

浙江大学

计算机摄影学课程设计



设计名称 Grab Cut

姓名与学号 叶培峰 3140104023

专 业 计算机科学与技术

学 院 计算机学院

摘要

本项目源于微软研究院的一个课题 Grab Cut。在这次课程设计中项目实现了基于论文《“GrabCut” — Interactive Foreground Extraction using Iterated Graph Cuts》的用于图像分割的工程。该工程分为两部分，首先是基于 GMM 的图像硬分割的实现，其次是优化分割结果的 Border Matting 的实现。

关键词 GMM Border-Matting

目录

摘要	I
第 1 章 Grab Cut 理论概述	1
1.1 从 Graph Cuts 到 Grab Cut	1
1.1.1 Graph Cut	1
1.1.2 Grab Cut.....	2
1.2 Border Matting.....	3
第 2 章 Grab Cut 具体实现	5
2.1 GMM 实现	5
2.2 Graph 实现及 MinCut.....	6
2.3 Border Matting 实现.....	8
第 3 章 效果展示与比较	12
3.1 Grab Cut 及 Border Matting 效果展示.....	12
3.2 性能评测	21
3.3 Grab Cut 硬分割阶段与 openCV 实现比较	23
参考文献	25

第1章 Grab Cut 理论概述

1.1 从 Graph Cuts 到 Grab Cut

Grab Cut 是从 Graph Cut^[1]发展而来，不同于 Graph Cuts 在计算时采用灰度图计算参数进行能量优化从而得到分割结果，Grab Cut 适用采用多通道图像作为参数训练模型，并参照 Graph Cuts 的思路和方法进行图像分割处理。

1.1.1 Graph Cut

Graph Cut 基于的思路是将图像分割问题与图的最小割问题相关联。在图像处理中，这样的图的构造不难联想，对于每一个像素与它邻接的 8×8 像素之间构造边。同时我们需要构造出两个虚拟的节点来表示源节点 S 与汇聚点 T，我们以合适的权值对这些边加以赋值，然后应用网络流算法中的 MaxFlow，这与 MinCut 等价，这样就能判定出节点是属于源点还是汇聚点，如下图。

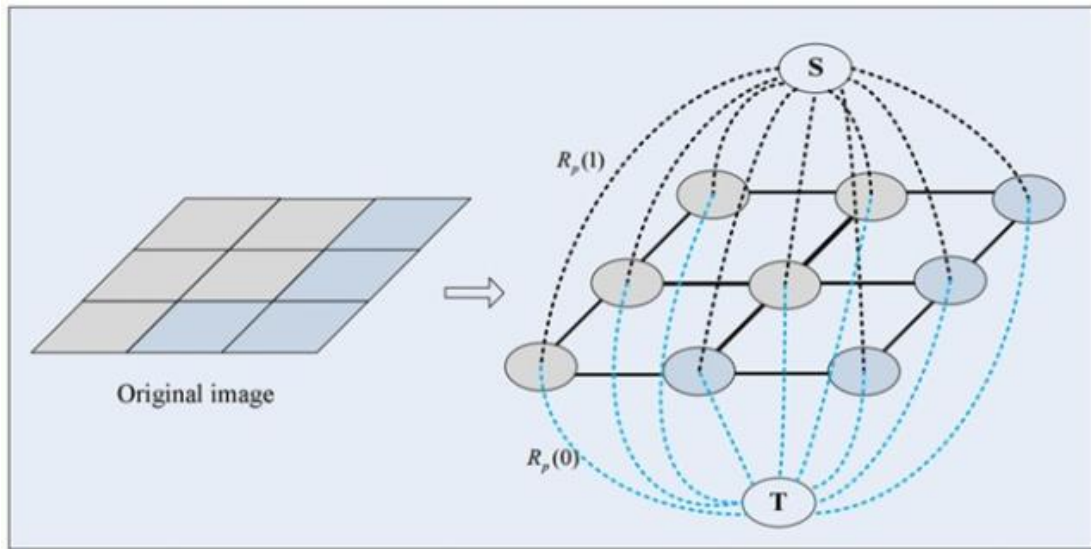


图 1

那么问题的关键就在于如何确定每条边的权值，这里 Graph Cut 应用的是用能量最小化方程的形式来抽象出这样一种网络流图边权值的赋值策略：

$$E(\alpha, \theta, z) = U(\alpha, \theta, z) + V(\alpha, z)$$

数据项 U 表示了当前像素点在当前 α 下对于灰度直方图 h 的匹配程度，实际意义即该像素属于前景还是后景的概率，该概率越低则对应的能量越高，这个 U 对应于网络流图中与源点和汇聚点的边的权值，这个权值以下面的形式度量：

$$U(\underline{\alpha}, \underline{\theta}, \mathbf{z}) = \sum_n -\log h(z_n; \alpha_n).$$

平滑项 V 用于度量当前 α 对于当前的像素点 z 在周围环境中的合适程度，用于构造网络流图中的点与他的邻居之间的边的权值，用下面的公式度量：

$$V(\underline{\alpha}, \mathbf{z}) = \gamma \sum_{(m,n) \in C} dis(m,n)^{-1} [\alpha_n \neq \alpha_m] \exp -\beta (z_m - z_n)^2,$$

$[\phi]$ 代表指示函数，对于预测 ϕ 取 0 或 1。而这里的系数 β 则代表 m, n 两点距离平方的期望的倒数：

$$\beta = \left(2 \left\langle (z_m - z_n)^2 \right\rangle \right)^{-1};$$

有了这些公式我们就依据这些公式来构造图，最后我们采用的 MinCut 算法用来最小化能量函数 E 。得到图的分割。

1.1.2 Grab Cut

Grab Cut 采用的策略与 Graph Cut 类似，区别在于这里采用的能量函数的数据项所采用的计算方式不同，也就是网络流图中与源点聚点的权值计算方式不同。Grab Cut 采用 GMM 聚类方法来聚类前景点和背景点，并根据聚类训练出的 GMM 参数来判定一个点属于前景或背景的概率，从而得到数据项。

Grab Cut 的能量函数形式与 Graph Cut 类似，定义如下：

$$E(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) = U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) + V(\underline{\alpha}, \mathbf{z}),$$

相比于 Graph Cut 多出的参数 k 用于指代该像素适用于哪个 GMM 的高斯分量。

数据项 U 如下刻画：

$$D(\alpha_n, k_n, \theta, z_n) = -\log[p(z_n|\alpha_n, k_n, \theta) \cdot \pi(\alpha_n, k_n)]$$

相比于论文中的公式，我更喜欢这样的公式表达形式因为更加直观，数据项就是在刻画该像素点在 k_n 这个高斯分量中的概率，在乘上该高斯分量在 GMM

模型中的系数，然后再取负对数，这样就得到了数据项的能量，概率越小则这个能量越大。

平滑项基本类似不再赘述：

$$V(\underline{\alpha}, \mathbf{z}) = \gamma \sum_{(m,n) \in C} [\alpha_n \neq \alpha_m] \exp -\beta \|z_m - z_n\|^2.$$

构造出这样的图后同样应用 maxFlow 的算法就可以得到更新后的 α 。依次迭代该步骤，则可以得到更好的结果。

1.2 Border Matting

Border Matting 的核心想法在于如何把 Grab Cut 中迭代得到的硬分割边缘变成软分割，即 α 不再简单取 0,1 而是可以取[0-1]，这样就能得到比较光顺的边缘效果。

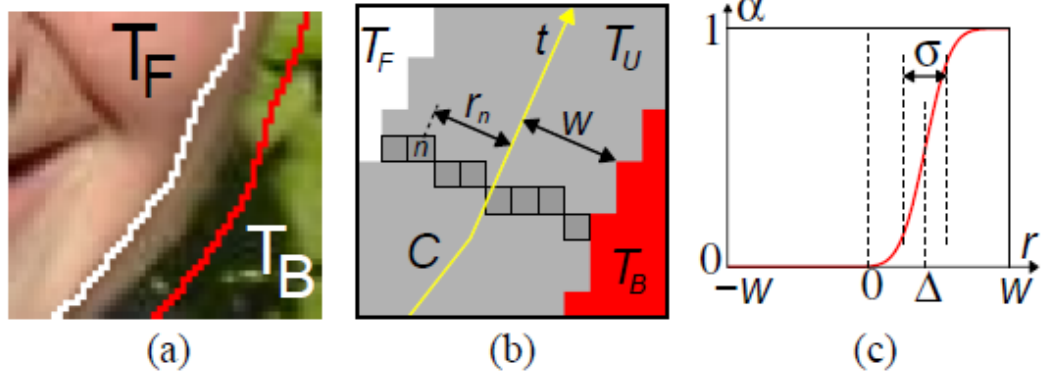


图 2

如图 2 中所示概念，首先构造出图像边缘及其邻居的集合 T_U ，采用平滑函数 g 来计算 α 的取值，核心想法是如何计算出中心点及其附近邻居的 α 使得结果最优，那么调整 g 的参数可以做到这一点，论文中提出的方法也是采用能量最小化的思想，这里的能量函数如下定义：

$$E = \sum_{n \in T_U} \tilde{D}_n(\alpha_n) + \sum_{t=1}^T \tilde{V}(\Delta_t, \sigma_t, \Delta_{t+1}, \sigma_{t+1})$$

这里的数据项 D 可以直观的理解为当前像素在周围像素的融洽程度，而这里的平滑项 V 则是指相邻的中心点前后之间的变化幅度，因此这个函数最小化的目的就是使得轮廓上的像素尽可能的有着比较好的 α 使得它在周围像素看起来不那么突兀比较融洽，同时轮廓的相邻点之间应该尽可能的平滑，变化幅度不应太大以至于产生撕裂状的锯齿。

所以 D 定义如下：

$$\tilde{D}_n(\alpha_n) = -\log \mathbf{N}(z_n; \mu_{t(n)}(\alpha_n), \Sigma_{t(n)}(\alpha_n))$$

即融洽程度被刻画为当前 α 在周围像素近似以高斯分布的情况下的概率，这个高斯分布的参数以如下的方式由前景和背景参数刻画：

$$\begin{aligned}\mu_t(\alpha) &= (1 - \alpha)\mu_t(0) + \alpha\mu_t(1) \\ \Sigma_t(\alpha) &= (1 - \alpha)^2\Sigma_t(0) + \alpha^2\Sigma_t(1).\end{aligned}$$

而平滑项 V 则是前后像素的拟合函数 g 不能差别太大：

$$\tilde{V}(\Delta, \sigma, \Delta', \sigma') = \lambda_1(\Delta - \Delta')^2 + \lambda_2(\sigma - \sigma')^2,$$

论文中提出使得该能量函数最小化用的是一种近似的 DP 算法，并指出如果不用 distance transform 加以优化则从一个像素移动到下一个需要 $O(N^4)$ 的复杂度，很显然说这是比较难以接受的。但是如果加上 distance transform 则可以以线性时间完成这个工作。

第2章 Grab Cut 具体实现

2.1 GMM 实现

根据论文中的算法流程，Grab Cut 开始迭代时第一件工作就是初始化 GMM 组件，并根据当前的 Mask 训练 GMM 模型。

我们运行 GrabCut()接口时，会附带参数指示当前的 Mask，如果是矩形的话则会把矩形外面的初始化为背景点，矩形内部的点初始化为可能的前景点。然后根据这个 Mask 导出两组 GMM 所需的数据集，即前景 GMM 的点集和背景 GMM 的点集。然后接下来的工作就是初始化以及训练 GMM。

```
GMM bgdGMM, fgdGMM;
vector<double> bgd, fgd;
for (in mask){
    if(mask) fgd.push(p);
    else    bgd.push(p);
}
fgdGMM.Train(fgd, fgd.size() / 3);
bgdGMM.Train(bgd, bgd.size() / 3);
```

本工程中 GMM 的初始化采用的是 kmeans 的方法，根据论文中设置聚类数为 5，即有 5 个高斯分量。然后初始化各个分量的参数 mean 和 var 为后面，值得注意的是，因为图像有三个通道，所以需要分别初始化每一个通道在每一个分量中的参数。

然后我们需要通过 EM 方法训练每个分量中每个通道的参数，算法流程如下：

1. 估计数据由每个 高斯分量生成的概率（并不是每个 Component 被选中的概率）：对于每个数据 x_i 说，它由第 k 个 高斯分量生成的概率为：

$$\gamma(i, k) = \frac{\pi_k \mathcal{N}(x_i | \mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(x_i | \mu_j, \Sigma_j)}$$

2. 通过极大似然法估计每个参数，就上式中的均值 μ 和方差 Σ 。

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^N \gamma(i, k) x_i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^N \gamma(i, k) (x_i - \mu_k)(x_i - \mu_k)^T$$

3. 重复迭代前两步直到方差和均值收敛即变化小于阈值或达到最大迭代次数不收敛。

如果迭代收敛我们就可以得到一个高斯分量的参数以及每一个高斯分量在 GMM 模型中所占的权重，通过这个训练好的 GMM 模型为后续构造图做准备。

2.2 Graph 实现及 MinCut

我们首先需要算出每个点他的邻居对于他的权值，也就是平滑项 V ，这里的具体实现即应用论文中的公式，举例计算左侧邻居的节点：

```
Vec3b Left = img.at<Vec3b>(y, x - 1);
left.at<double>(y, x) = gamma * exp(-beta*(Point-Left).dot(Point - Left));
```

值得注意的是，每个点周围虽然有 8 个邻居，但是需要计算并保存的只有 4 个，因为如果你只计算并保存你的上半部分邻居的值，你的下半部分会有你的邻居们完成计算。这些计算的 V 保存在与原图等尺寸的矩阵 `left`, `up`, `leftup`, `rightup` 中即 \leftarrow , \nwarrow , \uparrow , \nearrow 方向的邻居。

得到邻居后，我们就可以着手再次遍历原图得到每个点的数据项并把它们都加到图中。

本次项目中使用 `opencv` 自带的 `Graph` 库来构建这个网络流图。

```
if (mask.at<uchar>(y, x) == GC_PR_BGD || mask.at<uchar>(y, x) ==
GC_PR_FGD)
{
    double tmp[3] = { Point[0], Point[1], Point[2] };
    S = -log(bgd.GetProbability(tmp));
    T = -log(fgd.GetProbability(tmp));
}
else if (mask.at<uchar>(y, x) == GC_BGD)
{
    S = 0;
    T = lamda;
}
else
{
    S = lamda;
    T = 0;
}
```

```
graph.addTermWeights(NodeID, S, T);
```

然后我们把他们的邻居加入图中：

```
if (x-1>=0)
    graph.addEdges(NodeID, NodeID - 1, WeigtL, WeigtL);
if (x-1>=0 && y - 1 >=0)
    graph.addEdges(NodeID, NodeID - img.cols - 1, WeigtUL, WeigtUL);
if (y-1>=0)
    graph.addEdges(NodeID, NodeID - img.cols, WeigtU, WeigtU);
if (x + 1 < img.cols && y-1>=0)
    graph.addEdges(NodeID, NodeID - img.cols + 1, WeigtUR, WeigtUR);
```

最后运行网络流 maxflow 算法得到经过划分的图并修改 Mask 完成一次迭代：

```
ConstructGraph(src, mask, left, upleft, up, upright, bgdGMM, fgdGMM, graph);
graph.maxFlow();
for (int y = 0; y<mask.rows; y++)
    for (int x = 0; x < mask.cols; x++)
    {
        if (mask.at<uchar>(y, x) != GC_FGD)
        {
            if (graph.inSourceSegment(y*mask.cols + x))
                mask.at<uchar>(y, x) = GC_PR_FGD;
            else
                mask.at<uchar>(y, x) = GC_BGD;
        }
    }
}
```

2.3 Border Matting 实现

Border Matting 所需要做的工作主要也分为两步，首先是图像轮廓点的采集构建并有序化，然后是能量函数的最小化。

由于图像本身的轮廓可能千差万别，所以图像轮廓的有序构建还是需要一定的方法来实现的。

首先本项目先采用 OpenCV 自带的 Canny 算法来提取目标图像的轮廓，注意到这里不能直接放入原图像提取，原因很简单，Canny 并不知道你要提取的是哪一个物体的轮廓，他会把这张图片上所有的物体轮廓信息都提取出来，这是我们不希望得到的。所以我们这里对 Mask 图像 and 1 之后进行提取，这样提出来的的是经过 Grab Cut 硬分割后的图像的轮廓。

提取到轮廓后我们需要把这个轮廓矩阵有序化为一个连续像素点组成的向量 Contour，这里用到的技巧是遍历原图像，如果当前像素点是在提取的边缘矩阵中标记为边缘的，那么我就在当前的 Contour 向量中找一个与当前像素 RGB 空间上欧式距离小于 2 的点，然后把它插在后面，如果找不到就插到向量的末尾，不难想象因为图像遍历是严格按照扫描顺序的，如果轮廓本身是连续成环的，那么得到的 Contour 也必然满足这个性质。

```
void mybm::ConstructContour()
{
    for(int i=0; i<Src.rows; i++)
        for (int j = 0; j < Src.cols; j++)
        {
            if (Edge.at<uchar>(i, j))
            {
                Push(point(j, i), contour, 2);
            }
        }
}

void mybm::Push(point p, vector<Contour> &list, int threshold)
{
    int min = -1;
    for (int i = 0; i < list.size(); i++)
    {
        if (p.distance(list[i].p) <= threshold)
        {
            list.insert(list.begin() + i, Contour(p));
            return;
        }
    }
}
```

```

    }
}
list.push_back(Contour(p));
}

```

有了 Contour 的构造方法，那么把它的邻居加进去也可以用类似的方法，这样，这样 Contour 这个子类的元素就齐全了。这些数据结构定义如下：

```

struct LocalPara {
    Vec3b Bmean;
    Vec3b Fmean;
    double Bvar;
    double Fvar;
};

class point{
public:
    Point2d p;
    int delta, sigma;
    int dis;
    double alpha;
    LocalPara para;
    point(int x, int y, int dis=0)
    {
        this->p = Point2d(x, y);
        this->dis = dis;
    }
    point() {};
    int distance(point t)
    {
        return sqrt((this->p.x - t.p.x)*(this->p.x - t.p.x) + (this->p.y -
t.p.y)*(this->p.y - t.p.y));
    }
};

class Contour {
public:
    point p;
    vector<point> neighbor;
    Contour(point p) { this->p = p; };
};

```

收集到了需要处理的有序信息点之后，就可以着手开始能量最小化工作了。不同于论文中的 DP 实现，本项目中采用近似算法，采用贪心的策略以 $O(N)$ 的复杂度来完成减小能量的工作， N 为轮廓点的个数。这样做的好处是显而易见的，如果追求能量最小全局最优，则付出的时间代价是显然的，如果按照论文中用 distance transform 的方法优化之后， t 到 $t+1$ 的复杂度是 $O(N)$ 那么，全局的复杂度将达到 $O(N^2)$ ，如果不优化则更是达到了可怕的 $O(N^3)$ ，显然这是难以接受的。所以本项目采用贪心算法来提升效率，并且能保证至少获得 60% 的最优解的效果。

贪心的策略也很简单，取当前像素点和他的邻居选取的参数 σ 和 Δ 使得当前的 $D+V$ 即能量和最小。即算法实现为：

```
for (int i = 1; i < contour.size(); i++)
{
    LocalPara para;
    getLocalMandV(contour[i].p, para);
    contour[i].p.para = para;
    for (int j = 0; j < contour[i].neighbor.size(); j++)
    {
        point &p = contour[i].neighbor[j];
        getLocalMandV(p, para);
        p.para = para;
    }
    double min = 999999999;
    int ns, nd;
    for (int si = 0; si < 30; si++)
        for (int di = 0; di < 10; di++)
        {
            double D = dataTermPoint(contour[i].p,
toGray(Src.at<Vec3b>(contour[i].p.p.y, contour[i].p.p.x)), si, di, contour[i].p.para);
            for (int j = 0; j < contour[i].neighbor.size(); j++)
            {
                point &p = contour[i].neighbor[j];
                D += dataTermPoint(p, toGray(Src.at<Vec3b>(p.p.y, p.p.x)), si, di,
p.para);
            }
            double V = 2 * (si - delta)*(si - delta) + 360 * (sigma - di)*(sigma - di);
```

```
        if (D + V < min)
        {
            min = D + V;
            contour[i].p.delta = si;
            contour[i].p.sigma = di;
        }
    }
    sigma = contour[i].p.sigma;
    delta = contour[i].p.delta;
    contour[i].p.alpha = g[0][delta][sigma];
    for (int j = 0; j < contour[i].neighbor.size(); j++)
    {
        point &p = contour[i].neighbor[j];
        p.alpha = g[p.dis][delta][sigma];
    }
}
```

值得注意的是这里取得计算 α 的函数 g 是事先算好的，这样可以进一步加快效率，用查表的方式代替计算。我用的 g 是线性的函数，即在 σ 的范围内 α 线性增加， g 这张表的计算在 GrabCut 初始化的时候完成。

第3章 效果展示与比较

3.1 Grab Cut 及 Border Matting 效果展示

我们展示四组图片，展示的顺序分别为原图，Grab Cut 处理后的硬分割图，最后是应用过 Border Matting 后的图片：

哈士奇：

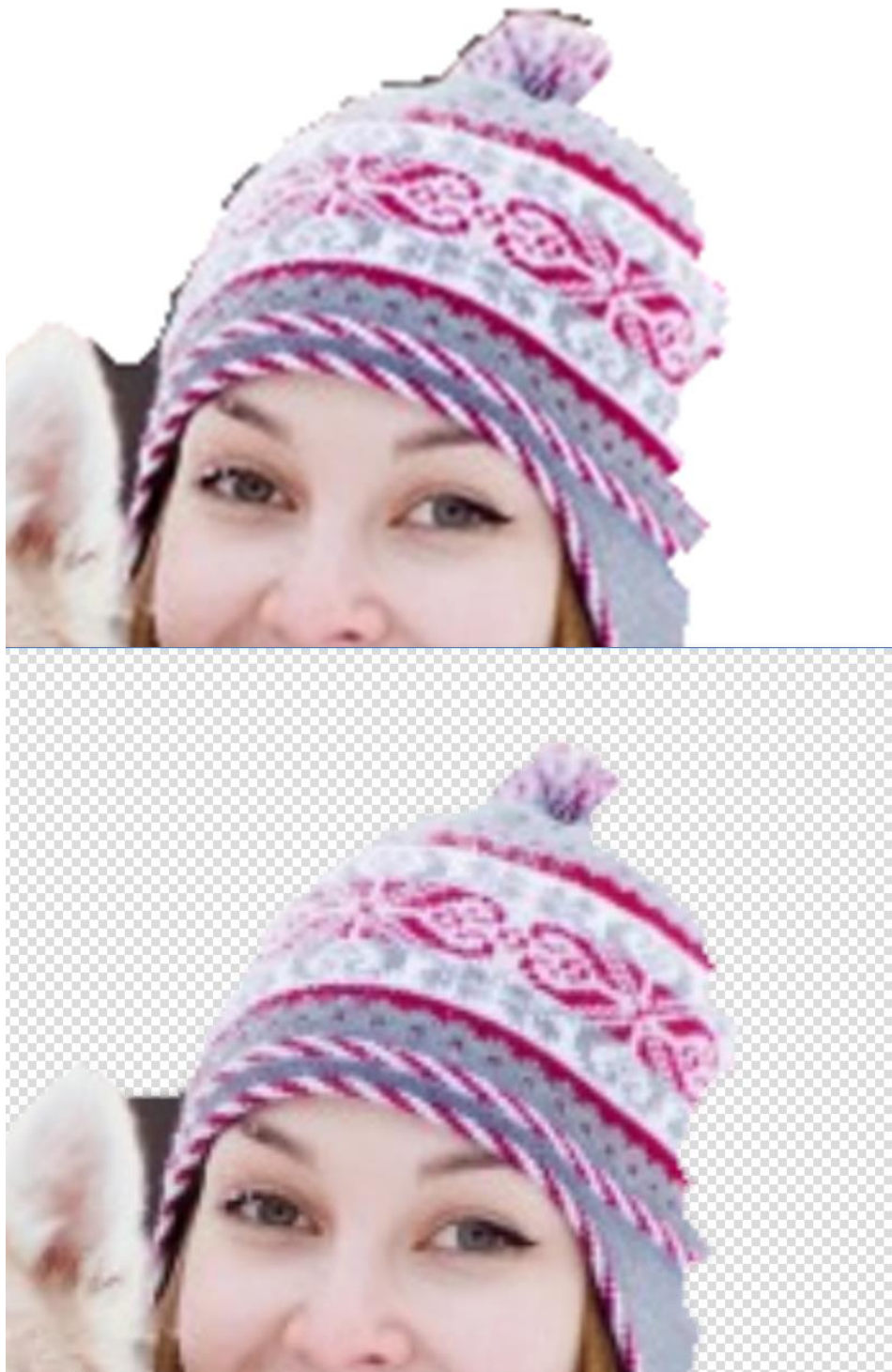




可以看到经过 Border Matting 硬分割带来的边缘色彩 Bleeding 的效果减弱了。



放大后看，由于 BorderMatting 经过了透明化处理所以背景是透明的，用软件打开会有网格：



论文中的图片骆驼:





论文中的人头：





可以看到绿色的颜色 Bleeding 被去掉了一些，部分去不掉的区域因为 BorderMatting 的 Strip 的宽度不够。



圣诞节的女孩：





放大后看毛发细节的处理：





可以看到对于明显锯齿残缺也有一定的修补作用。

3.2 性能评测

本机测试环境：

[查看有关计算机的基本信息](#)

Windows 版本

Windows 10 专业版
© 2017 Microsoft Corporation。保留所有权利。

Windows 10

系统

处理器:

Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz 4.00 GHz

已安装的内存(RAM):

16.0 GB

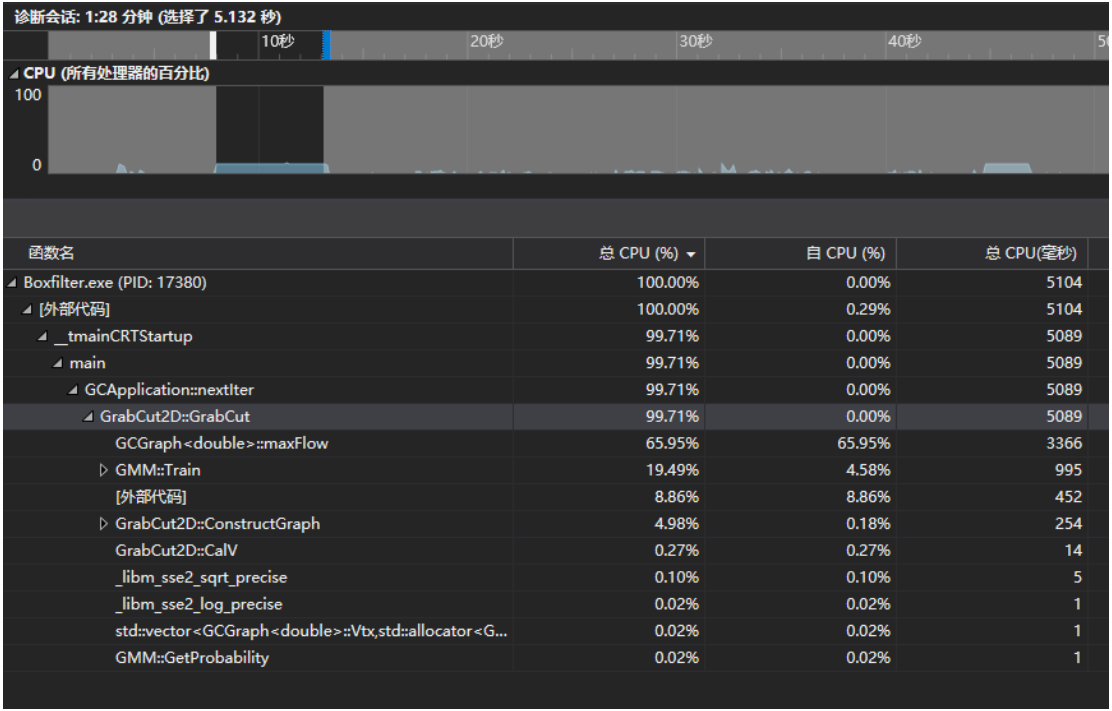
系统类型:

64 位操作系统, 基于 x64 的处理器

笔和触控:

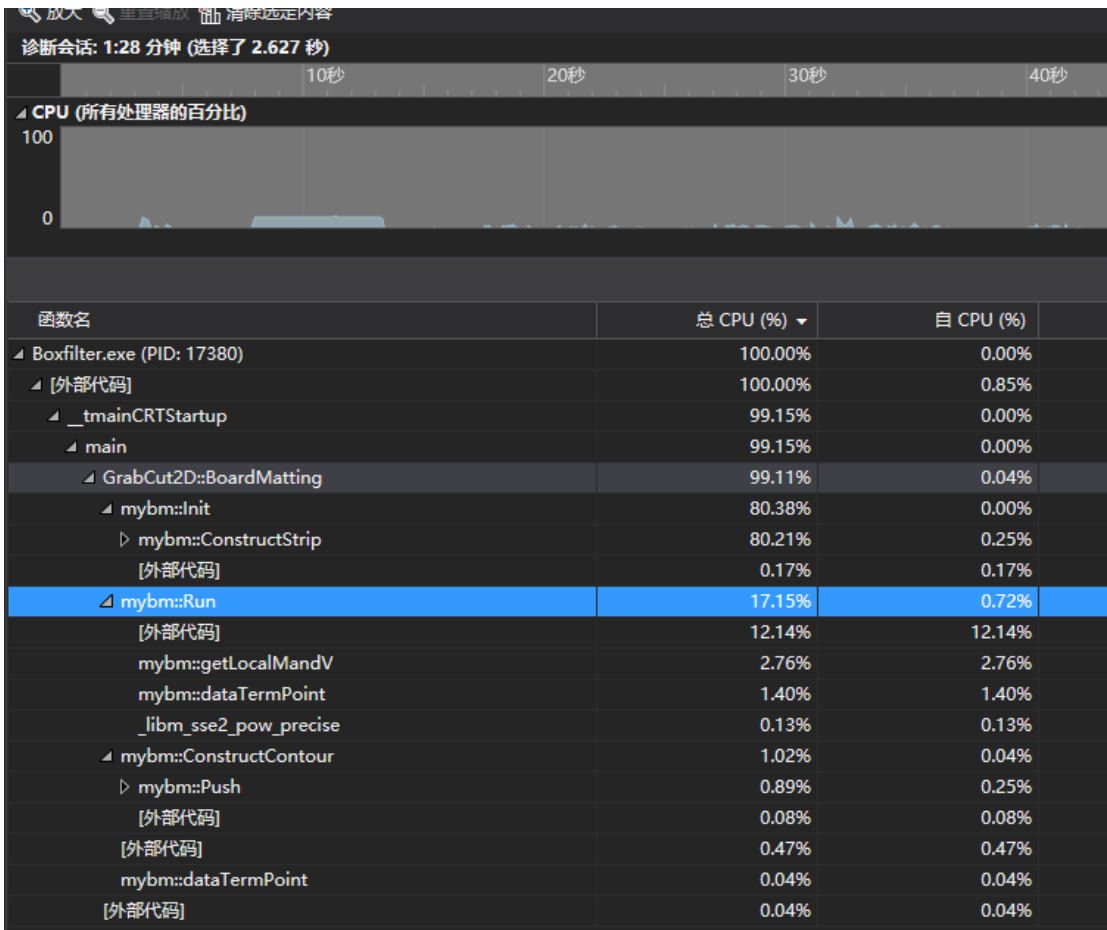
没有可用于此显示器的笔或触控输入

在 Grab Cut 硬分割阶段，对上面的圣诞节的图片 (768*600) 进行 CPU 性能评测结果如下：



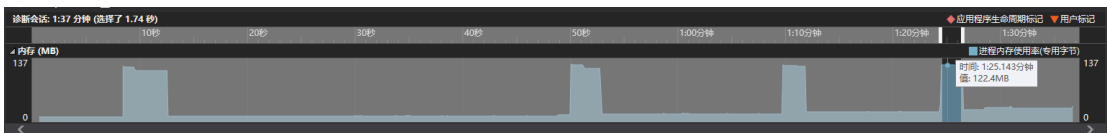
截取的部分是第一次进行 Grab Cut 迭代的部分，第一次用了 5.13s，但是后面的迭代中这个过程变得非常迅速，在上面的处理器占用比中没有明显的长时间占用。可以看到其中性能瓶颈主要是出现在图像分割中的 maxflow 阶段。

Border Matting 阶段处理一共 2.63s, 可以看到主要的 CPU 用时在于初始化 Strip, 就是初始化并有序化轮廓, 这个部分是遍历图中每个点, 如果是在轮廓点的附近的话, 然后把他加入集合复杂度为 $O(N*M)$ N 为轮廓点的个数, M 为轮廓点及轮廓点的邻居的个数。而能量最小化的过程因为前面的有序化所以非常快速, 只用了 17% 的时间在处理 Border Matting 的时候。



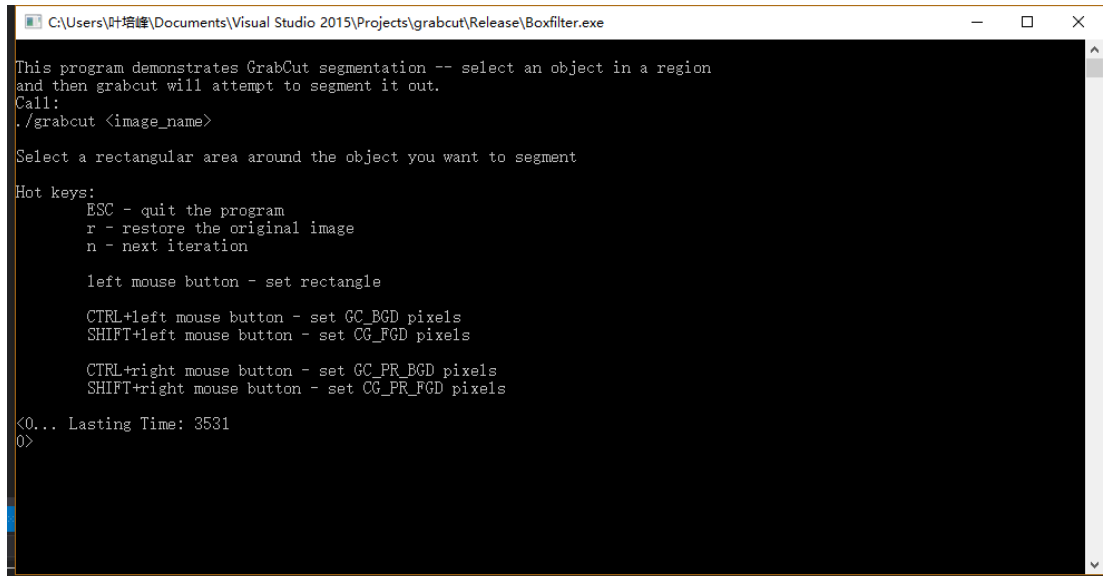
内存使用情况:

可以看到 Grab Cut 硬分割阶段最高内存使用量不超过 137MB, 而 Border Matting 阶段稳定 33MB 左右。



3.3 Grab Cut 硬分割阶段与 openCV 实现比较

同样处理上面的圣诞图，我写的 Grab Cut 第一次迭代用时如下：



```
C:\Users\叶培峰\Documents\Visual Studio 2015\Projects\grabcut\Release\Boxfilter.exe

This program demonstrates GrabCut segmentation -- select an object in a region
and then grabcut will attempt to segment it out.
Call:
./grabcut <image_name>

Select a rectangular area around the object you want to segment

Hot keys:
  ESC - quit the program
  r - restore the original image
  n - next iteration

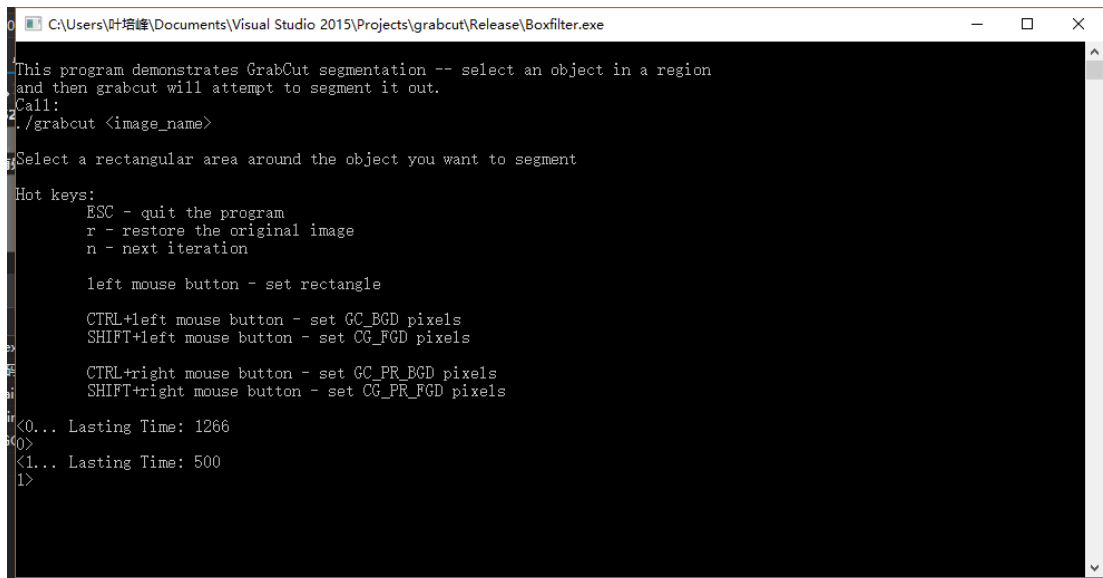
  left mouse button - set rectangle

  CTRL+left mouse button - set GC_BGD pixels
  SHIFT+left mouse button - set CG_FGD pixels

  CTRL+right mouse button - set GC_PR_BGD pixels
  SHIFT+right mouse button - set CG_PR_FGD pixels

<0... Lasting Time: 3531
0>
```

OpenCV 自带的 Grab Cut 用时如下：



```
C:\Users\叶培峰\Documents\Visual Studio 2015\Projects\grabcut\Release\Boxfilter.exe

This program demonstrates GrabCut segmentation -- select an object in a region
and then grabcut will attempt to segment it out.
Call:
./grabcut <image_name>

Select a rectangular area around the object you want to segment

Hot keys:
  ESC - quit the program
  r - restore the original image
  n - next iteration

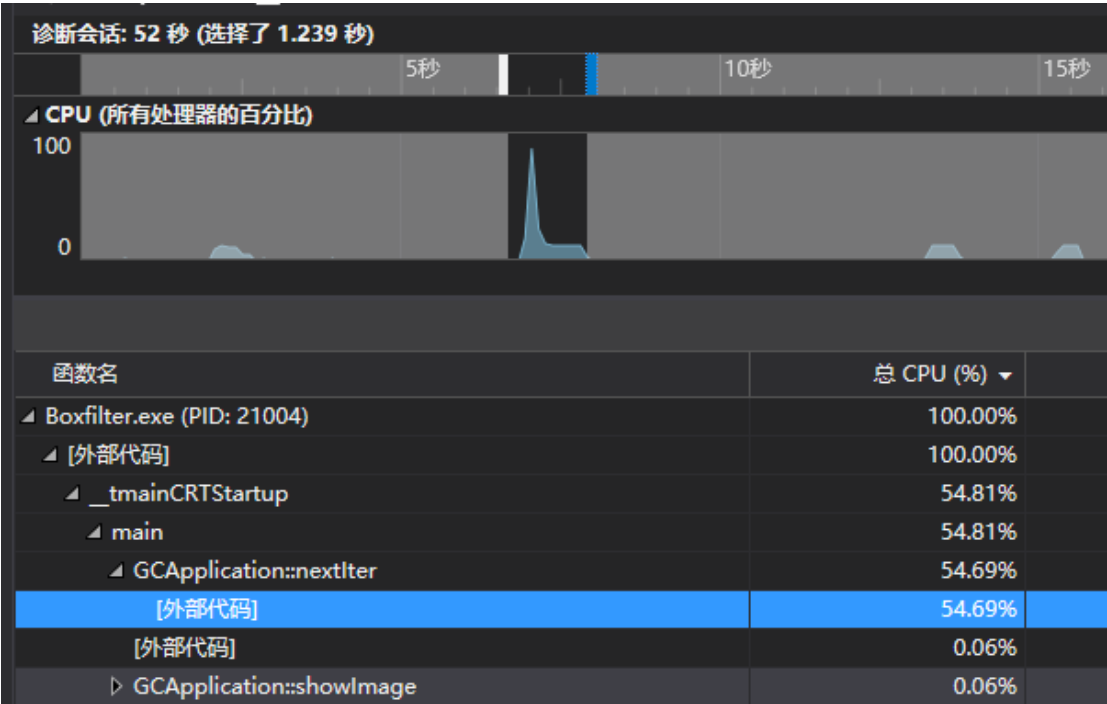
  left mouse button - set rectangle

  CTRL+left mouse button - set GC_BGD pixels
  SHIFT+left mouse button - set CG_FGD pixels

  CTRL+right mouse button - set GC_PR_BGD pixels
  SHIFT+right mouse button - set CG_PR_FGD pixels

<0... Lasting Time: 1266
0>
<1... Lasting Time: 500
1>
```

可见 openCV 的 GrabCut 比我写的快了非常多，用 VS 自带的性能分析器观察：



OpenCV 自带的 GrabCut 在计算的时候 CPU 利用率远高于我写的，我写的在计算的时候 CPU 利用率不超过 15%。但是 OpenCV 的 GrabCut 却可以达到接近 100% 所以，这就不奇怪为什么他的比我的快那么多了。

这里我估计是因为 openCV 对多核处理器的优化更加完善，毕竟我的单核单线程的程序利用率不高也是很正常的。

参考文献

- [1] Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D Images
- [2] “GrabCut” — Interactive Foreground Extraction using Iterated Graph Cuts
- [3] <http://blog.csdn.net/zouxy09/article/details/8534954>