计算机视觉作业3: 模板匹配技术

唐麒 21120299
qitang@bjtu.edu.cn

摘要—模板匹配是一项在一幅图像中寻找与另一幅模板图像最匹配 (相似) 部分的技术。本文分别采用相关匹配 (Correlation Matching)、基于 Hausdorff 距离匹配以及对场景图像距离变换 (Distance Transform) 的 Hausdorff 距离匹配三种方法对给定的模板图像在场景图像中进行定位。不同方法的定位精度和定位效率将在实验部分进行简单探讨。

关键词—计算机视觉、模板匹配、相关匹配、Hausdorff 距离

1 简介

模板匹配是通过一张模板图片去另一张图中找到与模板相似部分的一种算法。一般是通过滑窗的方式在待匹配的图像上滑动，通过比较模板与子图的相似度，找到相似度最大的子图。这种算法最核心部分在于如何设计一个相似性度量函数。一般最容易想到的相似性度量函数便是欧式距离。

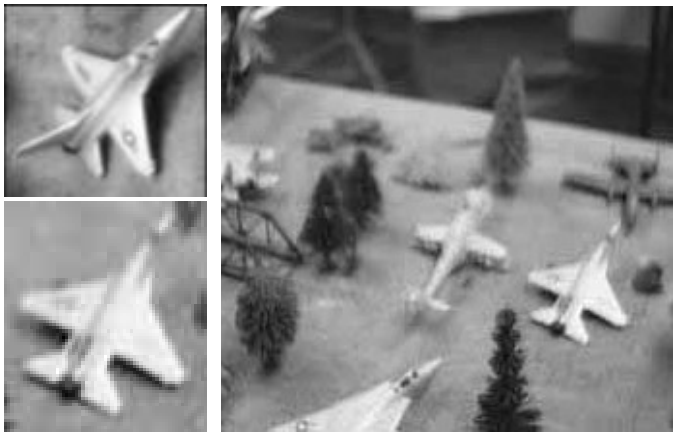
但有许多因素会使得模版匹配带来巨大的开销，例如场景图像的大小、模板的数量和角度等。针对不同的因素，我们可以采用不同的方法来减少计算开销。例如，可以利用图像金子塔来同时降低模板图像和场景图像的分辨率，并在较低的分辨率进行匹配，进而在更高的分辨率逐步优化；或是利用模版图像的边缘和场景图像的边缘来进行匹配。对于后者而言，欧式距离显然已不再适合对图像边缘之间的相似性进行度量，因而需要设计一个新的相似性度量函数。

Hausdorff 距离则被用于描述两组点集之间相似程度，是两个点集之间距离的一种定义形式。通俗来讲，Hausdorff 距离描述的是一个集合到另一个集合中最近点的最大距离。但 Hausdorff 距离本质上并没有减少匹配过程中的计算开销。为了实现更高效的搜索和匹配，可以考虑先对场景图像进行距离变换，即利用切比雪夫距离（Chebyshev Distance）预先计算生成边缘的距离图像，以此达到降低匹配复杂度的目的。

本文剩余部分将相关匹配、基于 Hausdorff 距离匹配以及对场景图像距离变换的 Hausdorff 距离匹配三种方法的原理、实现及实验进行介绍，所有实验涉及的方法、函数实现均基于 Python 语言。

2 方法

这一章节将对相关匹配、基于 Hausdorff 距离匹配以及对场景图像距离变换的 Hausdorff 距离匹配三种模版匹配方法进行介绍，其定位结果将在实验部分给出。



(a) (b)

图 1: 本文实验所使用的模板图像和场景图像：(a) 模版图像, (b) 场景图像.

2.1 相关匹配方法

一般而言，欧式距离是最容易想到的相似性度量函数。对于给定的模版图像 M 和场景图像 N，其大小分别为 $m \times m$ 和 $n \times n$ $(m > n)$，其欧式距离定义如下：

$$SSD = \sum_{i=1}^{n}\sum_{j=1}^{n}[M(i,j) - N(i,j)]^2 \tag{1}$$

将其展开可得：

$$\begin{aligned} SSD = &\sum_{i=1}^{n}\sum_{j=1}^{n}[M(i,j)]^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n}M(i,j)\cdot N(i,j) \\ &+ \sum_{i=1}^{n}\sum_{j=1}^{n}[N(i,j)]^2 \end{aligned} \tag{2}$$

可以看出，只有第二项是有意义的，因为第一项和第三项的值在模板图像和窗口选定后是固定的。对于欧式距离而言，其值越大代表二者越不相似，即第二项的值越小则越不相似。将第二项进行归一化，可以得到相关度量的计算公式，如下：

$$C = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} M(i,j)N(i,j)}{\left[\sum_{i=1}^{n}\sum_{j=1}^{n} M(i,j)^2 \sum_{i=1}^{n}\sum_{j=1}^{n} N(i,j)^2\right]^{1/2}} \tag{3}$$

代码实现如下：

```python
# 计算相关
def calculate_correlation(m, n):
    return np.sum(m * n) / np.sqrt(np.sum(m ** 2)
        * np.sum(n ** 2))
```

main.py

对于相关匹配而言，为了更加准确地计算模板图像与场景图像的相关度量，还需要分别对二者进行归一化处理，代码如下：

```python
# 相关匹配
def correlation_match(template, scene):
    scene = cv2.cvtColor(scene, cv2.COLOR_BGR2GRAY
        )
    template = template[2:template.shape[0] - 2,
        2:template.shape[1] - 2]

    max_template = np.max(template)
    min_template = np.min(template)

    max_scene = np.max(scene)
    min_scene = np.min(scene)

    template = (template - min_template) / (
        max_template - min_template)
    scene = (scene - min_scene) / (max_scene -
        min_scene)

    template_h, template_w = template.shape
    scene_h, scene_w = scene.shape
    strides = scene.itemsize * np.array([scene_w,
        1, scene_w, 1])
    new_h = scene_h - (template_h - 1)
    new_w = scene_w - (template_w - 1)
    scene_slides = np.lib.stride_tricks.as_strided
        (scene, shape=(new_h, new_w, template_h,
        template_w),strides=strides)
    print(scene_slides.shape)
    correlation = np.zeros((new_h, new_w))
    for h in range(new_h):
        for w in range(new_w):
            correlation[h][w] =
                calculate_correlation(template,
                scene_slides[h][w])
    position = max_value_position(correlation)

    return [*position, *template.shape]
```

main.py

2.2 Hausdorff 距离匹配方法

Hausdorff 距离是描述两组点之间相似程度的一种度量，它是集合和集合之间距离的一种定义形式。给定集合 $A = \{a_1, a_2, a_3, \cdots, a_n\}$ 和集合 $B = \{b_1, b_2, b_3, \cdots, b_n\}$，Hausdorff 距离的计算公式为:

$$h(A,B) = \max_{a\in A}\left\{\min_{b\in B}\{d(a,b)\}\right\} \tag{4}$$

$$H(A,B) = \max\{h(A,B), h(B,A)\} \tag{5}$$

$h(A,B)$ 即在计算出集合 A 中每个点 a_i 与集合 B 中相距最近的点 b_i 的距离的基础上，选出其中的最大值作为 $h(A,B)$ 的值。

代码实现如下：

```python
# 计算 hausdorff 距离
def calculate_hausdorff_distance(a, b):
    h_a_b = np.min((abs(a.reshape([-1, 1]).repeat(
        b.shape[0], 1) - b)).reshape([len(a.
        flatten()), -1]), axis=-1)
    h_b_a = np.min((abs(b.reshape([-1, 1]).repeat(
        a.shape[0], 1) - a)).reshape([len(b.
        flatten()), -1]), axis=-1)
    return np.max((np.max(h_a_b), np.max(h_b_a)))
```

main.py

Hausdorff 距离匹配方法的代码实现如下：

```python
# hausdorff 距离匹配
def hausdorff_match(idx, template, scene):
    t_dict = {"0": [150, 250], "1": [50, 250]}
    t = t_dict[str(idx)]
    template = cv2.Canny(template[2:template.shape
        [0] - 2, 2:template.shape[1] - 2], t[0], t
        [1])
    if idx == 0:
        scene = cv2.Canny(scene, 150, 250)
        scene_edge = scene.copy()
    image_show(scene)
    image_show(template)

    stride = 3
    template_h, template_w = template.shape
    scene_h, scene_w = scene.shape
    strides = scene.itemsize * np.array([scene_w *
        stride, 1 * stride, scene_w, 1])
    new_h = (scene_h - template_h) // stride + 1
    new_w = (scene_w - template_w) // stride + 1
```

```
    scene_slides = np.lib.stride_tricks.as_strided
        (scene, shape=(new_h, new_w, template_h,
        template_w),strides=strides)
    template_x, template_y = np.where(template >
        0)
    distance = np.zeros((new_h, new_w))
    for h in range(new_h):
        for w in range(new_w):
            win_x, win_y = np.where(scene_slides[h
                ][w] > 0)
            distance[h][w] = np.max([
                calculate_hausdorff_distance(
                template_x, win_x),
                calculate_hausdorff_distance(
                template_y, win_y)])
    position = min_value_position(distance)
    if idx == 0:
        return [(position[0] - 1) * stride, (
            position[1] - 1) * stride, *template.
            shape], scene_edge
    else:
        return [(position[0] - 1) * stride, (
            position[1] - 1) * stride, *template.
            shape]
```

main.py

2.3 图像距离变换方法

图像距离变换主要是针对场景图像进行处理，在提取场景图像边缘的基础上，计算图像矩阵每个位置到提取的边缘最近的切比雪夫距离，从而得到距离矩阵，其计算公式如下：

$$\mathrm{DT}(p) = \min\{\mathrm{d}(\mathrm{p},\mathrm{q})\} \text{ with } \mathrm{q} \in \mathrm{B} \tag{6}$$

$$d(p,q) = |i-x| + |j-y| \tag{7}$$

其中，p 为图像 P 中的点，B 是由 P 的边缘点 q 构成的集合，(i,j) 和 (x,y) 分别为 p 和 q 的坐标。

代码实现为:

```
# 计算距离变换
def calculate_hausdorff_distance_transform(scene):
    row, col = scene.shape
    distance = np.ones((row, col))
    max = np.max(scene)
    r, c = np.where(scene == max)
    for i in range(row):
        for j in range(col):
            distance[i, j] = np.min(abs(r - i) +
                abs(c - j))
    return distance
```

main.py

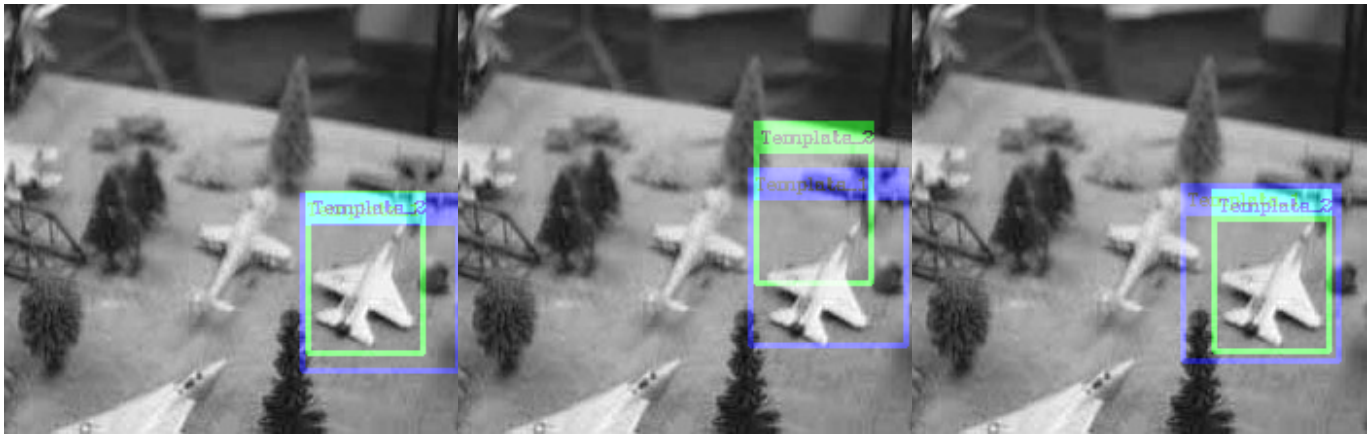
经过提前计算得到图像矩阵每个位置的变换距离后，在模版匹配的过程中，只需要在距离矩阵中进行查找即可很快计算得到相似性度量，代码实现如下：

```
# 距离变换匹配
def distance_transform_match(idx, template, scene)
    :
    t_dict = {"0": [150, 250], "1": [50, 250]}
    t = t_dict[str(idx)]
    template = cv2.Canny(template[2:template.shape
        [0] - 2, 2:template.shape[1] - 2], t[0], t
        [1])
    if idx == 0:
        scene = cv2.Canny(scene, 150, 250)
        scene_edge = scene.copy()
    image_show(scene)
    scene = calculate_hausdorff_distance_transform
        (scene)
    image_show(scene)
    stride = 1
    template_h, template_w = template.shape
    scene_h, scene_w = scene.shape
    strides = scene.itemsize * np.array([scene_w *
         stride, 1 * stride, scene_w, 1])
    new_h = (scene_h - template_h) // stride + 1
    new_w = (scene_w - template_w) // stride + 1
    scene_slides = np.lib.stride_tricks.as_strided
        (scene, shape=(new_h, new_w, template_h,
        template_w),strides=strides)
    distance = np.zeros((new_h, new_w))
    print(scene_slides.shape)
    for h in range(new_h):
        for w in range(new_w):
            distance[h][w] = np.mean(template *
                scene_slides[h][w])
    hausdorff_distance_show(np.arange(new_w), np.
        arange(new_h), distance)
    position = min_value_position(distance)
    if idx == 0:
        return [(position[0] - 1) * stride, (
            position[1] - 1) * stride, *template.
            shape], scene_edge
    else:
        return [(position[0] - 1) * stride, (
            position[1] - 1) * stride, *template.
            shape]
```

main.py

3 实验

这一章节将会利用上一章介绍的三种模版匹配方法在模版图像和场景图像上进行实验，并对其定位精度、定位效率进行展示和分析。



(a) (b) (c)

图 2: 不同方法的模版匹配结果: (a) 相关匹配方法, (b) Hausdorff 距离匹配方法, and (c) 图像距离变换匹配方法

如图 2 所示，(a)、(b)、(c) 分别为相关匹配方法、Hausdorff 距离匹配方法和图像距离变换方法的匹配结果。可以看到，相关匹配方法和图像距离变换方法的定位精度优于 Hausdorff 距离匹配方法，尤其就第二幅模版图像而言。而对比两幅模版的定位精度时，显然可以看到对于相关匹配方法和图像距离变换方法，第二幅模版图像的定位精度优于第一幅模版图像，其原因在于第一幅模版图像与场景图像的相似内容有一定的尺寸、旋转偏差，从而导致第一幅模版图像的定位精度欠佳。但同样对于 Hausdorff 距离匹配方法，第一幅模版图像的定位精度却优于第二幅模版图像，这可能是由于边缘检测结果欠佳造成的，如设置了不合适的阈值，导致边缘点较少等。

(a) (b)

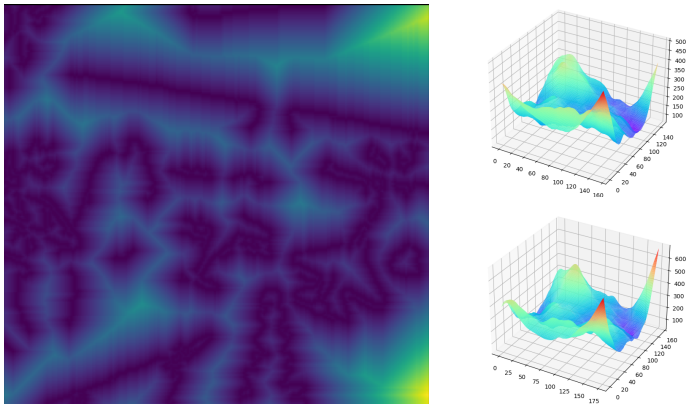
图 3: 基于 Canny 算法提取的图像边缘：(a) 模版图像, (b) 场景图像.

如图 3 所示，为利用 Canny 算法提取的模版图像和场景图像的边缘。可以看到，第一幅模版图像的边缘较为连续，而第二幅模版图像的边缘在右上角有类似于“噪声”的点，这些“噪声点”会在一定程度上影响到 Hausdorff 距离的计算，进而影响到最后的定位结果。

表 1: 不同匹配方法的定位效率比较

	相关匹配方法	Hausdorff 距离匹配方法	图像距离变换方法
运行时间（秒）	2.66	15.48	2.28

从定位效率来看，如表 1 所示，相关匹配方法和图像距离变换方法的定位效率远高于 Huasdorff 距离匹配方法，如图 4 所示，图像距离变换的方法是预先计算出了图像矩阵中距离边缘点最近的切比雪夫距离，从而在匹配过程中只需要查找得到当前位置的距离，以此提高定位效率。出自之外，其定位精度相对于 Huasdorff 距离匹配方法也有所提升，可能是由于相对于 Huasdorff 距离，该方法鲁棒性更强。



(a) (b)

图 4: 图像距离变换可视化结果：(a) 场景图像距离变换可视化, (b) 模版图像距离分布.

4 总结

本文主要对模版匹配的三种方法进行了探讨和实验，即相关匹配方法、Hausdorff 距离匹配方法和图像距离变换方法。

在对模版图像进行归一化处理后，相关匹配方法的定位精度略高于 Hausdorff 距离匹配方法，这是由于在边缘检测时可能存在的噪声点对 Hausdorff 距离的计算影响较大，从而会影响到后续的定位精度。而对于定位效率而言，图像距离变换方法的定位效率远高于 Hausdorff 距离匹配方法，这是由于 Hausdorff 距离匹配方法只是一种点集之间的相似性度量，实质上并没有减少运算量，而图像距离变换方法可以预先计算出图像中每个位置与边缘点的距离，从而在匹配过程中仅需查找即可得到匹配需要的距离，以此减少运算量，提升定位效率。