

MP Assignment Report

Alec Maughan · 2021

Contents

1 · Introduction.....	0
2 · Approach & Technique	1
2.1 · General Approach	1
2.1.1 · Preprocessing	1
2.1.2 · Segmentation.....	2
2.1.3 · Number Detection.....	3
2.1.4 · Classification	6
2.2 · Approach for Task One.....	7
2.2.1 · Number Detection.....	7
2.2.2 · Classification	7
2.3 · Approach for Task Two.....	8
2.3.1 · Number Detection.....	8
2.3.2 · Classification	9
3 · Performance	10
3.1 · Task One	10
3.2 · Task Two	11
4 · Source Code	14
4.1 · assignment.py	14
4.2 · number_finder.py	17

1 · Introduction

I state that this assignment has been fully completed as specified in the assignment specification.
There are no references to declare.

2 · Approach & Technique

2.1 · General Approach

This section outlines the pipeline that both task one and two follow. Specifics for task one and two can be found in their respective sections. For this section we will use the provided image `val/task1/Va101.jpg` as an example.

2.1.1 · Preprocessing

The detection of numbers is done using edge detection, contours, and contour bounding box properties.

The source image (Figure 1) is run through preprocessing to give a small Gaussian blur to the image to reduce the impact of noise.

The image is then converted to greyscale and run through an aggressive Canny edge detection algorithm to get the binary edges image that highlights the numbers (Figure 2).



Figure 1

The original image

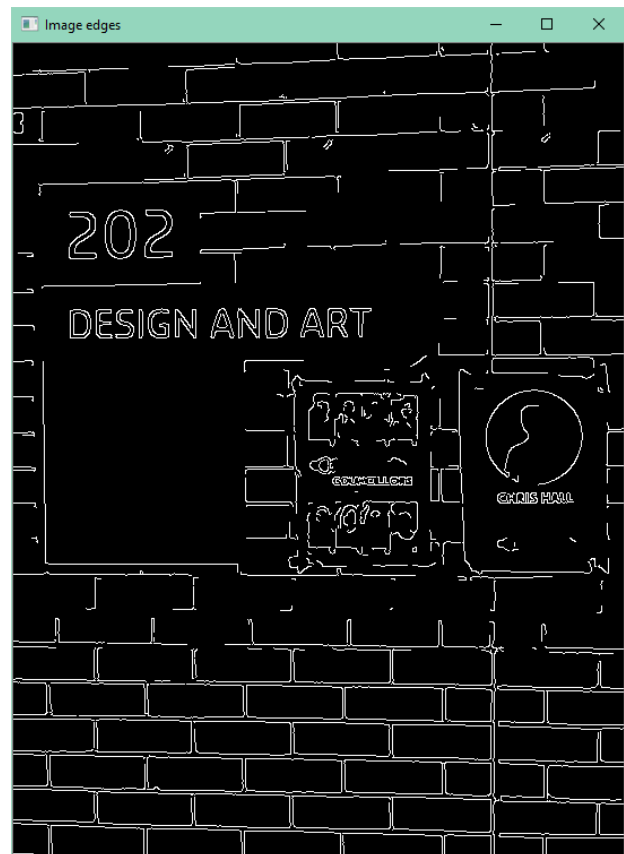


Figure 2

Image with Gaussian blur and Canny edge detection applied

2.1.2 · Segmentation

The detected edges are run through the number finding algorithm outlined in the next section, and the resulting bounding boxes that enclose the detected numbers are used to create a larger bounding box to encompass the entire detected area (Figure 3).

Then, the angle of skew is found between the first and the last number digit, to allow for rudimentary perspective correction.

The image is then cropped to the bounding box with this angle, with this forming the detected area output (Figure 4).



Figure 3

The fully enclosing bounding box for the number area

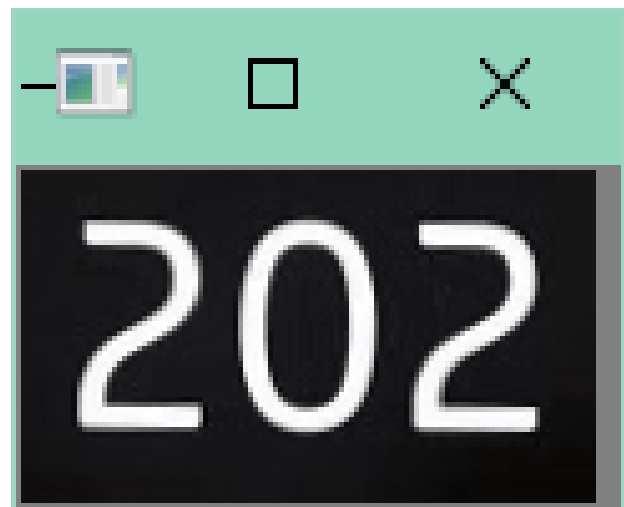


Figure 4

The cropped and rotated bounding box

2.1.3 · Number Detection

This covers the algorithm used to find the numbers. This is run before segmentation to find the segment, and then again on the segmented cropped image to get a more accurate reading, as the cropped image has slight perspective correction and omits the background.

Firstly, the OpenCV Find Contours function is used on the edges image to generate a list of contours (Figure 5). These contours are then filtered to only include those of significant size, and of an aspect ratio within the bounds of a typical digit. This leaves contours that are likely to be numbers (Figure 6).



Figure 5

All found contours



Figure 6

Contours filtered by size and aspect ratio

Then the (non-rotated) bounding box of these contours is taken (Figure 7) and these boxes are worked with from here on out.

For the next stage, every possible size-three permutation of these bounding boxes is found. One of these permutations will be the three building numbers from left to right, we just need to find it.

Then, these permutations are filtered to only include ones that are left to right by their x value (Figure 8).



Figure 7

Bounding boxes of filtered contours



Figure 8

Bounding boxes that are left-to-right on the x axis

Then, iteratively increasing a threshold value until we find a match, these permutations are filtered to ones that have very similar y values and very similar heights by comparing the difference of these values to the threshold value.

Once the threshold is increased to a value that allows at least one permutation through the filtering, these remaining permutations are taken to be building numbers (Figure 9).



Figure 9

Bounding boxes of filtered contours

2.1.4 · Classification

A template matching approach is used, with templates from the provided `digits_original` directory. These images are all 28 by 40 pixels in size, so some preprocessing must be done on the images being classified before template matching can be applied.

Starting with the bounding boxes that surround each building digit, each bounding box is padded slightly to better match the scale of the template images (Figure 10).

The bounding boxes are then run through an algorithm to pad either the x or y dimension such that the aspect ratio of the box roughly matches the aspect ratio of the template (Figure 11).

We can then use this box to crop the digit out of the original image, and resize it to match the size of the template, and because the aspect ratios match, there is minimal skewing.

With these cropped and properly sized digit images, we can use template matching. First a dictionary is built that maps numbers from zero to ten to lists of template images for those digits, and strings 'l' and 'r' to lists of template images for those arrow directions.

Then, for each digit image, each list of digit templates is matched over the image using the normalised correlation coefficient template matching algorithm, as it is normalised across template images, and gives the best results from testing.

The template that produces the maximum match value is selected as the match for that digit, which is found using OpenCV's Min Max Location algorithm.

The result of this classification is taken to be the true value of the building number to be output.



Figure 10

Slightly padded bounding boxes

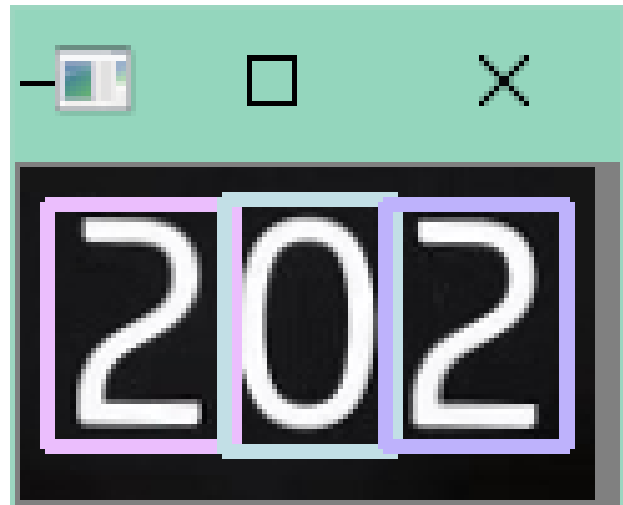


Figure 11

Bounding boxes padded for the correct aspect ratio

2.2 · Approach for Task One

This section covers the specifics and differences in the techniques used for detecting and recognising building signage. For this section we will continue to use the provided image val/task1/Val101.jpg as an example.

2.2.1 · Number Detection

When filtering permutations with iterative thresholding using the discussed number detection algorithm, the threshold value is initialised to six, as this produced best results.

After finding at least one building number using this algorithm (Figure 12), the permutation with the largest total area is taken to be the single building number in the image (Figure 13).

2.2.2 · Classification

The digit classification algorithm discussed is used to classify the single building number detected in the image.



Figure 12

Permutations after filtering



Figure 13

Permutation with largest area

2.3 · Approach for Task Two

This section covers the specifics and differences in the techniques used for detecting and recognising directional signage. For this section we will use the provided image `val/task2/val103.jpg` as an example.

2.3.1 · Number Detection

When filtering permutations with iterative thresholding using the discussed number detection algorithm, the threshold value is initialised to ten, as this produced best results. The filter also includes an additional check that the three digits are sufficiently close together on the x axis, which was needed due to the higher complexity of the directional signage images.

The permutations can be seen before this filtering (Figure 14) and after (Figure 15).

After finding at least one building number using the discussed algorithm, the permutations are sorted based on their y value, and this is taken to be the list of all building numbers in the image.

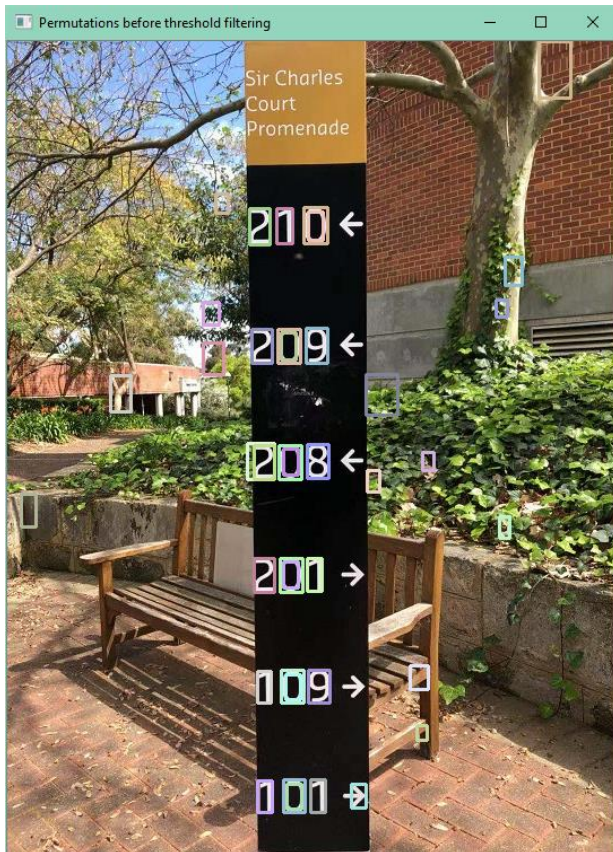


Figure 14

Permutations before threshold filtering

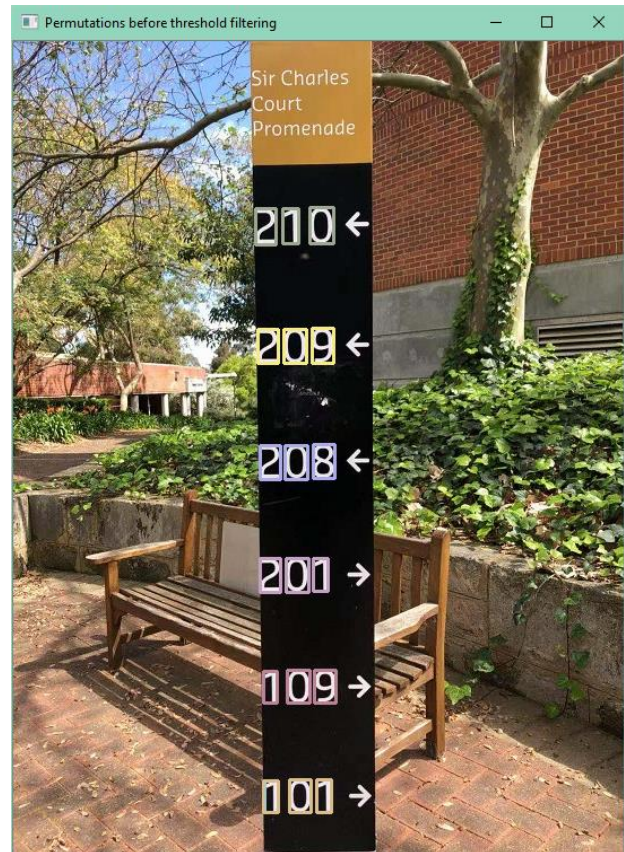


Figure 15

Permutations after threshold filtering

2.3.2 · Classification

As there are potentially multiple building numbers, the discussed digit classification algorithm is run on each building number.

Also, steps are taken to remove duplicate readings of building numbers from the output, as this was found to be a problem when dealing with multiple detected building numbers. This is done by ignoring duplicate (x, y) values for the first digit of a building number, and ignoring duplicates of classified building numbers. While the latter step would be sufficient in removing all duplicates, performance was found to be better when both approaches were used.

For arrow direction classification, firstly the bounding box around the arrow was found using the values of the bounding boxes of each of the digits (Figure 16).

This bounding box was then padded to the correct aspect ratio and resized just like the digits (Figure 17), and then run through the discussed classification algorithm, using the arrow templates instead of digits.

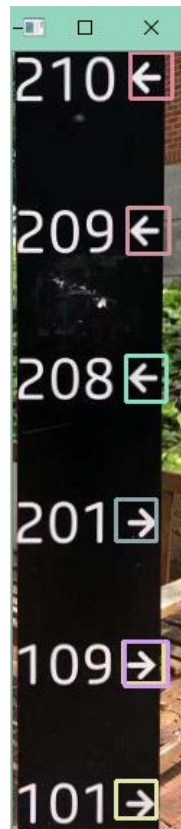


Figure 16

Computed arrow bounding boxes



Figure 17

Arrow bounding boxes with corrected aspect ratios


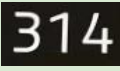








3 · Performance

This section covers the performance of the algorithms against the set of validation images.

3.1 · Task One

The entire process for task one is very accurate, correctly detecting the area of every building number, and correctly recognising every number.

This result is caused by the careful tweaking of the algorithm parameters to work well with these kinds of images, and the simplicity of the task, with only one building number present in the image.

Validation Image	Actual number	Recognised number	Detected area
Val01.jpg	202	202	
val02.jpg	314	314	
val03.jpg	301	301	
val04.jpg	109	109	
val05.jpg	206	206	
val06.jpg	312	312	
val07.jpg	209	209	
val08.jpg	207	207	
val09.jpg	215	215	
val10.jpg	204	204	


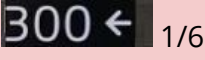

3.2 · Task Two

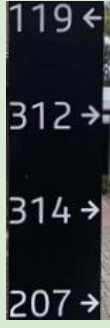
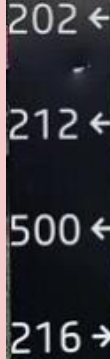
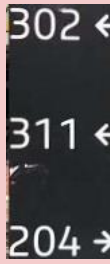
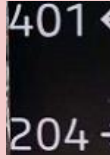
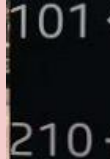
Task two is a slightly different story. Across these validation images, it detects an average of 53% of the building numbers present. Of it's detected signs, it correctly recognises 86% of the numbers, and 55% of the arrows.



Number recognition results are good due to the time put into tweaking the algorithm parameters from task one.

Area detection suffers due to lack of a good algorithm to single out groups of bounding boxes in this scenario. A better approach may have been to segment the sign area based on the colour of the sign.

Arrow direction recognition also suffers due to the rudimentary approach to obtaining the area in which the arrow sits, by going off the numbers rather than actually detecting the arrow, often the arrow is not well cropped before being classified. A better approach may have been to detect the arrow contour itself to get an exact area.

Validation Image	Actual signage	Recognised signage	Detected area
val01.jpg	208 ← 501 ← 205 → 202 → 206 →	684 ← 084 ←	
val02.jpg	303 ← 305 ← 300 ← 301 → 312 → 309 →	300 ←	
val03.jpg	210 ← 209 ← 208 ← 201 → 109 → 101 →	210 ← 209 ← 208 ← 201 → 109 ← 009 ← 101 →	

val04.jpg	119 ← 312 → 314 → 207 →	312 ← 314 ← 207 ←	 4/4
val05.jpg	202 ← 212 ← 500 ← 216 → 209 → 210 →	202 ← 212 ← 500 ← 216 ←	 4/6
val06.jpg	302 ← 311 ← 204 → 599 → 201 →	302 ← 311 → 204 ←	 3/5
val07.jpg	599 ← 105 ← 401 ← 204 → 215 → 205 →	401 ← 204 ←	 2/6
val08.jpg	201 ← 109 ← 101 ← 210 → 209 → 208 →	101 ← 210 ←	 2/6

val09.jpg	599 ← 105 ← 401 ← 204 → 215 → 205 →	593 ← 105 ← 401 ←	 599 ← 105 ← 401 ← 3/6
val10.jpg	201 ← 109 ← 101 ← 210 → 209 → 208 →	101 ← 210 ←	 101 ← 210 → 2/6

4 · Source Code

What follows is the Python 3 source code for this assignment.

4.1 · assignment.py

This file contains the main function, and has a number of command-line arguments required to configure which task it runs, and what directories it works from.

```
import argparse
import os

import cv2 as cv
import number_finder

def main():
    '''Runs the assignment based on given command-line args.'''

    # Resolve command line args
    args = _resolveArgs()

    # Use args to parse required data
    args = _resolveParams(args)

    if args.task == 'task1':
        number_finder.task1(
            test_imgs=args.test_imgs,
            output_dir=args.output_dir,
            templates_dict=args.templates_dict
        )
    else: # task2
        number_finder.task2(
            test_imgs=args.test_imgs,
            output_dir=args.output_dir,
            templates_dict=args.templates_dict
        )

def _resolveParams(args):
    '''Take in command-line args and add to them the output directory,
    the test images, and the template map.'''

    # OS file path separator (\ or /)
    sep = os.path.sep

    # Get the current project directory path (src/../../)
```

```

projDir = f'{os.path.dirname(os.path.abspath(__file__))}{sep}..{sep}'

# Directory of images to test
if args.task == 'task1':
    testDir = f'{args.test}{sep}task1{sep}'
else:
    testDir = f'{args.test}{sep}task2{sep}'

# Directory for output of program
if args.task == 'task1':
    args.outputDir = f'{args.output}{sep}task1{sep}'
else:
    args.outputDir = f'{args.output}{sep}task2{sep}'
# Ensure the output directory exists
os.makedirs(args.outputDir, exist_ok=True)

# Number of images under test directory
numTrain = args.num
# Images to test
if args.task == 'task1':
    filename = lambda imgNum: f'{testDir}BS{imgNum:02d}.jpg'
else:
    filename = lambda imgNum: f'{testDir}DS{imgNum:02d}.jpg'
args.testImgs = [cv.imread(filename(n)) for n in range(1, numTrain + 1)]

templatesDir = f'{projDir}digits{sep}'
templatesDict = {
    0: 'Zero',
    1: 'One',
    2: 'Two',
    3: 'Three',
    4: 'Four',
    5: 'Five',
    6: 'Six',
    7: 'Seven',
    8: 'Eight',
    9: 'Nine',
    'l': 'LeftArrow',
    'r': 'RightArrow'
}
# Maps from a key from templatesDict to a list template images
args.templatesDict = {
    key: [
        cv.imread(f'{templatesDir}{name}{i}.jpg') for i in range(1,6)
    ] for key, name in templatesDict.items()
}

```



```

    }

    return args

def _resolveArgs() -> argparse.Namespace:
    '''Constructs a command-line argument namespace with attributes task, test,
    output, and num.'''

    parser = argparse.ArgumentParser(
        description='MP Assignment by Alec Maughan',
        add_help=True,
        epilog='dependencies: python 3.7.3, opencv-python 3.4.2.16, matplotlib'
    )

    parser.add_argument(
        'task',
        help='task1 (Building numbers) or task2 (Directional numbers)',
        type=str,
        choices=('task1', 'task2'),
    )

    parser.add_argument(
        '--test', '-t',
        help='Directory of images to process',
        type=str
    )

    parser.add_argument(
        '--output', '-o',
        help='Directory of output',
        type=str
    )

    parser.add_argument(
        '--num', '-n',
        help='Number of images under the test directory',
        type=int
    )

    return parser.parse_args()

if __name__ == '__main__':
    main()

```

4.2 · number_finder.py

This contains all of the machine perception logic for task one and two.

```

from types import LambdaType
from typing import Any, Dict, Iterable, List, Tuple, Union
import cv2 as cv
import numpy as np
import random
from itertools import permutations
from statistics import median, mean

# Tasks -----

def task1(
    testImgs: List[Any],
    outputDir: str,
    templatesDict: Dict[Union[int, str], Any]
) -> None:
    '''The structure for task1 - take in a list of images, and find the
    area containing the building number, and classify the building number
    itself'''

    for n, img in enumerate(testImgs, start=1):

        # Crop the image to the area containing the building number
        cropped = _cropToNumbers(img)

        cv.imwrite(f'{outputDir}DetectedArea{n:02d}.jpg', cropped)

        # Find the numbers within the cropped area
        numberRects = _findNumbers(_findEdges(cropped), relSizeThresh=0.006)

        # Filter the templates to only include digits
        digits = {
            k: v for k, v in templatesDict.items() if k != 'l' and k != 'r'
        }

        # Classify the numbers
        actualNumbers = _classifyRects(cropped, numberRects, digits)

        with open(f'{outputDir}Building{n:02d}.txt', 'w') as file:
            a, b, c = actualNumbers
            file.write(f'Building {a}{b}{c}')

```

```

def task2(
    testImgs: List[Any],
    outputDir: str,
    templatesDict: Dict[Union[int, str], Any]
) -> None:
    '''The structure for task2 - take in images of multiple building numbers
    with arrows, find the area containing these signs, and the values and
    directions of the numbers and arrows respectively.'''

    for n, img in enumerate(testImgs, start=1):

        # Crop the image to the sign area
        cropped = _cropToNumbersDirectional(img)

        cv.imwrite(f'{outputDir}DetectedArea{n:02d}.jpg', cropped)

        # Find the groups of bounding boxes around the digits and arrows
        numberRectGroups = _findNumbersDirectional(_findEdges(cropped))

        # Split the digitsDict into separate dicts for digits and arrows
        digits = {
            k: v for k, v in templatesDict.items() if k != 'l' and k != 'r'
        }
        arrows = {
            k: v for k, v in templatesDict.items() if k == 'l' or k == 'r'
        }

        # Classify the numbers and directions
        actualNumbersGroups, directions = _classifyRectsDirectional(
            cropped, numberRectGroups, digits, arrows
        )

        with open(f'{outputDir}Building{n:02d}.txt', 'w') as file:
            for actualNumbers, direction in zip(
                actualNumbersGroups, directions
            ):
                a, b, c = actualNumbers
                file.write(f'Building {a}{b}{c} to the {direction}\n')

# Find number boxes -----

def _findNumbers(

```

```

edges: Any,
relSizeThresh=0.0006,
minRatio=0.4,
maxRatio=0.8
) -> Tuple[tuple, tuple, tuple]:
    '''The full algorithm to find the three building numbers from an image,
    as a tuple of bounding rectangles (Tuples of `x, y, width, height`).'''

    _, contours, _ = cv.findContours(
        edges, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE
    )

    # Reduce the size by filtering out small contours, and ones far from the
    # desired aspect ratio
    contours = [c for c in contours if _relativeSize(edges, c) > relSizeThresh]
    contours = [c for c in contours if minRatio < _aspectRatio(c) < maxRatio]

    # Map contours to their bounding rectangles
    rects = [cv.boundingRect(c) for c in contours]

    # Get all permutations of size 3 of the bounding boxes
    perms = permutations(rects, 3)
    # Filter to only get left to right perms
    perms = [p for p in perms if p[0][0] < p[1][0] < p[2][0] and
        (p[0][0] + p[0][2]) < (p[1][0] + p[1][2]) < (p[2][0] + p[2][2])
    ]

    # Filter to only get perms of similar heights and y values to each other
    # Loop until the filtered list is non-empty
    filteredPerms = []
    thresh = 6
    while (len(filteredPerms) == 0):
        filteredPerms = [p for p in perms if
            _avgDiff(p, lambda a, b: abs(a[1] - b[1])) < thresh and
            _avgDiff(
                p, lambda a, b: abs((a[1] + a[3]) - (b[1] + b[3]))
            ) < thresh
        ]
        thresh += 1

    # After all of this filtering, we can assume the largest remaining
    # permutation is that of the building number
    perm = max(

```



```

        filteredPerms,
        key=lambda p: (
            (p[0][2] * p[0][3]) + (p[1][2] * p[1][3]) + (p[2][2] * p[2][3])
        )
    )

    return perm

def _findNumbersDirectional(
    edges: Any,
    relSizeThresh=0.0003,
    minRatio=0.4,
    maxRatio=0.8
) -> List[Tuple[Tuple[int, int, int, int]]]:
    '''The full algorithm to take in an edges image and find the bounding
    boxes around each of the separate building numbers, for use on a
    directional sign.'''

    _, contours, _ = cv.findContours(
        edges, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE
    )

    # Reduce the size by filtering out small contours, and ones far from the
    # desired aspect ratio
    contours = [c for c in contours if _relativeSize(edges, c) > relSizeThresh]
    contours = [c for c in contours if minRatio < _aspectRatio(c) < maxRatio]

    # Map contours to their bounding boxes
    rects = [cv.boundingRect(c) for c in contours]

    # Get all permutations of size 3 of the bounding boxes
    perms = permutations(rects, 3)

    # Filter to only get left to right perms
    perms = [p for p in perms if p[0][0] < p[1][0] < p[2][0] and
        (p[0][0] + p[0][2]) < (p[1][0] + p[1][2]) < (p[2][0] + p[2][2])
    ]

    # Filter to only get perms of similar heights and y values to each other,
    # and close to each other on x.
    # Loop until the filtered list is non-empty, increasing the threshold.
    filteredPerms = []
    thresh = 0

```

```

while len(filteredPerms) == 0:
    thresh += 1

    filteredPerms = [p for p in perms if
        _avgDiff(p, lambda a, b: abs(a[1] - b[1])) < thresh and
        _avgDiff(
            p, lambda a, b: abs((a[1] + a[3]) - (b[1] + b[3]))
        ) < thresh and
        p[0][0] + p[0][2] - p[1][0] < thresh and
        p[1][0] + p[1][2] - p[2][0] < thresh
    ]

# Sort by y value
filteredPerms = sorted(filteredPerms, key=lambda p: p[0][1])

return filteredPerms

# Crop to number area -----

def _cropToNumbers(img: Any) -> Any:
    '''The full algorithm to take in an image and crop it to just the
    area of the numbers. To actually find the numbers it uses `findNumbers`.'''

    edges = _findEdges(img)

    # Find the three bounding boxes surrounding the three numbers, in left
    # to right order
    numberRects = _findNumbers(edges)

    # Find the angle that these numbers are skewed at
    angle = _numberAngle(numberRects)

    # Pad the rectangle a bit
    pad = round((sum([p[3] for p in numberRects]) / 3) * 0.25)
    boundingRect = (
        numberRects[0][0] - pad,
        numberRects[0][1] - pad,
        (numberRects[2][0] + numberRects[2][2]) - numberRects[0][0] + pad + pad,
        max(numberRects, key=lambda p: p[3])[3] + pad + pad
    )

    # Rotate the rectangle by the computed angle
    bx, by, bw, bh = boundingRect

```

```

rotBounding = (bx+(bw/2), by+(bh/2)), (bw, bh), angle

# Crop the original image to this rotated rectangle
cropped = _cropImg(img, rotBounding)

return cropped

def _cropToNumbersDirectional(img: Any) -> Any:
    '''Full algorithm to take in an image of multiple directional building
    numbers and give the cropped area containing just the numbers and arrows.'''

    # Apply edge detection
    edges = _findEdges(img)

    # Find the groups of bounding boxes around the numbers
    numberRectGroups = _findNumbersDirectional(edges)

    # Create a bounding box around the entire group
    pad = round((sum(
        sum(pp[3] for pp in p) / 3 for p in numberRectGroups
    ) / len(numberRectGroups)) * 0.1)
    firstRects = numberRectGroups[0]
    lastRects = numberRectGroups[len(numberRectGroups) - 1]
    boundingRect = (
        round(median(p[0][0] for p in numberRectGroups)) - pad,
        round(mean(pp[1] for pp in firstRects) - pad),
        round((
            lastRects[2][0] + lastRects[2][2]
        ) - firstRects[0][0] + pad * 2 + firstRects[0][2] * 2.5),
        lastRects[2][1] + lastRects[2][3] - firstRects[0][1] + pad * 2
    )

    # Find the skew that the numbers are at
    angle = mean(_numberAngle(p) for p in numberRectGroups)

    # Define a rotated bounding box using this angle
    bx, by, bw, bh = boundingRect
    rotBounding = (bx+(bw/2), by+(bh/2)), (bw, bh), angle

    # Crop the image to this rotated bounding box
    cropped = _cropImg(img.copy(), rotBounding)

    return cropped

```

```

# Classification -----

def _classifyRects(
    cropped: Any,
    numberRects: List[Tuple[int, int, int, int]],
    digits: Dict[int, Any]
) -> Tuple[int]:
    '''The full algorithm to take in an area of a building number, the bounding
    boxes of the numbers, and a digit classificaiton map, and return the
    true digits.'''

    # Pad the number rectangles a bit
    numberRectsPadded = [
        _padRect(rect, (rect[3] * 0.08), (rect[3] * 0.08))
    ] for rect in numberRects

    # Defined by size of images in digits directory
    minW, minH = 28, 40
    templateRatio = minW / minH

    # Pad for aspect ratio
    for i, numberRect in enumerate(numberRectsPadded):
        _, _, w, h = numberRect
        ratio = w / h

        if ratio != templateRatio:
            if ratio < templateRatio:
                padX = (h * templateRatio) - w
                numberRectsPadded[i] = _padRect(numberRect, padX, 0)
            else:
                padY = (w / templateRatio) - h
                numberRectsPadded[i] = _padRect(numberRect, 0, padY)

    # Use the rects to crop out the number images
    numberImgs = [
        _cropImg(
            cropped, _rectToRotRect(numImg)
        ) for numImg in numberRectsPadded
    ]

    # Scale to fit the matcher digits images
    resizedNumberImgs = [

```



```

        cv.resize(numImg, (minW, minH)) for numImg in numberImgs
    ]

    # Run through the classifier
    actualNumbers = [
        _matchNum(numberImg, digits) for numberImg in resizedNumberImgs
    ]

    return tuple(actualNumbers)

def _classifyRectsDirectional(
    cropped: Any,
    rectGroups: List[List[tuple]],
    digits: Dict[int, Any],
    arrows: Dict[str, Any]
) -> Tuple[List[List[int]], List[str]]:
    '''The full algorithm to take in an area containing multiple building
    numbers with arrows, the bounding boxes of the numbers, and a digit and
    arrow classificaiton map, and return the true digits and arrow directions.
    '''

    # Pad the rects
    numberRectGroupsPadded = [[
        _padRect(pp, pp[3] * 0.08, pp[3] * 0.08) for pp in p
    ] for p in rectGroups]

    # Takes in three number rects and gets the rect of the adjacent arrow
    def arrowBox(p: tuple) -> tuple:
        a, _, c = p
        floatRect = (
            c[0] + c[2] + (mean(pp[2] for pp in p)) * 0.3,
            mean(pp[1] for pp in p),
            (c[0] - a[0]) * 0.6,
            mean(pp[2] for pp in p) * 1.4
        )
        return tuple(round(pp) for pp in floatRect)

    # Add the arrow box to the groups
    numberRectGroupsPadded = [
        (p[0], p[1], p[2], arrowBox(p)) for p in numberRectGroupsPadded
    ]

    # Defined by size of images in digits directory

```

```

minW, minH = 28, 40
templateRatio = minW / minH

# Initialise lists for output
actualNumbersGroup = []
directionsGroup = []

# Allows us to ignore duplicates
visited = []

# Iterate through groups
for numberRectsPadded in numberRectGroupsPadded:
    # Initialise inner padded rects list
    newRects = []

    # Only proceed if not a duplicate (x, y) coord
    if numberRectsPadded[0][:2] not in visited:
        visited.append(numberRectsPadded[0][:2])

    # For each number/arrow rect, pad to match the template aspect ratio
    for numberRect in numberRectsPadded:
        _, _, w, h = numberRect
        ratio = w / h

        if ratio != templateRatio:
            if ratio < templateRatio:
                padX = (h * templateRatio) - w
                newRects.append(_padRect(numberRect, padX, 0))
            else:
                padY = (w / templateRatio) - h
                newRects.append(_padRect(numberRect, 0, padY))
        else:
            newRects.append(numberRect)

    # Use the rects to crop out the number/arrow images
    numberImgs = [
        _cropImg(cropped, _rectToRotRect(rect)) for rect in newRects
    ]

    # Scale to match the template image size
    resizedNumberImgs = [
        cv.resize(numImg, (minW, minH)) for numImg in numberImgs
    ]

```

```

# Classify numbers
actualNumbers = [
    _matchNum(numImg, digits) for numImg in resizedNumberImgs[:-1]
]
actualNumbers = actualNumbers if actualNumbers[1] != 6 else (
    actualNumbers[0], 0, actualNumbers[2]
)

# Classify arrow direction
direction = _matchNum(resizedNumberImgs[-1], arrows)
directionStr = 'left' if direction == '1' else 'right'

# Add to group output list if not a duplicate
if actualNumbers not in actualNumbersGroup:
    actualNumbersGroup.append(actualNumbers)
    directionsGroup.append(directionStr)

return actualNumbersGroup, directionsGroup

def _matchNum(
    img: Any,
    digits: Dict[Union[int, str], Any]
) -> Union[int, str]:
    '''Takes in a source image and map from desired values to lists of image
    templates, and uses it to classify the source image as one of the digits.'''

    # matchTemplate technique
    method = cv.TM_CCOEFF_NORMED

    # Initialise map of max values from matchTemplate
    maxima = {value: 0 for value in digits.keys()}

    img = cv.cvtColor(img.copy(), cv.COLOR_BGR2GRAY)

    # Iterate through digits and match the template, populating maxima
    # with the result
    for key, templates in digits.items():
        maxMatch = 0

        for template in templates:
            template = cv.cvtColor(template, cv.COLOR_BGR2GRAY)

```

```

        matchImg = img.copy()
        matchRes = cv.matchTemplate(matchImg, template, method)
        _, resValue, _, _ = cv.minMaxLoc(matchRes)

        maxMatch = max(maxMatch, resValue)

    maxima[key] = maxMatch

# Find the best matching template
maxKey, _ = max(maxima.items(), key=lambda item: item[1])

return maxKey

# Utilities -----

def _aspectRatio(contour: Any) -> float:
    '''Computes the aspect ratio of a contour's bounding rectangle
    (width / height).'''
    _, _, width, height = cv.boundingRect(contour)

    return width / height if height != 0 else 0

def _findEdges(img: Any, gaussian=3, t1=300, t2=400) -> Any:
    '''Applies the Canny edge detector to an image with preset values and
    preprocessing.'''
    blurred = cv.GaussianBlur(img, (gaussian, gaussian), 0)
    greyBlurred = cv.cvtColor(blurred, cv.COLOR_BGR2GRAY)
    edges = cv.Canny(greyBlurred, t1, t2)

    return edges

def _relativeSize(img: Any, contour: Any) -> float:
    '''Computes the relative size of a contour as a proportion of the source
    image size.'''
    imgWidth, imgHeight = img.shape[:2]
    _, _, cntWidth, cntHeight = cv.boundingRect(contour)

    imgArea = imgWidth * imgHeight
    contourArea = cntWidth * cntHeight

    return contourArea / imgArea if imgArea != 0 else 0

def _avgDiff(iterable: Iterable, key: LambdaType) -> float:

```

```

'''Finds the average differences between elements in an iterable, given
a mapping function'''
total = sum([key(x[0], x[1]) for x in permutations(iterable, 2)])
numElements = len(list(permutations(iterable, 2)))

return total / numElements

def _cropImg(img: Any, rotRect: Tuple[tuple, tuple, float]) -> Any:
    '''Crops a given image mat to a rotated rectangle, using it's angle.
    `rotRect` can be sourced from cv.minAreaRec and is a tuple of format
    `(x, y), (width, height), angle`.'''
    _, (width, height), _ = rotRect
    rotBoundingCont = np.int0(cv.boxPoints(rotRect))
    points = rotBoundingCont.astype('float32')
    dest = np.array(
        [
            [0, height - 1],
            [0, 0],
            [width - 1, 0],
            [width - 1, height - 1],
        ],
        dtype='float32'
    )

    perspectiveMat = cv.getPerspectiveTransform(points, dest)
    cropped = cv.warpPerspective(img, perspectiveMat, (width, height))

    return cropped

def _numberAngle(numbers: Tuple[tuple, tuple, tuple]) -> float:
    '''Takes in a tuple of three bounding rects and finds the angle they slant
    at.'''
    angleOpp = ((numbers[2][0] + numbers[2][2]) - numbers[0][0])
    angleAdj = (
        (numbers[2][1] + numbers[2][3]) - (numbers[0][1] + numbers[0][3])
    )

    return np.arctan(angleOpp / angleAdj) if angleAdj != 0 else 0

def _padRect(
    rect: Tuple[int, int, int, int],
    padX: Union[int, float],
    padY: Union[int, float]

```

```

) -> Tuple[int, int, int, int]:
    '''Takes in a bounding rectangle and pads it on x and y.'''
    padX = round(padX)
    padY = round(padY)
    boundingRect = (
        rect[0] - padX,
        rect[1] - padY,
        rect[2] + padX * 2,
        rect[3] + padY * 2
    )
    return boundingRect

def _rectToRotRect(
    rect: Tuple[int, int, int, int]
) -> Tuple[Tuple[float, float], Tuple[float, float], float]:
    '''Converts a rect to a rotated rect'''
    x, y, w, h = rect
    return (x+(w/2), y+(h/2)), (w, h), 0

```