Algorithm Experiment
Seam Carving

李胜锐 2017012066

2020 年 4 月 2 日

Date Performed: 2020 年 4 月 2 日

1 Experiment Environment

Operating System: macOS 10.15.2
Processor: 2.6 GHz 6-Core Intel Core i7
Memory: 16GB
Language: python 3.7
IDE: PyCharm

2 Algorithm Analysis

2.1 Calculate Pixel Damage to Get Image Damage

```
def get_pixel_damage(img_data, damage_array, row_col):
    row = row_col[0]
    col = row_col[1]
    damage = 0
    for row_bias in [-1, 0, 1]:
        for col_bias in [-1, 0, 1]:
            if row + row_bias in range(0, img_data.shape[0]) \
                    and col + col_bias in range(0, img_data.shape[1]):
                damage_r = abs(
                    int(img_data[row][col][0]) - int(img_data[row + row_bias
                                                              ][col + col_bias][
                                                              0]))
                damage_g = abs(
                    int(img_data[row][col][1]) - int(img_data[row + row_bias
                                                              ][col + col_bias][
                                                              1]))
                damage_b = abs(
                    int(img_data[row][col][2]) - int(img_data[row + row_bias
                                                              ][col + col_bias][
                                                              2]))
                damage += (damage_r + damage_g + damage_b)/(abs(row_bias)+
                                                            abs(row_bias)+2)
    damage_array[row][col] = damage

def get_img_damage(img_data):
    damage_array = np.empty([img_data.shape[0], img_data.shape[1]])

    print('calculate damage...')
    for row in tqdm(range(damage_array.shape[0])):
        for col in range(damage_array.shape[1]):
            get_pixel_damage(img_data, damage_array, (row, col))
```

```
    return damage_array
```

Only pixels around the calculated pixel are concerned. First, we get the value of every damage $D(i,j) = |r(i) - r(j)| + |g(i) - g(j)| + |b(i) - b(j)|$ between the pixel and its surroundings. Then, each of the value are multiplied with a weight according to the distance, shown as followed:

图 1: Weight of damage.

All the damage of pixels in the image are calculated this way.
The time complexity is:

$$T(n,m) = \Theta(nm)$$

2.2 Calculate the Least Damage using DP

The following are only row-calculated codes. The column-calculated codes are not listed here.

```
def get_sum_damage_row(damage_array):
    sum_array = np.empty([damage_array.shape[0], damage_array.shape[1]])

    for col in range(sum_array.shape[1]):
        sum_array[0][col] = damage_array[0][col]
```

```
    print('calculate sum of damage by row...')
    for row in tqdm(range(1, sum_array.shape[0])):
        for col in range(sum_array.shape[1]):
            sum_array[row][col] = damage_array[row][col]
            if col == 0:
                sum_array[row][col] += min(sum_array[row - 1][col],
                                           sum_array[row - 1][col
                                            + 1])
            elif col == sum_array.shape[1] - 1:
                sum_array[row][col] += min(sum_array[row - 1][col - 1],
                                           sum_array[row - 1][col
                                           ])
            else:
                sum_array[row][col] += min(sum_array[row - 1][col - 1],
                                           sum_array[row - 1][col
                                           ],
                                       sum_array[row - 1][col + 1])

    return sum_array
```

We leverage the idea of dynamic programming. For each pixel p, the sum of damage must be:

$$S(p) = min_{q \in pre(p)}(S(q)) + D(p)$$

where $S(P)$ is the sum of the least damage and $D(p)$ is the pixel damage calculated before. $pre(p)$ is the set of all precursor pixels, ususally having 3 elements and sometimes only 2.
The time complexity is also:

$$T(n, m) = \Theta(nm)$$

2.3 Construct Cross link

```
def get_acrosslink(damage_sum_row, damage_sum_col, img_data):
    print('construct across link...')
    head = None
    links = []
    for row in tqdm(range(img_data.shape[0])):
        head = Pixel(img_data[row][0][2], img_data[row][0][1], img_data[row]
                                              [0][0], row, 0,
                     damage_sum_row[row][0], damage_sum_col[row][0])
```

```
        cur = head
        for col in range(1, img_data.shape[1]):
            cur.right = Pixel(img_data[row][col][2], img_data[row][col][1],
                                                    img_data[row][col][0], row
                                                    , col,
                              damage_sum_row[row][col], damage_sum_col[row][
                                                                    col])
            cur.right.left = cur
            cur = cur.right
        links.append(head)

    head = Pixel(-1, -1, -1, -1, -1, math.inf, math.inf)
    head.right = links[0]
    links[0].left = head
    while links[0]:
        for row, node in enumerate(links):
            if row + 1 < len(links):
                node.down = links[row + 1]
                node.down.up = node
            links[row] = node.right

    return head
```

To conduct seam carving for many times, we have to constrcut the Curb Table(十字链表) of our image.

The time complexity is also:

$$T(n, m) = \Theta(nm)$$

2.4 Seam Carving

```
def seam_carve(head):
    print('seam carving...')
    row_carve = int(arr_d.shape[0] / 2)
    col_carve = int(arr_d.shape[1] / 2)
    share = min(row_carve, col_carve)
    for _ in tqdm(range(share)):
        seam_carving_by_row(head)
        seam_carving_by_col(head)
    for _ in range(share, row_carve):
        seam_carving_by_row(head)
    for _ in range(share, col_carve):
        seam_carving_by_col(head)

    return head
```

```
def seam_carving_by_row(head):
    def find_up(node):
        if not node.up:
            return None
        up = node.up
        if not up.left:
            if up.row_sum < up.right.row_sum:
                return up
            else:
                return up.right
        elif not up.right:
            if up.row_sum < up.left.row_sum:
                return up
            else:
                return up.left
        else:
            if up.left.row_sum < min(up.row_sum, up.right.row_sum):
                return up.left
            elif up.right.row_sum < min(up.row_sum, up.left.row_sum):
                return up.right
            else:
                return up

    last_row = head.right
    while last_row.down:
        last_row = last_row.down

    to_be_delete = last_row

    min_sum = math.inf
    while last_row:
        if last_row.row_sum < min_sum:
            to_be_delete = last_row
            min_sum = last_row.row_sum
        last_row = last_row.right

    while to_be_delete:
        up_delete = find_up(to_be_delete)
        if to_be_delete.left:
            to_be_delete.left.right = to_be_delete.right
        if to_be_delete.right:
            to_be_delete.right.left = to_be_delete.left
        if up_delete:
            to_be_delete.up.down = up_delete.down
            up_delete.down.up = to_be_delete.up
```

```
        to_be_delete = up_delete
```

The process of seam curving is very like a greedy algorithm.
We first find the pixel i with the least $S(i)$ in the last row(column). Then its precursor j in the previous row(column) with the least $S(j)$ is found. Then we delete pixel i, let $i \leftarrow j$ and do the same operation recursively.
In our algorithm, we delete a row, a column and again a row like this way:

$$delete\ row \rightarrow delete\ column \rightarrow delete\ row \rightarrow \cdots$$

For each iteration, the time cost is $\Theta(m)$ or $\Theta(n)$.
If we cut the image of $m \times n$ into $m/2 \times n/2$

$$T(n, m) = \Theta(nm)$$

As a conlusion, the over-all time complexity is $T(n, m) = \Theta(nm)$

3 Operation Demo

Use command:

python seamcarving.py [IMAGENAME]

During processing:

图 2: During processing.

Examples: