

连连看综合实验

- 无 36
- 李思涵
- 2013011187

原创性声明

此实验的代码 & 实验报告均为原创。

第一章 制作自己的连连看

1.1 熟悉游戏

嗯嗯背景里面的两只皮卡丘好萌哇 😊

1.2 实现判断是否可消除的功能

首先，我们来判断直线连接的情况，即0个转折的情况。

这种情况很容易判断，只需要判断两个块是否在同一条直线上，同时其间的块都是0。代码实现如下：

```
%% zero_turn: Judge whether two blocks have only spaces in between.
function bool = zero_turn(mtx, x1, y1, x2, y2)
    left = min(x1, x2);
    right = max(x1, x2);
    down = min(y1, y2);
    up = max(y1, y2);

    bool = (x1 == x2 && all(mtx(x1, down+1:up-1) == 0) || ...
            y1 == y2 && all(mtx(left+1:right-1, y1) == 0));
end
```

然后，我们来判断拐弯连，即1个转折的情况。

通过观察我们不难发现，这种情况中的连线，实际上就是以两个块为顶点的矩形的两条相邻边，而这样的可能连接有两条。对于两条中的某一条，都可以用如下情况判断：

1. 确定转折点的坐标；
2. 判断转折点和两个块之间，以及转折点本身是否都为空。

具体代码实现如下，注意到这里我们调用了 `zero_turn` 来进行一部分判断，从而避免了在这里判断两个块的相对位置。

```
%% one_turn: Judge whether two blocks can be linked in one turn.
function bool = one_turn(mtx, x1, y1, x2, y2)
    bool = (zero_turn(mtx, x1, y1, x2, y1) && ...
            zero_turn(mtx, x2, y1, x2, y2) && ...
            mtx(x2, y1) == 0 || ...
            zero_turn(mtx, x1, y1, x1, y2) && ...
            zero_turn(mtx, x1, y2, x2, y2) && ...)
```

```

        mtx(x1, y2) == 0);
    end

```

最后，我们来判断拐俩弯的情况。为了简化函数，我们假设已经排除了前两种情况。注意到这种情况中，连线可能出现在地图边界以外，我们先使用 `padarray` 将地图向外用0扩张一圈：

```

%% two_turns: Assuming not zero or one turn.
function bool = two_turns(mtx, x1, y1, x2, y2)
    % Pad the matrix for the possible links.
    mtx = padarray(mtx, [1, 1]);
    x1 = x1 + 1;
    y1 = y1 + 1;
    x2 = x2 + 1;
    y2 = y2 + 1;

```

然后我们使用如下的策略来判断：

1. 从块1开始，向四个方向尽可能延伸，直到遇到非零块或边界；
2. 对于延伸的每一个块，判断其是否能与块2拐弯连，若能则说明块1能与块2拐俩弯连。

具体代码实现如下：

```

direction = [ 0 1
              0 -1
              1 0
              -1 0];

for k = 1:4
    delta = direction(k, :);
    pos = [x1 y1] + delta;

    % Toward if possible.
    while all(pos > [0 0] & pos <= size(mtx)) && mtx(pos(1), pos(2)) == 0
        if one_turn(mtx, pos(1), pos(2), x2, y2)
            bool = 1;
            return
        end
        pos = pos + delta;
    end
end

bool = 0;

```

最后，利用这三个函数，我们可以很简单地实现 `detect` 函数：

```

function bool = detect(mtx, x1, y1, x2, y2)
    % ===== 参数说明 =====

    % 输入参数中，mtx为图像块的矩阵，类似这样的格式：
    % [ 1 2 3;
    %   0 2 1;
    %   3 0 0 ]
    % 相同的数字代表相同的图案，0代表此处没有块。
    % 可以用[m, n] = size(mtx)获取行数和列数。
    % (x1, y1)与 (x2, y2) 为需判断的两块的下标，即判断mtx(x1, y1)与mtx(x2, y2)
    % 是否可以消去。

    % 注意mtx矩阵与游戏区域的图像不是位置对应关系。下标(x1, y1)在连连看界面中
    % 代表的是以左下角为原点建立坐标系，x轴方向第x1个，y轴方向第y1个

    % 输出参数bool = 1表示可以消去，bool = 0表示不能消去。

```

```

%% 在下面添加你的代码O(n_n)0
bool = mtx(x1, y1) == mtx(x2, y2) && ...
    (zero_turn(mtx, x1, y1, x2, y2) || ...
    one_turn(mtx, x1, y1, x2, y2) || ...
    two_turns(mtx, x1, y1, x2, y2));

end

```

最后在移走 detect.p，移入 detect.m，在游戏中测试我们的算法...成功 ♪(▽〃)

1.3 实现外挂功能

我们设计这样的 omg 算法：

1. 由相同图案的块才能相消这一条件，找出所有的潜在相消对；
2. 遍历所有潜在相消对，若发现能相消的则记录并消除，同时去除消除后不再存在的潜在相消对；
3. 重复步骤 2，直到一次遍历中没有发生消除。

步骤 2 中剔除潜在相消对的目的是减少后续循环量，否则后期的循环中会重复遍历大量无效的潜在相消对，降低算法的效率。

代码实现如下：

```

function steps = omg(mtx)
% ----- 输入参数说明 -----

% 输入参数中，mtx为图像块的矩阵，类似这样的格式：
% [ 1 2 3;
%   0 2 1;
%   3 0 0 ]
% 相同的数字代表相同的图案，0代表此处没有块。
% 可以用[m, n] = size(mtx)获取行数和列数。

% 注意mtx矩阵与游戏区域的图像不是位置对应关系。下标(x1, y1)在连连看界面中
% 代表的是以左下角为原点建立坐标系，x轴方向第x1个，y轴方向第y1个

% ----- 输出参数说明 ----- %

% 要求最后得出的操作步骤放在steps数组里，格式如下：
% steps(1)表示步骤数。
% 之后每四个数x1 y1 x2 y2，代表把mtx(x1,y1)与mtx(x2,y2)表示的块相连。
% 示例： steps = [2, 1, 1, 1, 2, 2, 1, 3, 1];
% 表示一共有两步，第一步把mtx(1,1)和mtx(1,2)表示的块相连，
% 第二步把mtx(2,1)和mtx(3,1)表示的块相连。

%% ----- 请在下面加入你的代码 O(n_n)0~ -----

steps = [];

msize = size(mtx);
patterns = unique(mtx);
patterns = patterns(patterns ~= 0);

% Generate possible pairs.
pairs = [];
for pattern = patterns'
    poses = find(mtx == pattern);
    len = length(poses);
    for k1 = 1:len-1
        for k2 = k1+1:len
            pairs = [pairs
                    poses(k1) poses(k2)];
        end
    end
end

used = 1; % To enter the loop.

```

```

while any(used)
    pair_num = size(pairs, 1);
    used = logical(zeros(pair_num, 1));

    for k = 1:pair_num
        if used(k)
            continue
        end
        index1 = pairs(k, 1);
        index2 = pairs(k, 2);

        [x1, y1] = ind2sub(msize, index1);
        [x2, y2] = ind2sub(msize, index2);
        if detect(mtx, x1, y1, x2, y2) % A match.
            mtx(index1) = 0;
            mtx(index2) = 0;
            steps = [steps x1 y1 x2 y2];
            used(any((pairs == index1 | pairs == index2))) = 1;
        end
    end
    pairs(used, :) = []; % Remove used.
end

steps = [length(steps) / 4, steps];
end

```

让我们用注释中的样例测试我们的算法：

```

m = [1 2 3
      0 2 1
      3 0 0];
omg(m)
% ans =
%
% Columns 1 through 11
%
%      3      1      2      2      2      3      1      1      3      1      1
%
% Columns 12 through 13
%
%      2      3

```

可以看到，我们的实现的算法给出了正确的结果，乌拉~

第二章 攻克别人的连连看

2.1 对屏幕截图分块

为了借助傅里叶变换实现图像分块，我们首先要确定每块的大小，以及左上方的空白大小。这只需要对行/列去除直流分量，并取平均，然后用 `fft` 处理，提取出基频分量即可。其中，基频分量的提取使用了实验一中的实现方式（在最高峰频率 $1/2$, $1/3$, $1/4$ 处寻找基频）。代码如下：

```

%% find_horizontal_period: Find
function [period, blank] = find_horizontal_period(img)
% Remove AC component.
hline = mean(double(img) - mean(mean(img)));

len = length(hline);
f = [0:len-1] / len;
f_domain = fft(hline);

index = find_baseband_index(f_domain); % Find the baseband.

```

```

period = round(1 / f(index));
phase_pixel = round(angle(f_domain(index)) / (2 * pi * f(index)));
if phase_pixel <= 0
    blank = -phase_pixel;
else
    blank = period - phase_pixel;
end
end

%% find_baseband_index: Find the index of the baseband
function baseband = find_baseband_index(f_domain)
    f = abs(f_domain);
    [max_wight, max_band] = max(f);

    %% Find base band.
    baseband = max_band;
    for ratio = [2, 3, 4]
        band = (max_band - 1) / ratio + 1;
        [maximum, index] = max_around(f, band, 0.05);
        if maximum > 0.8 * max_wight
            baseband = index;
        end
    end
end

%% max_around: Find maximum around a index.
function [maximum, index] = max_around(x, index, error_ratio)
    from = ceil(index * (1 - error_ratio));
    to = floor(index * (1 + error_ratio));
    [maximum, index] = max(x(from:to));
    index = index + from - 1;
end

```

然后，我们只需要利用上面得到的信息，将图像分割开来即可：

```

%% divide_img: Divide image into blocks
function [blocks, h_period, h_blank, v_period, v_blank] = divide_img(img)
    [h_period, h_blank] = find_horizontal_period(img);
    [v_period, v_blank] = find_horizontal_period(img');

    blocks = cell(floor((size(img) - [v_blank h_blank]) ./ ...
        [v_period h_period]));

    row_max = size(img, 1) - v_period + 1;
    col_max = size(img, 2) - h_period + 1;

    k = 1;
    blocks = blocks';
    for row = v_blank+1:v_period:row_max
        for col = h_blank+1:h_period:col_max
            blocks{k} = img(row:row+v_period-1, col:col+h_period-1);
            k = k + 1;
        end
    end
    blocks = blocks';
end

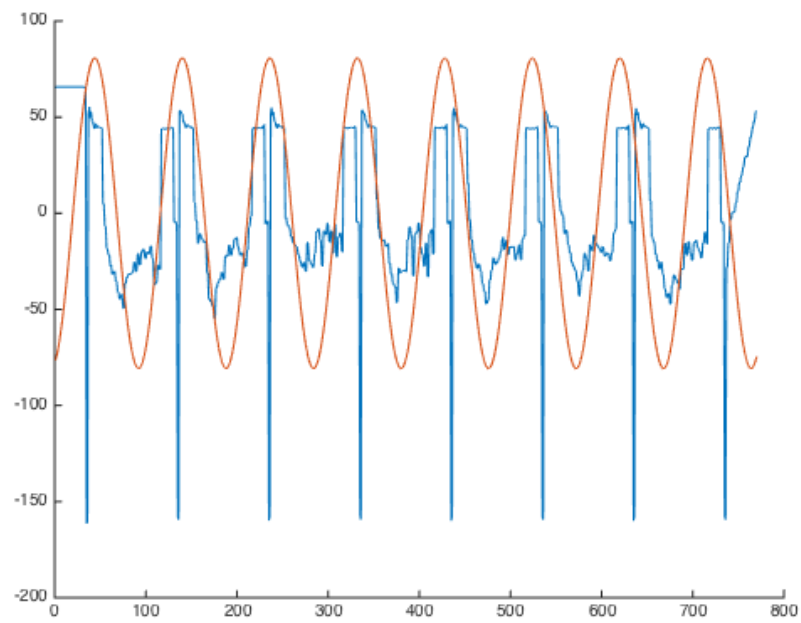
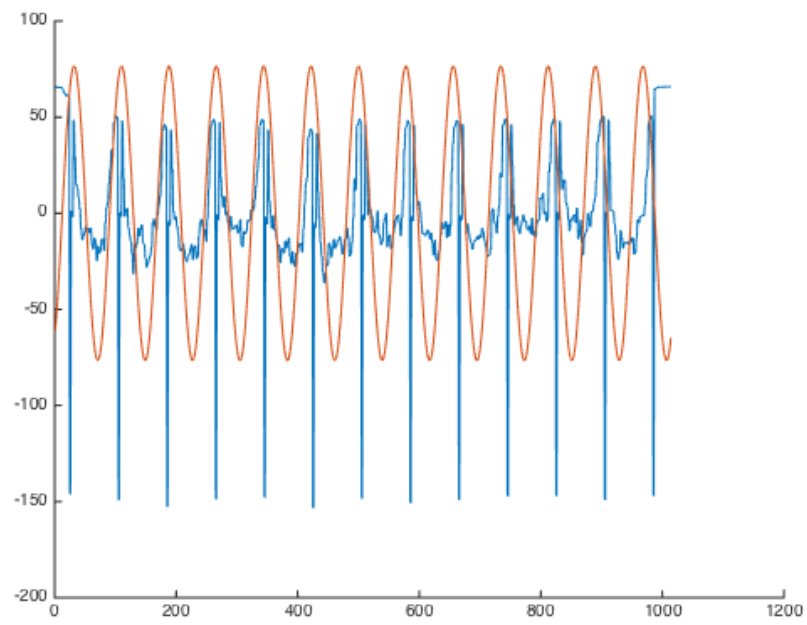
```

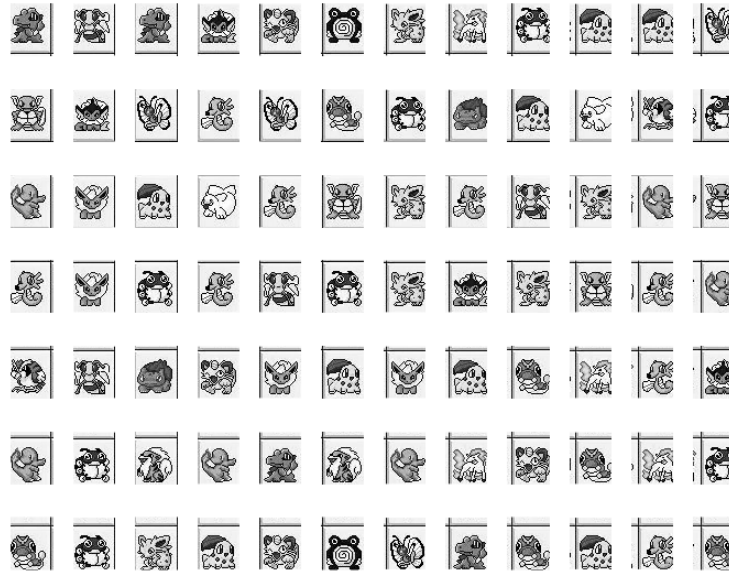
让我们来检测一下分块结果：

```

imgs_truth = divide_img(graygroundtruth);
show_divided_img(imgs_truth);

```



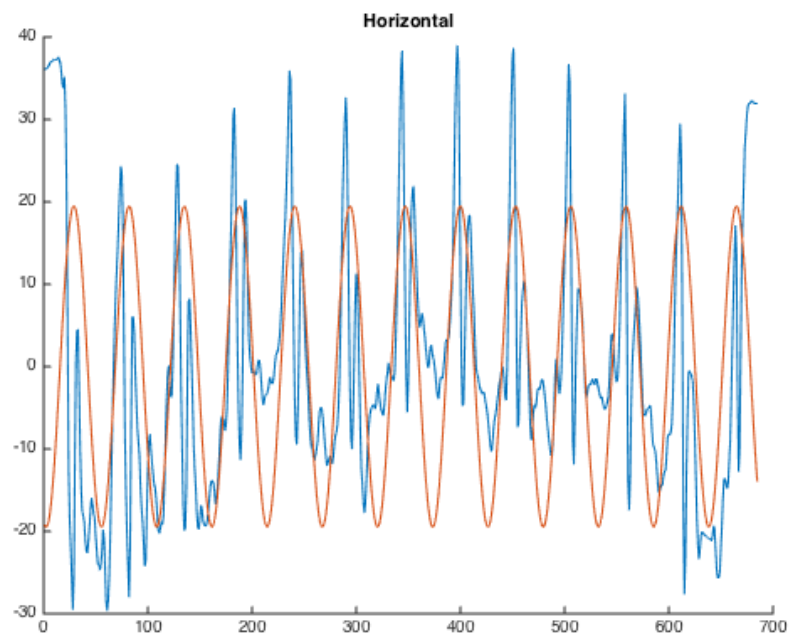


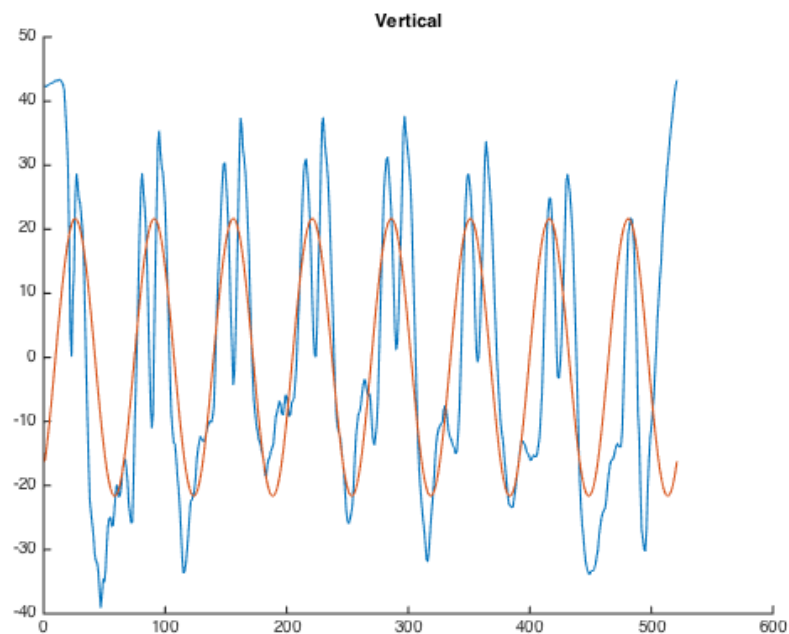
可以看到，傅里叶变换的基频确实反映出了块的周期性，而我们也得到了基本正确的分块结果。

2.2 对摄像头采集到的图像分块

让我们对摄像头采集到的图像进行同样的处理：

```
imgs = divide_img(graycapture);
show_divided_img(imgs);
```





可以看到，我们的分块结果仍然基本正确。

从较上方的两幅图中可以看出，横向及纵向交流均值的频谱相对于 2.1 有比较大的扰动，例如，横向似乎出现了一个频率很低的低频分量。在挑选基频时，这显然会给我们的程序带来一定的困扰。

不过，最后正确的结果也说明，这种利用傅里叶变换的分块方法有着不错的抗干扰能力，能在图像有一定扰动、失焦、变形的情况下分析出正确结果。

2.3 计算分块相似性

为了使各个分块的纹理更加突出，我们先对每个分块进行高通滤波：

```
% highpass_img: Keep only the high-freq component of the img.
function high_freq_img = highpass_img(img, order)
```



```

% Ensure an even order.
if mod(order, 2)
    order = order + 1;
end

filter_1d = fir1(order, 0.5, 'high');
filter_2d = zeros(order + 1);
center = order / 2 + 1;

for row = 1:order+1
    for col = 1:order+1
        r = round(sqrt((row - center)^2 + (col - center)^2));
        if r > center - 1
            value = 0;
        else
            value = filter_1d(center - r);
        end
        filter_2d(row, col) = value;
    end
end

high_freq_img = filter2(filter_2d, img);

```

然后对各对图像块之间进行匹配:

```

%% match_imgs: Match images
function similarity = match_imgs(imgs, order, margin_ratio)
    img_num = numel(imgs);
    high_imgs = cell(img_num, 1);
    similarity = eye(img_num);

    margin = round(margin_ratio * size(imgs{1}));

    for k = 1:img_num
        high_imgs{k} = highpass_img(imgs{k}, order);
    end

    for k1 = 1:img_num-1
        for k2 = k1+1:img_num
            img1 = high_imgs{k1};
            img2 = high_imgs{k2};

            corr1 = max(max(normxcorr2(img1(1+margin(1):end-margin(1), ...
                                     1+margin(2):end-margin(2)), ...
                                     img2)));
            corr2 = max(max(normxcorr2(img2(1+margin(1):end-margin(1), ...
                                     1+margin(2):end-margin(2)), ...
                                     img1)));
            corr = max([corr1 corr2]);

            similarity(k1, k2) = corr;
            similarity(k2, k1) = corr;
        end
    end
end

```

对结果进行排序:

```

%% sort_match: Sort matching result.
function [matches, values] = sort_match(similarity)
    img_num = length(similarity);

    % Remove useless elements.
    [values, index] = sort(similarity(:), 'descend');
    from = mod(index - 1, img_num) + 1;
    to = ceil(index / img_num);
    matches = [from to];

    values = values(from < to);

```

```
matches = matches(from < to, :);
```

我们先将匹配结果存至变量中：





















```
sim = match_imgs(imgs, 20, 0.2);  
sim_truth = match_imgs(imgs_truth, 20, 0.2);  
[matches, values] = sort_match(sim);
```

同时，我们编写函数展示匹配结果：

```
% show_matches: Show matches in two columns.  
function show_matches(imgs, matches, values)  
    row = length(matches);  
  
    for k = 1:row  
        match = matches(k, :);  
  
        subplot(row, 2, 2 * k - 1);  
        imshow(imgs{match(1)});  
        title(['Block ' num2str(match(1))]);  
        ylabel(values(k))  
  
        subplot(row, 2, 2 * k);  
        imshow(imgs{match(2)});  
        title(['Block ' num2str(match(2))]);  
    end
```

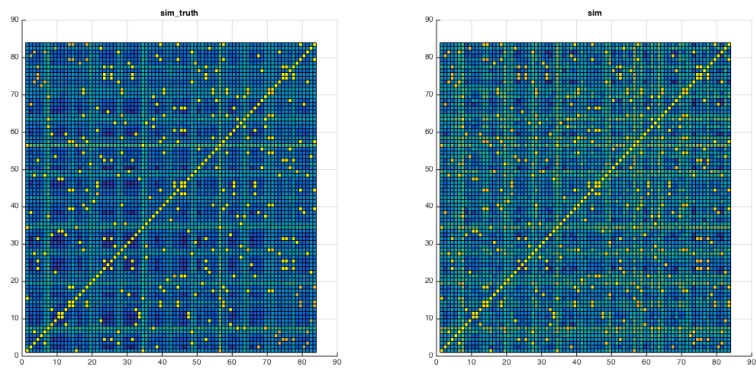
然后我们显示相似度最大的十对图像块：

```
match_range = 1:10;  
show_matches(imgs, matches(match_range, :), values(match_range));
```

0.989722	Block 13 	Block 14 
0.988194	Block 63 	Block 69 
0.987549	Block 61 	Block 69 
0.98676	Block 10 	Block 11 
0.986703	Block 61 	Block 63 
0.986079	Block 33 	Block 47 
0.984905	Block 69 	Block 84 
0.98478	Block 40 	Block 54 
0.984675	Block 75 	Block 77 
0.984028	Block 39 	Block 44 

可以看到，这十对图片确实是正确匹配的。

为了更形象化地看到我们匹配的结果，我们对 `sim` 和 `sim_truth` 进行绘图：

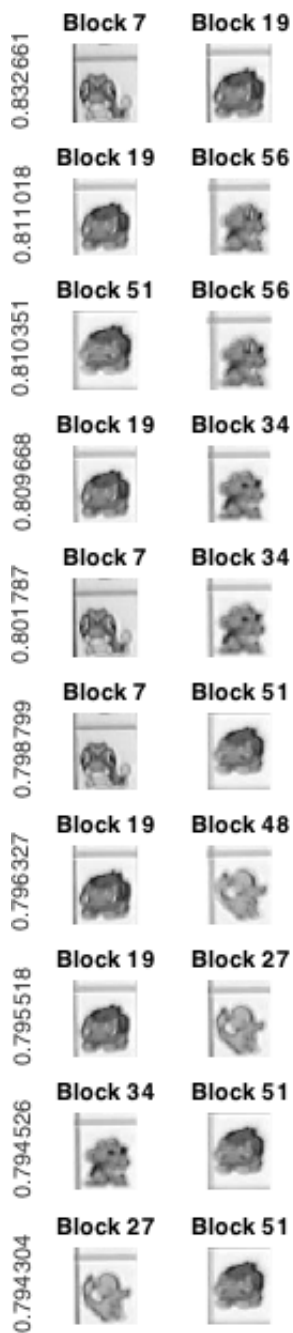


可以看到，很明显地，有一些元素的值明显高于周围元素的值。同时，虽然右图中背景杂音较大，但还是能够清晰地辨认出匹配对。这便是匹配函数设计合理的标志。

2.4 找到前十误匹配

通过人工查找，将相似度最大的前十误匹配显示出来：

```
match_range = [181 183:191];  
show_matches(imgs, matches(match_range, :), values(match_range));
```



可以看到，这十个误匹配对里都出现了妙蛙种子。仔细观察可以发现，图中的妙蛙种子由于模糊而丢失而很多细节。所以误匹配的产生可能是由于滤去了低频分量之后，妙蛙种子的纹理剩下得很少。

2.5 映射游戏区域

为了将图像块映射成索引值数组，我们设定一个阈值，并接受所有阈值以上的匹配对。然后，我们只需要找到这些匹配对中的各个等价类，便可以得到索引值数组。同时，从每个等价类中取出一块作为图例，便可得到分块对照表。代码如下：

```
%% map_game: Map imgs into a game ground.
```

```

function [ground, legends] = map_game(imgs, matches, values, accept_threshold)
    ground = zeros(size(imgs));

    kind_num = 1;

    % Accpet all close matches.
    last_close = find(values < accept_threshold) - 1;
    for k = 1:last_close
        match = matches(k, :);
        kinds = ground(match);
        if kinds == 0 % New kind.
            ground(match) = kind_num;
            legends{kind_num} = imgs{match(1)};
            kind_num = kind_num + 1;
        elseif any(kinds == 0) % One old kind, one not classified.
            ground(match) = max(kinds);
        elseif kinds(1) ~= kinds(2) % Old kind & close, combine.
            ground(ground == kinds(2)) = kinds(1);
        end % Else already the same.
    end

    % Normalize kind number.
    new_kind_num = 1;
    for kind = 1:kind_num-1
        poses = find(ground == kind);
        if poses
            ground(poses) = new_kind_num;
            legends{new_kind_num} = legends{kind};
            new_kind_num = new_kind_num + 1;
        end
    end

    legends = legends(1:new_kind_num-1);

```

这里我们选取阈值为 0.84:

```
[ground, legends] = map_game(imgs, matches, values, 0.84);
```

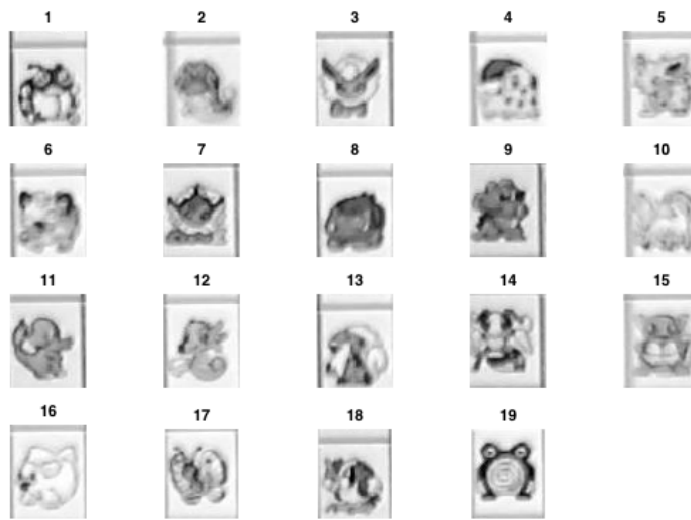
得到索引数组如下:

```

9  14  9  7  6  19  5  10  1  4  4  17
15  7  17 12 17  2  1  8  4 16 18  1
11  3  4 16 12 15  5 12 14  5 11 15
12  3  1 12 14  1  5  7  5 15 12 11
18 14  8  6  3  4  3  4  2 10 12  7
11  1 13 11  9 13 11 10  6  2 10  1
2   1  5  4  6 19 17  9  2  4 12  2

```

图例如下:



可以看到，我们成功正确识别了所有块。

2.6 模拟自动连连看

我们编写 `sim_play` 函数来实现模拟自动连连看。

由于匹配的速度较慢，我们将识别结果 `gound` 作为参数传入。同时，我们使用之前编写的 `omg` 函数来计算游戏的解。

我们在每次消除前都使用 `pause` 函数暂停 0.5s，并将将要消除的两块用灰色框框出，并在右侧放大显示，以便于我们判断算法是否正确。

具体代码实现如下：

```
%% sim_play: Simulate auto-playing.
function sim_play(img, ground)
    WIDTH = 5;

    [blocks, h_period, h_blank, v_period, v_blank] = divide_img(img);
    block_size = [v_period, h_period];
    blank = [v_blank, h_blank];

    steps = omg(ground);

    row = (1:size(img, 1))';
    col = 1:size(img, 2);

    for k = 2:4:length(steps)
        x(1) = steps(k);
        y(1) = steps(k + 1);
        x(2) = steps(k + 2);
        y(2) = steps(k + 3);

        % Indicate that two blocks.
        subplot(2, 5, 5)
        imshow(blocks{x(1), y(1)});
        subplot(2, 5, 10)
        imshow(blocks{x(2), y(2)});

        mask = logical(zeros(size(img)));

        for k = [1 2]
            ul = blank + [x(k) - 1, y(k) - 1] .* block_size + [1 1];
            lr = blank + [x(k), y(k)] .* block_size;

            row_range = (row >= ul(1) & row <= lr(1));
```

```

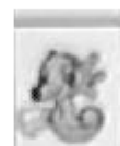
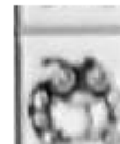
col_range = (col >= ul(2) & col <= lr(2));
row_bound = row_range & (row - ul(1) < WIDTH | ...
    lr(1) - row < WIDTH);
col_bound = col_range & (col - ul(2) < WIDTH | ...
    lr(2) - col < WIDTH);

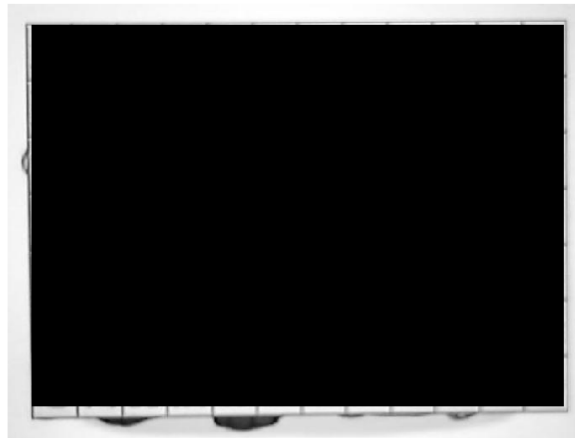
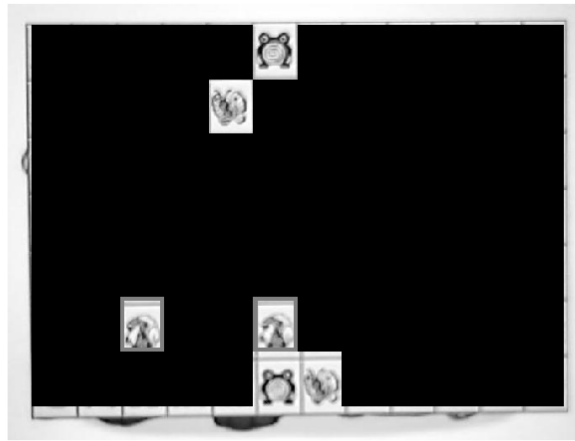
frame = bsxfun(@and, row_bound, col_range) | ...
    bsxfun(@and, col_bound, row_range);
mask = mask | bsxfun(@and, row_range, col_range);

img(frame) = 128;
end

subplot(2, 5, [1, 9])
imshow(img)
pause(0.5)
img(mask) = 0;
end
imshow(img) % Show Last frame.

```





可以看到，我们确实实现了模拟的自动连连看。