Algorithms Coding Assignment 2 Solution

Ashutosh Dayal (22EC30066)

November 2024

Required packages: *numpy*, *matplotlib*

1 Matching Criteria

The given problem statement states a MSE error matching criteria for varying window sizes along edges, corners and central portion. This section proposes an alternate matching criteria which is shown to be equivalent, but computationally less expensive.

Proposition 1. *Pixels along the edges, and the corners can be ignored for matching purpose.*

Proof. This can be easily deduced by matching criteria that the object must be at least one pixel thick for detection, since the MSE matches only for the pixels for which surrounding pixels are also same in both images (may be at an offset). So if any object touches the edges, the edge is going to ignored anyways during MSE evaluation, since MSE of window of two different configuration (edge and central portion) remains unmatched. □

Proposition 2. *To compare two pixels, rather than using $MSE = 0$, we can directly verify if each pixel and its surrounding neighbors match in color and position within a specified window.*

Proof. MSE is of the form $\frac{1}{n}\sum_{i=0}^{n-1}(a_i - b_i)^2$. For this to be zero $a_i = b_i \ \forall \ i$. So we can directly check in a square window of size 9×9, for equality of each pixel. □

2 Solution 1 - Brute Force

2.1 Code

```
Dx=[0, 0, 1, -1, 0, 1, -1, -1, 1]
Dy=[0, 1, 0, 0, -1, 1, -1, 1, -1]

def is_match(mat1,mat2,i,j,a,b): #O(1)
    for k in range(9):
```

```
6              ni,nj,na,nb=i+Dx[k],j+Dy[k],a+Dx[k],b+Dy[k]
7              if 0<=ni<n and 0<=nj<m and 0<=na<n and 0<=nb<m:
8                  if mat1[ni,nj]!=mat2[na,nb]: return 0
9        return 1
10
11   def get_mask_1(mat1, mat2): #O(nm^2)
12      res=np.zeros_like(mat1,dtype='int64')
13      for i in range(0,n-1):
14        for j in range(m-1,-1,-1):
15          for k in range(j,m):
16            if is_match(mat1,mat2,i,k,i,j):
17              res[i,k]=k-j
18      return res
```

2.2 Algorithm description

Let n and m be the number of rows and columns respectively in $mat1/mat2$ (assuming similar sizes).

1. *Line 1:* **`is_match` Function definition:**
   - Input: $mat1, mat2 \in \mathbb{Z}^{n\times m}[0,255]$; $i,j,a,b \in \{0,1,\ldots n-1\}$. $mat1$ and $mat2$ represents the image description as in the problem statement. (i,j) and (a,b) are represents two locations on $mat1$ and $mat2$ respectively.
   - Returns: The alternate criteria mentioned in *Section 1*, is used for comparing two pixels. Returns 1 if matched, otherwise 0.
2. *Line 2-6:* Iterates over all neighbours and the pixel, check their validity (whether they are inside image of not?). If valid and any neighbour do not match, return 0, other 1.
3. *Line 8*: **`get_mask_1` Function definition:** Same input description as of `is_match` for $mat1$ and $mat2$. Assuming $mat2$ is left shifted version of $mat1$.
4. *Line 10-14*: Iterate over each pixel (i,j) in $mat2$ and check whether $\exists\ k \geq j$ such that, $mat1_{i,k} \equiv mat2_{i,j}$. If matched, color res according to offset $k-j$.

2.3 Complexity

- *Time complexity:* `is_match` function is $O(1)$, since it runs for constant number of iterations (exactly 9). For `get_mask_1`, it is clear there are 3 nested loops, so the overall runtime is $O(nm^2)$.
- *Memory complexity:* $O(nm)$

3 Solution 2 - Optimal Solution

3.1 Binary exponentiation

Binary exponentiation is an efficient algorithm to calculate a^b for given a and b using $O(logb)$ multiplications.

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ a \cdot a^{b-1} & \text{if } b \text{ is odd} \\ (a \cdot a)^{\frac{b}{2}} & \text{if } b \text{ is even} \end{cases} \tag{1}$$

3.2 String hashing

The good and widely used way to define the hash of a string s of length n is:

$$hash(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + ... + s[n-1] \cdot p^{n-1} \mod m$$

$$= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,$$

where p and m are some chosen, positive numbers. It is called a polynomial rolling hash function.

It can be proved that for two string with hash value, are equal, for a relatively large modulo domain. The same hashing concept is used for comparison of two pixels for the images. We apply a forward polynomial hash of each pixel with a window size of 3×3 for both images. Now to compare whether two pixels are equal (according to the alternate criteria mentioned in *Section 1*, we can verify the equality of hash values.

One major advance of hashing is that, we can iterate in backward direction in $mat2$ (for each row), and keep a track of values appearing in $mat1$ in a Hash Map (Python dictionary). This reduced the time complexity by order of m.

Please refer `https://cp-algorithms.com/string/string-hashing.html` for further details regarding this method.

3.3 Code

```python
M=1000000007 #Modulo domain used for program
HH_=100000073 #Polynomial hashing constant

def binpow(a, b):
  a %= M
  res = 1
  while b>0:
    if b&1: res = res * a % M
    a = a * a % M
    b >>= 1
  return res

pr=[binpow(HH_, i) for i in range(9)] #Used as prime coefficient
```

```
14
15 def hash_mat(mat): #O(nm)
16   res=np.zeros_like(mat,dtype='int64')
17   for i in range(1,n-1):
18     for j in range(1,m-1):
19       for k in range(9):
20         res[i,j]=(res[i,j]+mat[i+Dx[k],j+Dy[k]]*pr[k]%M)%M
21   return res
22
23 def get_mask_2(mat1, mat2): #O(nm)
24   m1,m2=hash_mat(mat1),hash_mat(mat2)
25   res=np.zeros_like(m1,dtype='int64')
26   for i in range(1,n-1): #This runs n-1 times
27     dd={}
28     for j in range(m-2,0,-1): #This runs m-1 times
29       if m2[i,j] in dd: res[i,j]=dd[m2[i,j]]-j # O(1) operation
30       dd[m1[i,j]]=j # O(1)
31   return res
```

3.4 Algorithm description

1. *Line 4-11*: `binpow(a,b)` returns the a^b in modulo domain M in $O(logb)$. For description of this particular implementation, please visit: `https://cp-algorithms.com/algebra/binary-exp.html`.

2. *Line 13*: Power of a prime number $HH\_$ is calculated and stored in an array *pr* for fast access for hashing the window.

3. *Line 15-21*: This function returns a hash equivalent of the matrix. For each each (i, j) in *mat*, it calculates the forward hash (same as string hashing) starting from upper left pixel and ending at bottom right pixel in a 3×3 window centered at (i, j), and stores in *res* at the same location.

4. *Line 23-24*: **`get_mask_2`** function has the same description as `get_mask_1`. $m1$ and $m2$ stores the equivalent hash matrices of $mat1$ and $mat2$.

5. *Line 26-30*: For each row i, create an empty hashmap *dd*. For each pixel at location j (iterating from back) in $m2$, check there exists a key in *dd* corresponding to pixel value, it it exists, *then there also exists a pixel in* $m1$*, which matches for some* $\exists\ k \geq j$, so update the *res* matrix. After updating *res*, update the *dd* map by adding a $(m1_{i,j}, j)$ key-value pair in the map.

3.5 Complexity

- *Time complexity*: Since Python dictionary is has a hashmap implementation, insertion of a key-value pair, and retrieval of a value is a $O(1)$ operation. `hash_mat` function is $O(mn)$, due to two nested loops. By similar logic, `get_mask_2` is $O(mn)$ in the worst case.

- *Memory complexity*: $O(mn)$

Since we need to load/generate the images pixel-by-pixel, at least $O(mn)$ is required. Also, the algorithm stated in this section has $O(mn)$ complexity, so this is the most optimal runtime possible for the problem.

Moreover, for storage, $O(mn)$ is the optimal for both the stated algorithms, since images need to be stored pixel wise.

"The weak cannot forgive; the strong can, forgiveness is an attribute of the strong."
$\sim$ *Shri Krishna*