: Inload picture , Draw picture , Choose picture , Start! 其中的部分功能如下：

① Start!（开始求解）：点击后，程序开始自动求解左边区域的拼接方法，介时程序可能会处于卡顿状态无法响应用户输入，在求解完毕后左边区域会显示求解结果，若求解失败，则会显示程序试图求解的最终结果。求解的时长因输入图像而异，时长范围在顷刻间到 2min 不等，笔者测试 80%的图像可以在半分钟以内得解。

② Choose picture（从图库中选择图片）：点击后，弹出图库选择对话框，用户在勾选某个图像前的 checkbox 后点击”Yes”可以将图库图像导入左侧展示区。



③ Inload picture（输入图像文件）：点击后，自动打开系统文件目录，用户可以选择一张硬盘中的图像文件放入左侧展示区，并要求程序求解，需要注意，输入图像的要求如下：

(i) 输入图片格式支持*.png *.bmp *.jpg 三种

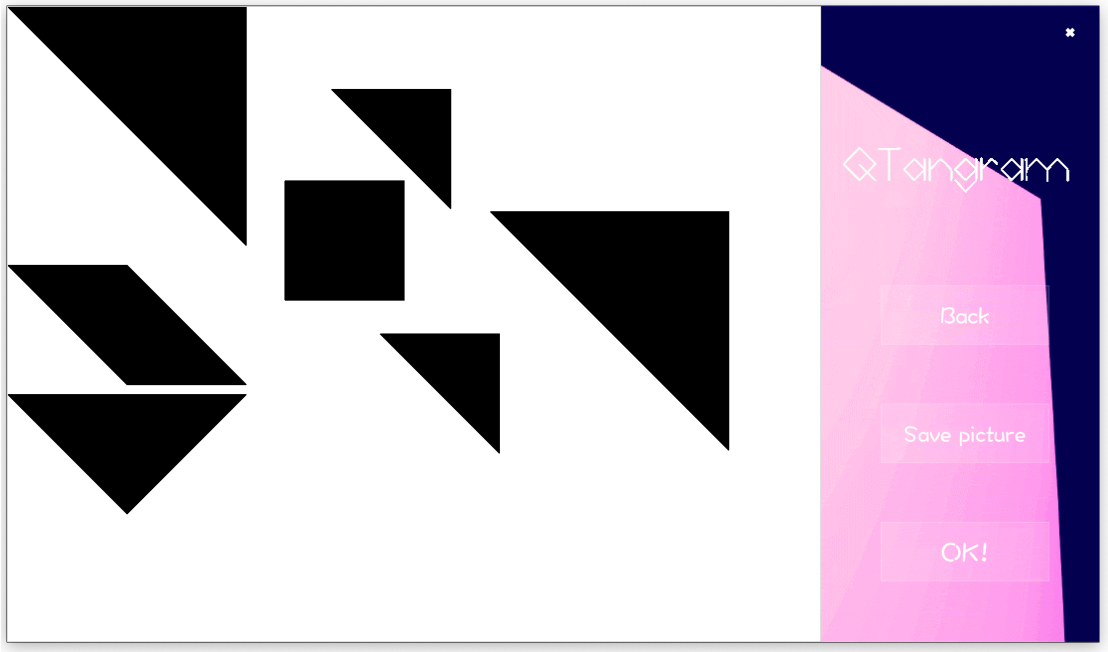
(ii) 图片需为黑白两色，虽然算法对躁点和非纯黑/白点有一定鲁棒性，但如果大面积有效图形都是其他颜色（如红色）则可能导致算法无法正确求解。

(iii) 图片中黑色待求解区域不能变形或损伤得太离谱，算法对边缘毛刺和个别图形元件的尺寸不合适有一定鲁棒性，但如果原图损伤较为严重，或各元件之间比例明显不协调则可能导致算法无法正确求解。

(iv) 图片不能用 sRGB IEC61966-2.1 渲染！！ 这是由于开发软件 QT 自身图像处理时 QImage 对象无法正确读取这类文件所致，如果非要拼接这种图像中的图形，烦请在 PS 中打开图像->导出为 web 通用->取消勾选“采用 sRGB IEC61966-2.1”->另存为即可。

1.3.2 用户拼图界面介绍

按下 Draw picture 后，会自动进入拼图创作状态，如下图所示：



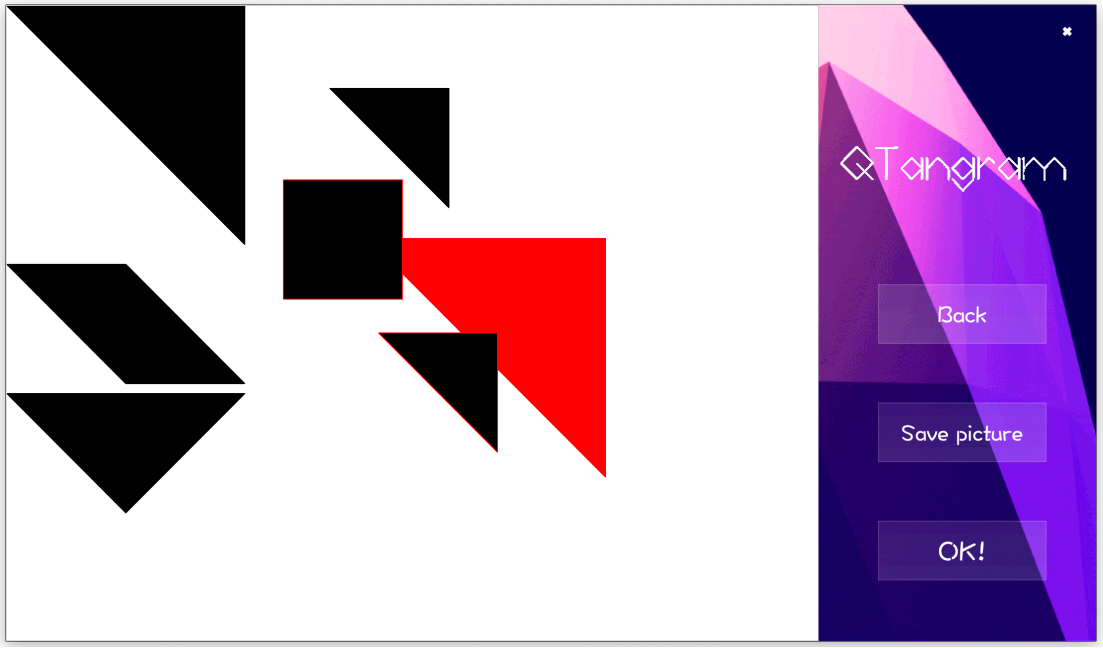
此时，左边展示区的黑色元件全部变成可活动的，用户只要按照下面的几条规则创作自己的作品，就可以交由程序求解：

(i) 可以鼠标**左键拖动**元件，**右键**点击可以**旋转**元件改变其姿态

(ii) 创作满意后，点击 Save picture 可以选择将作品存储在硬盘上的某个文件夹内，或点击 OK!直接将您的作品返回搜索求解模式，也可以点击 Back 暂时离开拼图创作状态（相应的图像会暂存，用户下次进入拼图创作状态时还是上次离开时的状态）

(iii) 在拖动过程中，请**不要把元件拖到幕布外面，也不要让任何元件的任何部位超出白色展示区的展示范围**，否则可能会导致搜索失败（这是由于笔者在做越界检测时出现未知 bug，出于时间关系没有完成理想的越界检测，敬请谅解），如果您不小心将某个元件拖出白色展示区，烦请重新运行程序

(iv) 碰撞检测：当元件发生碰撞或相互遮挡时，相关的元件会被标红，即将碰撞时，会用红色警戒线显示相关元件的轮廓，请务必**保证最后左侧展示区内没有任何红色（图形轮廓、图形本身都不可是红色的）后再进入搜索求解模式**。有红色轮廓提示线的帮助，您很容易拼接出元件之间不相互遮挡而又紧密贴合的图形。下图为碰撞和遮挡检测示意图：



1.3.3 观察搜索求解过程

由于搜索过程中的状态可能很多，变化范围很大（从几步到上千步不等），且生成中间状态的图形会大大影响搜索速度，因此可执行程序中并没有留出让用户观察每一步搜索过程的接口，如果您想观察其搜索过程，烦请进入源代码 picture.cpp 文件中，将 592 行 savePlace 变量中的第一段路径改为您想要保存的位置，并取消 595 行的注释，重新编译运行后程序即可将搜索过程的每一步图形生成到您想要的位置。

```
590
591
592     QString savePlace ="/Users/tnoy/Desktop/n3/"  + QString::number(printNu
593     printNum++;
594
595     //newImage->save(savePlace);
596
```

2 问题建模与算法逻辑

2.1 问题建模

问题本质是在一张二维像素数据上进行搜索的过程，QTangram 程序对这一问题的建模方式十分简洁明了，为了节约内存开销并加速像素点的随机访问，程序首先将读入的图片等长宽转换成一张 bool[][]二维矩阵，若某个坐标对应原图中点为黑色则设为 1，白色则设为 0.由于一个 bool 量只占到 1 个 bit 位，因此这是不压缩图像的情况下最节约空间的表示方式，另外二维 bool 阵采用基址变址寻址的像素访问在计算机内存中处理速度很快，方便了对图像每个像素的随机访问。

另外，为了算法的可扩展性，程序进行搜索的思路偏向于纯图像访问，而非检测图形边界、顶点再用几何的方法搜索，因为那种思路较局限于七巧板搜索这一具体问题，可扩展性不高。

程序搜索时判定某个元件能否放入某个位置的方法是根据该元件的姿态找到其轮廓的数学表达，再判断原 bool 阵在这一轮廓范围内有没有足够多的 ”true”（即黑色，下统一称“黑点”）。因此理论上可以检测任意形状的元件能否放入某个位置，增加了算法的可扩展性。

因此对于该问题，只要每一块元件都找到其轮廓范围内有足够多黑点的位置，那么求解就算完成。

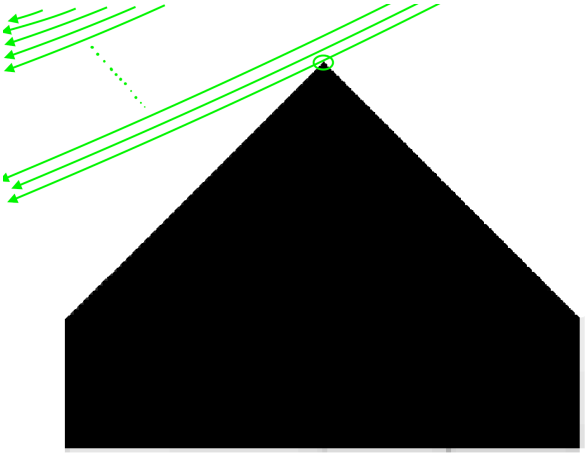
2.2 搜索算法介绍

可以先在 3.1 部分浏览一下算法逻辑的关键类 UML 图再来看这个部分。

2.2.1 算法大体框架

这一部分用简洁的语言介绍算法的主要思路。

算法性质上属于深度优先搜索，大体的思路为：



如上图，在将原图转化为一个大型二维 bool 阵后，按图中绿色平行线所示的方向对该 bool 阵进行扫描，在扫描到黑点并确认该点不是图像噪点后，尝试“放入”元件，即检查元件轮廓范围内有无足够多的黑点，程序事先把以这种方式扫描扫到元件顶点后元件种类和姿态的所有情况进行建模并存储，并在扫到黑点后判定哪个元件以哪种姿态可以放入以当前扫到黑点为顶点的区域内，判定成功后，将 bool 阵中该元件轮廓区域内的值等效地改为”false”即白色（注意不是真的改原 bool 阵，而是等效修改），然后继续扫描新的图像并重复这一步骤。

如果最终 7 块元件全部找到了合适的位置，则搜索成功并退出，如果扫描到某个点后，所有当前剩余元件以任何姿态都无法放入，说明之前元件的放置不对（亦有可能是图像自身问题），则拆除之前的一步放置，在放置上一步之前的状态下探索其他的放置方法。因此本质上该搜索可以分类为深度优先搜索。

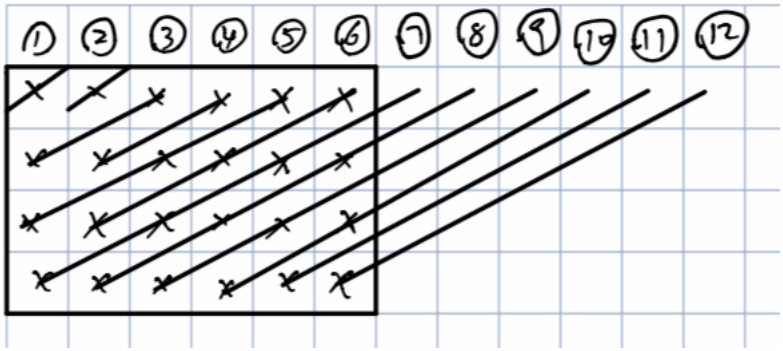
程序采用面向对象的思维编程，搜索算法的核心类只有 2 个：

Picture 类在每次输入待求解图像后生成，用于存放原图生成的 bool 阵以及当前有哪些元件放在了哪个位置（即搜索状态）。

Shape 类用于建模每块 7 巧板元件，并确定其位置、划定它们轮廓的数学表示。

2.2.2 算法细节描述

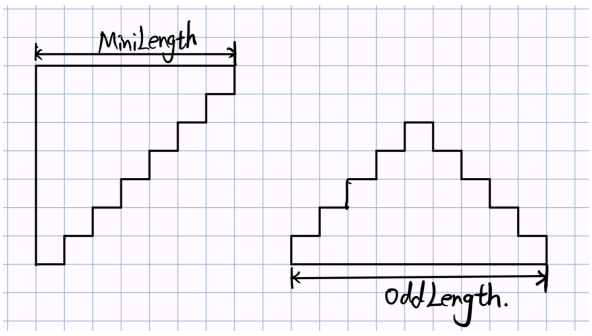
①扫描过程



如上图所示，在扫描一个规模为 m*n 的 bool 矩形阵时，用斜率为 1/2 的直线进行移动扫描，这种方式可以扫到矩阵中的每一个点，扫描的总次数为 2*(m-1)+n，在加上边界条件的判定，扫描的总体复杂度为$O(m*n)$，是遍历一个矩阵所能达到复杂度的下限。此外，由于 7 巧板所有元件顶角都为 45 度及其倍数，且旋转最小单元也为 45 度，因此扫描线不会与图形边界线重合，使得扫描只会扫到图形顶点。

②尺寸自适应

程序需要根据输入图片中巧板总面积来确定每一块的尺寸，而由于数字图像像素存储的离散性，无法做到在图形旋转时保持每条边像素点数一致，因此整个程序需要两个指标来唯一确定所有的元件大小，在程序中设计的变量为 MiniLength 和 OddLength 分别代表最小三角形直角边平放时的直角边长和斜边平放时的斜边长：



二者之间的关系约为：（理论上是$\sqrt{2}$倍）

$$OddLength = 1.4 \times MiniLength$$

根据图像中黑色待解区域的总面积（黑色块数）可以确定以上两个尺寸，而七巧板中所有元件都可以通过小三角形拼接而成，因而这两个尺寸确定后每个元件以任意姿态摆放的具体尺寸也都确定了。

通过图形总面积得到尺寸的关系为：（中括号为向下取整）

$$MiniLength = \left[\sqrt{\frac{BlackArea}{8}}\right] + 2$$

$$OddLength = \left[\sqrt{\frac{BlackArea}{4}}\right] + 2$$

每个尺寸+2 来平衡取整的造成的损失，同时使元件微微膨胀（这可以增强算法的鲁棒性）

③状态及其转换：

程序的每个 Picture 对象代表一张待求解的图，其中有通过原图生成的二维 bool 阵，以及一个存放 Shape 对象的容器，代表当前图上都放有哪些元件，Shape 对象自身包含元件种类、元件位置、元件姿态信息，因此搜索过程中的状态其实就是 Shape 容器的状态，具体归纳如下：

(i) 初始状态：Shape 容器为空，代表当前图上没有放板子

(ii) 目标状态：Shape 容器中元件对象个数为 7，代表 7 块板都成功找到了合适的位置，搜索结束

(iii) 中间状态：Shape 容器中元件对象个数介于 0～7 之间，其中每个对象都说明了“当前状态下，这一个元件，以这一种姿态，放在了原图中这个位置”信息

(iv) 状态转换：假如当前状态下 Shape 容器中有 n 个元素，$0 \le n < 7$，则：

(iv.i)试图放入第 n+1 个元素的操作为：扫描到当前状态下的一个顶点，在手头剩余的元件中搜索有无哪个元件能够以某种姿态放在该顶点上，若有，则将对应的 Shape 对象放入 Shape 容器中，并递归地开始下一层扫描和查找。

(iv.ii)若扫到的顶点无法放入手头任何元件，则说明之前的放置不正确，拆除 Shape 容器中最近放入的元素，即回到图中只有 n-1 个元素的状态，重新在该状态下探索其他的拼图方式

需要说明的是，当 Shape 放入容器中时，Picture 中的 bool 阵其实没有任何变

化，其中值为 True（黑色）的点并没有因为放入板子而变成 False（白色），而是 Picture 实现了 Picture::black(x,y)接口，它自动判断输入 x,y 坐标对应的点在原图中是否是黑点、以及是否被当前状态 Shape 容器中某个元件覆盖，只有原图中黑色且当前不被覆盖的点才返回 True。

2.2.3 算法鲁棒性处理

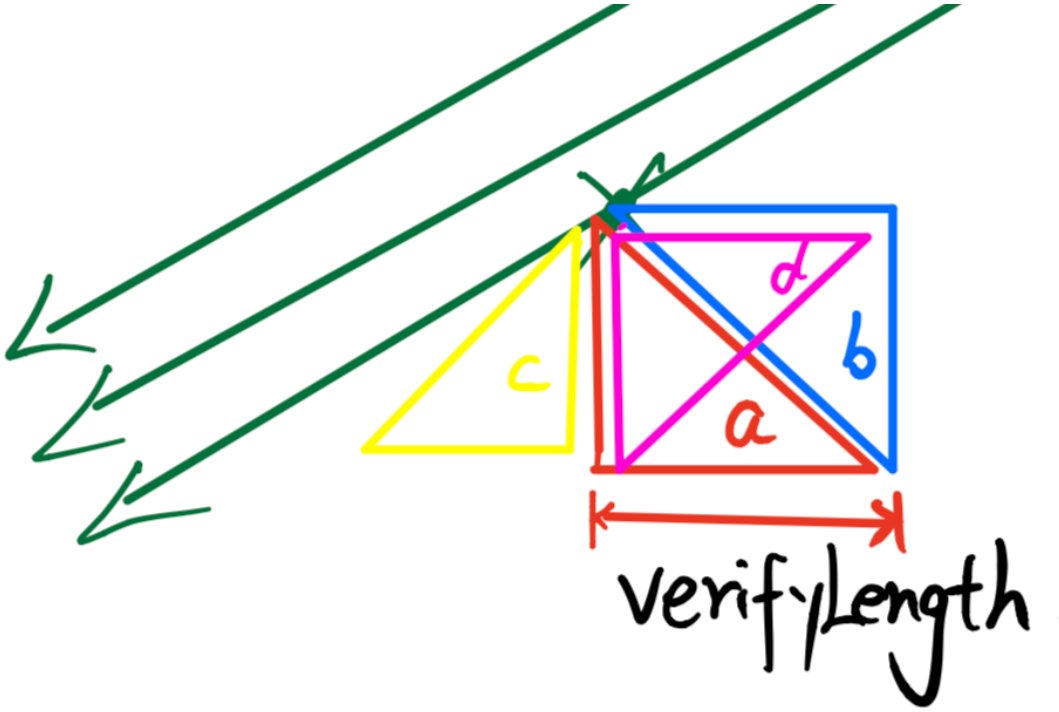
算法本身要对图像的噪点、图像中元件缺陷、元件比例不标准等非理想情况拥有鲁棒性，而要求输入的原图完美无缺必然有失合理性，因此这些非理想情况是十分常见的，如果不对这些实际情况进行考虑，算法永远是空中楼阁。为增加算法鲁棒性笔者做了以下处理：

①扫描过程中噪点检测：

在扫描图像 bool 阵过程中，如果不加识别地扫到点就开始拼，那么个别黑色噪点会带来致命的影响，因为离散的噪点无法放入任何元件，程序可能就会“不自信”地拆掉之前有可能正确的拼图方式，因此噪点检测十分重要，程序中用了

```
bool picture::insertAble(int x, int y)  （picture.cpp,line172）
```

函数来判断扫描到的黑点到底是离散的噪点还是可以拼接的顶点，检测的逻辑是：

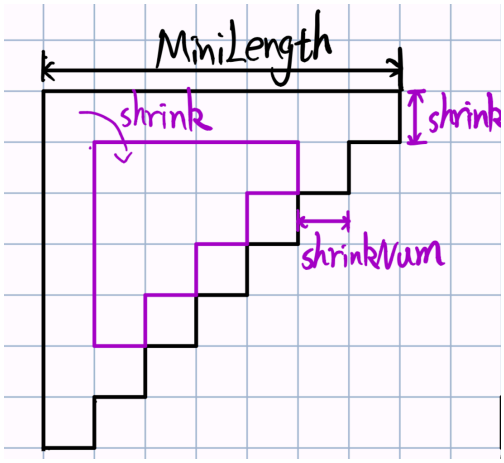


如图，在扫描到一个黑色点后，在直角边长为 verifyLength 的 4 个直角三角形区域 a , b , c , d 中检测有无足够多的黑点，只要其中 1 个区域被黑点填满，即认为该点是可以拼接的，这是因为任何元件以任何姿态摆放后，顶点被扫描到时，上述 4 个区域中至少有一个是被填满的。然而如果某个噪点也通过了该检测，说明该噪点在原图像上已经形成了可以明显察觉到的污渍区域，这种不合要求的图像的输入是可以提前避免的，如用画图软件手工处理掉这些污渍。对于一般的 1k

✖ 1 k 尺寸量级的图片来说，verifyLength 取 10 左右为宜。

② **元件放入时的 Shrink 处理和可放入判断**

由于原图中元件的尺寸有可能不合适，或元件的边界有缺陷或毛刺，因而需要判断元件能否放入时对这些情况有鲁棒性，笔者在程序中所采用的方法是让元件判定时尺寸缩小，以元件图形的几何中心为放缩点进行等比放缩：



如图，将判定元件能否放入的判定范围按原图放缩，可以减免原图边缘毛刺和缺陷带来的影响。

同时，在判定相应范围内是否有足够多的黑点时，采用

$$Ratio = \frac{Blacknode\ Numbers\ in\ the\ area}{Allnode\ Numbers\ in\ the\ area} \times 100\%$$

来判定，当 Ratio>98%时即认为可以放入相应元件，这增加了对图形内部离散白色缺陷的鲁棒性。

对于一般的 1k✖ 1 k 尺寸量级的图片来说，shrinkNum 取 10 左右为宜。

③ **尺寸自适应时使图形略微膨胀**

在搜索过程中，程序通过黑色待求解区域的总面积计算出了 7 巧板的对应尺寸，但由于浮点数转为整数时向下取整，会使得计算出的图形尺寸比实际尺寸略小，导致图形在放入后，有时会留下一条很长的黑色带状区域，当有更多的板子放入时，这些带状区域造成的误差将不可忽视，可能会导致算法崩溃，因此在计算 MiniLength 和 OddLength 时给它们一个微小的增大偏置，使图形微微膨胀，这时就可以减免这种带状区域造成的误差，增强算法的鲁棒性。

④ **采用②、③方法的理由**

笔者设计程序时采用了上述**②、③方法**来提高程序鲁棒性，即在判断能否放入元件时，使图形有效范围缩小，在放入后使图形略微膨胀，这实际上是在对图形的边缘进行等效软化处理，由于程序中每个像素点非白即黑（bool 值存储），因此其边缘是非常“硬”的，即边界处梯度很大，这样的 bool 二维存储虽然节约了内存，但会导致图像本身对噪点的鲁棒性不够，因此采用上述②、③方法来等效地软化其边缘，从算法层面来弥补存储结构的缺点。

2.2.4 算法正确性与完备性证明

对该深度优先搜索算法的正确性和完备性证明如下：

证明：

若当前待求解图像的任何 1 个解为：

$$S_7 = \begin{pmatrix} (x_0, y_0, T_0, P_0) \\ (x_1, y_1, T_1, P_1) \\ (x_2, y_2, T_2, P_2) \\ (x_3, y_3, T_3, P_3) \\ (x_4, y_4, T_4, P_4) \\ (x_5, y_5, T_5, P_5) \\ (x_6, y_6, T_6, P_6) \\ \\ \end{pmatrix}$$

上面的矩阵即代表 Shape 容器的最终状态，每个横分量(x_i, y_i, T_i, P_i)代表：**形状为 T_i 的元件，以姿态 P_i 放在原图形的 (x_i, y_i) 位置处**，由于对最终拼接成功的结果来说，横分量的前后顺序是没有影响的，不妨假设从上至下它们按照扫描时扫到的先后顺序出现。

假设在搜索算法运行中，没有找到该解，则易知状态：

$$S_6 = \begin{pmatrix} (x_0, y_0, T_0, P_0) \\ (x_1, y_1, T_1, P_1) \\ (x_2, y_2, T_2, P_2) \\ (x_3, y_3, T_3, P_3) \\ (x_4, y_4, T_4, P_4) \\ (x_5, y_5, T_5, P_5) \\ \\ \\ \end{pmatrix}$$

亦没有被找到，因为如果找到了上面的状态，算法必然会扫描到顶点(x_6, y_6)并推算出S_7. 同理可知：

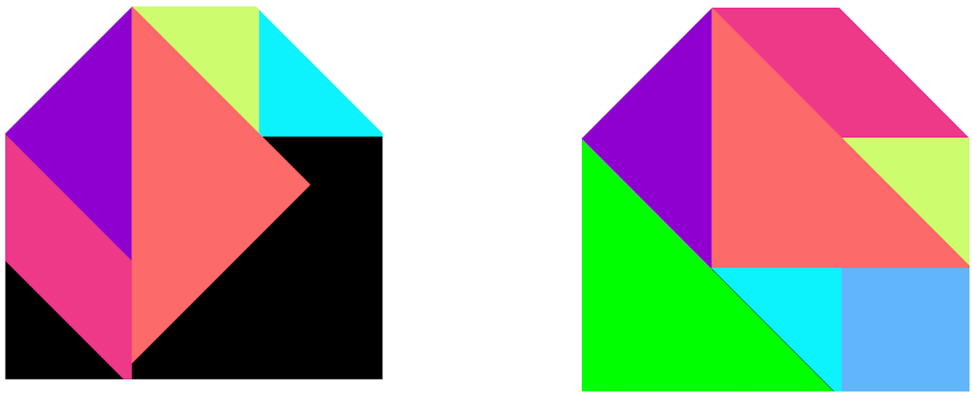
$$S_5 = \begin{pmatrix} (x_0, y_0, T_0, P_0) \\ (x_1, y_1, T_1, P_1) \\ (x_2, y_2, T_2, P_2) \\ (x_3, y_3, T_3, P_3) \\ (x_4, y_4, T_4, P_4) \\ \\ \\ \\ \end{pmatrix} \quad S_4 = \begin{pmatrix} (x_0, y_0, T_0, P_0) \\ (x_1, y_1, T_1, P_1) \\ (x_2, y_2, T_2, P_2) \\ (x_3, y_3, T_3, P_3) \\ \\ \\ \\ \\ \end{pmatrix} \ldots\ldots S_1 = \begin{pmatrix} (x_0, y_0, T_0, P_0) \\ \\ \\ \\ \\ \\ \\ \\ \end{pmatrix}$$

都没有被找到，而由于算法第一次扫描时一定会扫到(x_0, y_0)，并且形状为 T_0 的元件，以姿态 P_0 可以放置在原图形的 (x_0, y_0) 位置处，因此S_1状态一定会被找到，相互矛盾，因此假设不成立。**证毕。**

因此理论上该深度优先算法可以找到待求解图形的所有解，但实际的搜索树可能会呈现宽度爆炸，寻找所有解有时会非常耗时，因此在本程序中设置了找到 1 个解即停止。

2.2.5 算法的优势与不足

优势：
①采用二维 bool 阵存储图像，充分节约了内存空间并发挥了计算机对这种数据结构随机访问快的优势
②理论上可以找到所有可行解，且能够较快地找到第一个可行解
③算法对图像的突破口在于扫描到的顶点，在一般情况下，对于棱角比较分明、元件比较分散的待解图像有很快的求解速度
④可扩展性强，不依赖于七巧板自身的几何结构进行搜索，可以自然扩展到更高阶巧板，只需对每个元件的轮廓进行数学建模，并对扫描线斜率进行微调即可
不足：
①算法的 1 个很大不足在于，在深度优先策略下，算法求解速度会显著受元件放置优先顺序的影响，如某些图放置三角形时优先直角边平放只需几步即可得解，但在优先斜边平放时需要搜索 1000 步才能得解。
②如果待解图形是由元件紧密拼接而成的大块简单图形，棱角较少（如大正方形、大平行四边形、大梯形等），可能会导致算法搜索空间大幅上升，如果元件放置优先顺序不合适（如①所述）可能会导致求解速度较慢。



如上图所示，粉色大三角形开始放置时选择斜边平放优先，而实际情况应该是直角边平放，这会导致在没有意义的分支下进行一些不必要的探索。

2.2.6 算法改进空间与选做拓展思路

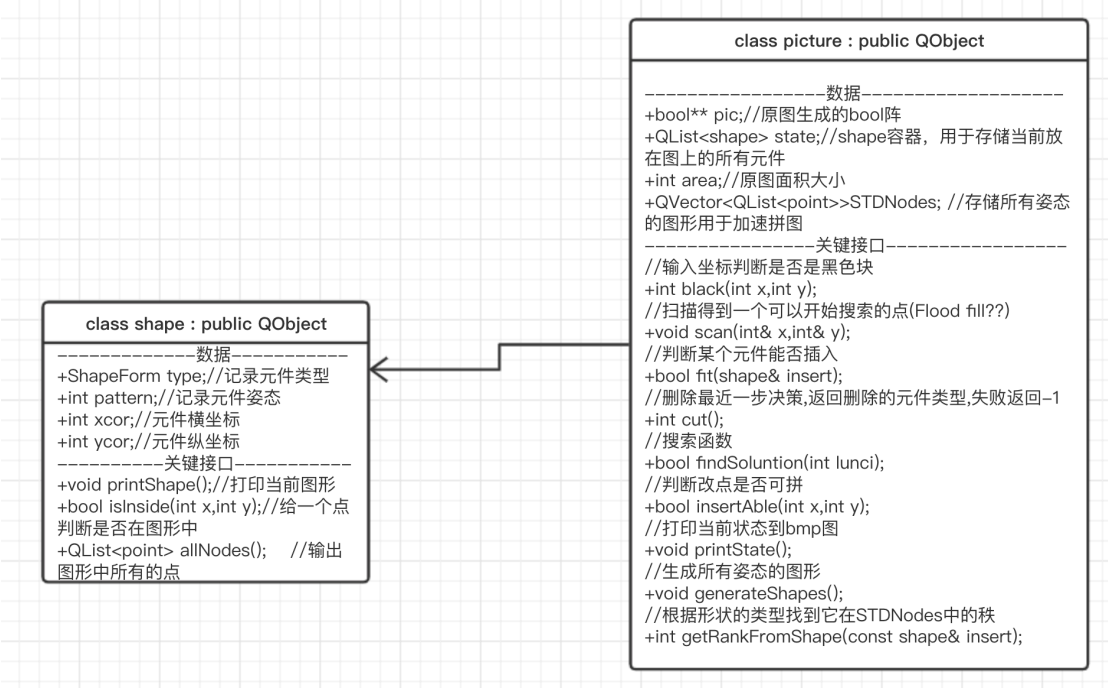
①加入剪枝：上面所说由于元件放置优先顺序不合适导致的搜索空间增大实际上可以通过合理的剪枝来避免，比如上面左图中右侧出现了无法拼接的形状，就及时剪枝停止无效探索，但笔者在开发过程中暂时没有想到快速识别无效节点并剪枝的策略。
②加入启发函数：该算法采用纯深度优先策略，在某个节点探索到某个元件可以

放入后便立刻放入，进入下一个状态，如果能够拥有一个理想的评估在当前节点上能放进去的所有图形好坏的启发函数，则算法会更智能，减小其盲目性，对上面所说的顺序敏感问题也能有所改善。
③对选做任务的想法：选做任务 1 需要求解更高阶的巧板，由于这一算法自身的可扩展性较强，在求解更高阶时只需对每个元件进行数学建模即可，这一工作由于时间关系笔者并没有能完成，但笔者相信这一算法能够自然地拓展到更高阶情形。

3 类结构设计

3.1 后端类结构设计

①算法逻辑的关键类：



如上图，算法逻辑的两个关键类及其接口的 UML 如上，其中 ShapeForm 枚举类型定义为：

```
enumShapeForm{BigTri1,BigTri2,Square,Parallel,MidTri,SmallTri1,SmallTri2};
```

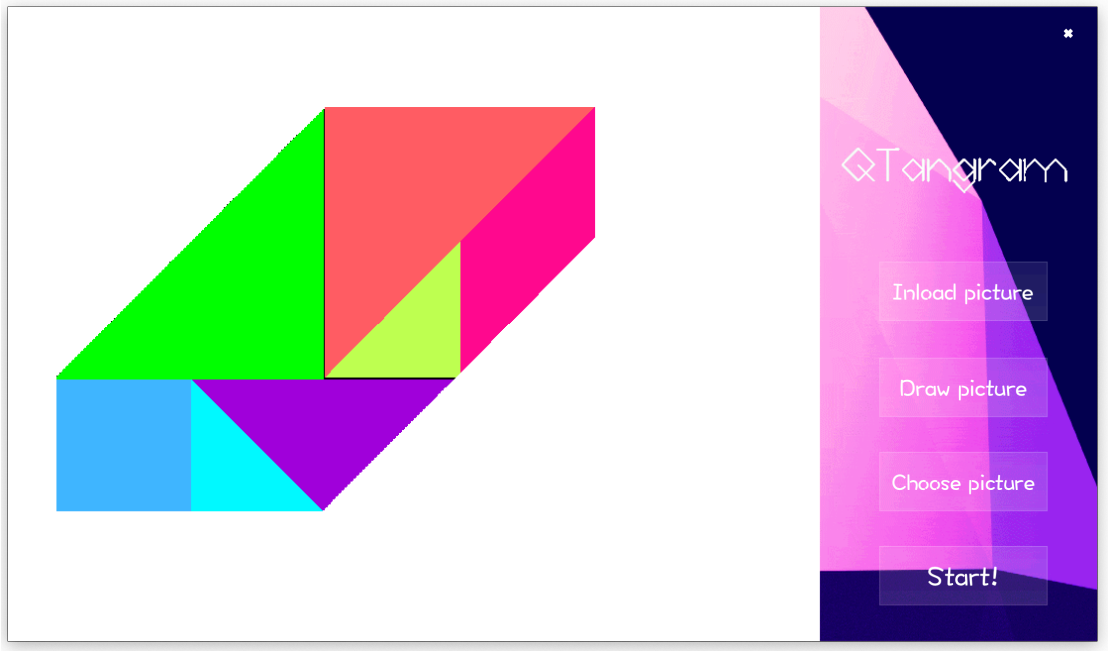
定义了 7 种不同的元件：两个大三角形、正方形、平行四边形、中三角形、两个小三角形

②用户自定义图形类 Mpolygon:

该类继承自 QGraphicsPolygonItem，用于生成用户创作界面的元件，其中定义了鼠标点击、拖动、释放等事件、重写了基类中标明图形范围的两个函数，实现了模拟现实中七巧板的移动、旋转、拼接功能，并加入了碰撞检测放置巧板之间相互堆叠遮挡，但由于时间关系没能做出理想的图形越界检测，较为遗憾。

3.2 界面类结构设计

①HomepageUI 类：



为前端最重要的类，作为用户主页，实现了与用户交互的点击、按钮等功能，并通过信号槽等机制与后端对象通信实现前后端交互。

在 UI 美工方面，笔者秉承极简主义的美学原则，只留下最核心的功能模块、不添加干扰用户操作的不必要插图，界面跳转结构也是简单的两状态机，使得用户“一看到，就会用”。

色调搭配时，笔者仔细挑选了几种视觉观感协调优美的颜色，优化用户视觉体验；在右侧加入 poly 风格的渐变动态图增强动态感；右侧 QTangram Logo 和动图都是笔者经 PS 绘制而成。

另外，笔者删除了系统自带的窗口上边栏，重新定义了界面中的鼠标事件，使得用户按住界面上任何一处都可以拖动界面，动态感较强。

② ChoosePictureUI 类：

是用户从图像库中选择图像导入左侧展示区的对话框。

另外 widget 类没有用，它是系统生成工程时自带的。

4 项目总结

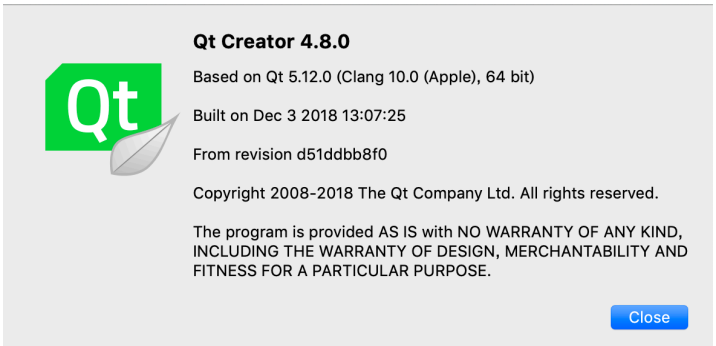
①经过几个星期的奋战、无数次细节遇坑、多次鲁棒性调整后，笔者终于实现了能够自行求解七巧板问题的人工智能，收获了巨大的快乐和成就感

②开发过程中，程序的核心与灵魂所在是整套搜索算法的实现，笔者构思算法大概用了 1 周多的时间，期间想过无数种搜索算法，在不断地权衡、更迭后才确定了这一算法。由于七巧板的搜索算法并没有火柴棍任务那样一目了然，因此构思搜索算法的过程的确是非常考验思维的灵活性和创造性的，尤其是七巧板求解搜

索需要和数字图像处理知识技能相配合，也使得这一问题处理起来比较棘手。
③在写完算法的逻辑框架后，笔者发现自己的算法是无法使用的，这是因为算法本身应该对图像的噪点、图像中元件缺陷、元件比例不标准等非理想情况拥有鲁棒性，而要求输入的原图完美无缺必然有失合理性，因此这些非理想情况是十分常见的，如果不对这些实际情况进行考虑，算法永远是空中楼阁，而笔者的实践过程中，最考验工程直觉、最磨人的便是这一工作。笔者首先发现图像中的离散噪点对算法是致命的，因此在查阅资料后决定使用洪水填充（Flood Fill）算法解决离散噪点，但后来又出现的图片中元件自身缺陷或尺寸问题也使得算法无法执行，遂又想了新的元件顶点识别算法，期间经过了许些版本更迭和参数调整，才进化出 2.2.3①的降噪算法，达到了较为理想的效果。
④笔者在构思工程初期，想到节约内存的用 bool 阵存储图片方法，但这一方法存储的像素点非黑即白，使得图形的边缘十分 Hard，这导致算法在初期时常常崩溃，比如原图稍有缺陷即判断无法放入元件，这一问题困扰了笔者很久，甚至想过用 float 阵代替 bool 阵来软化边缘，但这会带来数倍的内存消耗，后来在与工程经验丰富的同学交流后想到了 2.2.3②③的算法，从算法优化的层面弥补了数据结构的缺点
⑤为了丰富用户体验和人机交互，笔者在做出可以输入图片文件的程序后，又增加了用户创作模式，笔者为增加这一模式临时学习了 QT 的 QGraphicsView 显示与交互体系，发现可以用它来实现很多人机交互的创新模式，但出于时间原因最终没能解决越界检测问题，比较遗憾。

5 相关问题的说明

开发者计算机系统：macOS Mojave 10.14
开发者使用的集成开发环境：QT Creator for Mac 版本 4.8.0



编译环境：Desktop QT 5.12.0 clang 64bit
UI 设计辅助：PhotoShop CC2018 、 GIPHY