Algorithmique et Programmation Avancée

Anh-Kiet DUONG

December 21, 2022

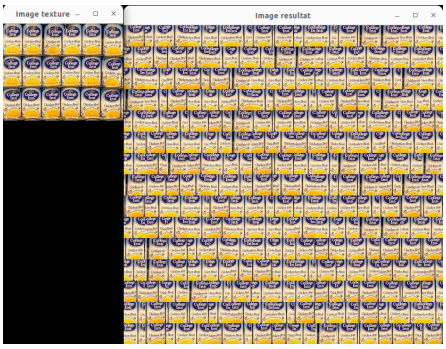
1 Introduce

My name: Anh-Kiet DUONG.
Program: Master (M1) CRYPTIS.
Source code: github

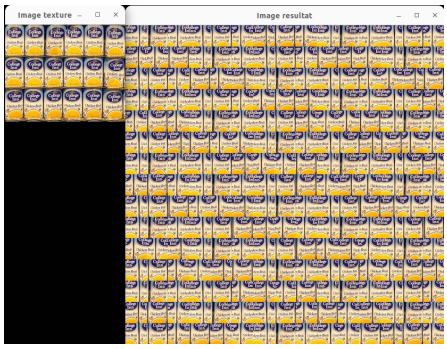
Initially, my group had 4 members. But I texted them and they never replied. They don't seem to be working on this project. Then I decided to work alone and they agreed to it. With experience after many times participating in **ICPC regional (ASEAN-Vietnam)**, the algorithm is not too complicated for me. So it is not impossible for me to complete this project alone.

2 Required work

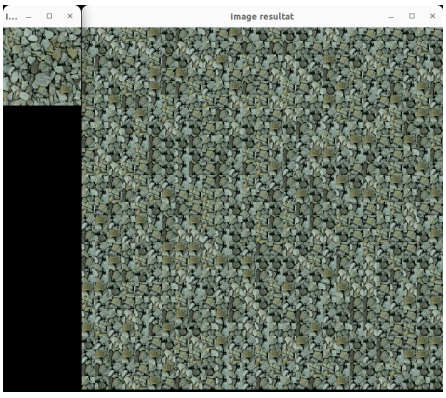
2.1 Experiment with algorithms (a) and (b) provided with different images, and different parameters.



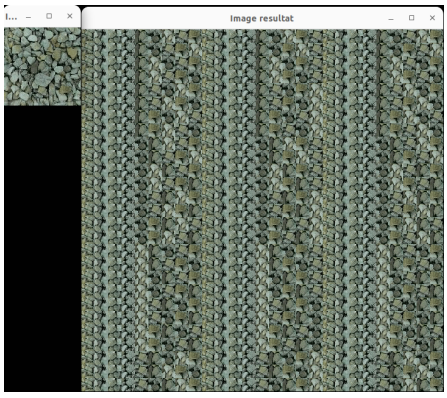
(a) ./tp images/canettes2.tif 0 3 20 600 600



(b) ./tp images/canettes2.tif 1 3 20 600 600

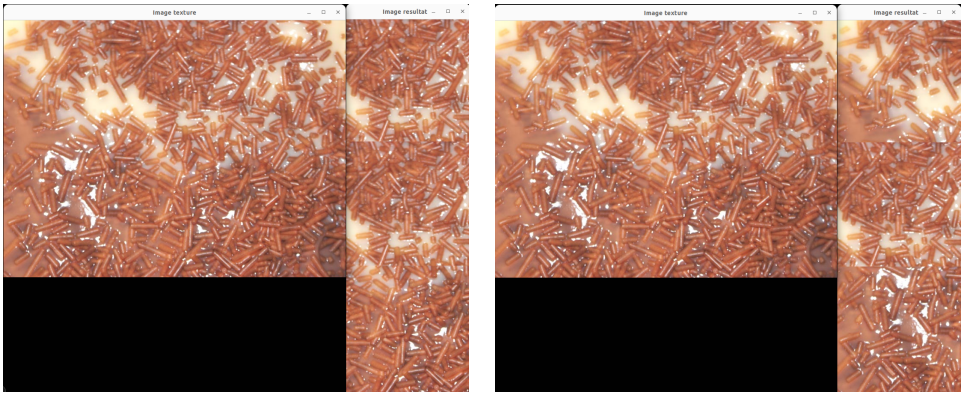


(c) ./tp gravier.tif 0 3 20 600 600



(d) ./tp gravier.tif 1 3 20 600 600

Figure 1: Experiment with algorithms (a) and (b) provided with different images.



(a) ./tp images/13.jpg.tif 0 2 10 500 1000 (b) ./tp images/13.jpg.tif 1 2 10 500 1000

Figure 2: Experiment with algorithms (a) and (b) provided with different images, and different parameters.

2.2 Implement a variant of the algorithm (a). A random block permutation generator balances the number of occurrences of each block.

Done in **permuteur.cpp**. There are 3 algorithms that generate permute: dictionary order, heap recursive, and heap iterative. See the code for more.

Change the **testpermute.cpp** file to test more cases. See the code and README.md for more.

2.3 What do e and E represent?

In the equation:

$$E_{ij} = e_{ij} + min\left(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1}\right) \tag{1}$$

Variable e_{ij} is the distance at point (i, j). Calculate by $e_{i.j} = \left|B^1_{i,j} - B^2_{i,j}\right|$.

Variable E is the total cost for the cut line.

2.4 What is the relationship between the cost of the optimal cut and the E_{ij}?

Variable E_{ij} is the minimum cost for the cut line end at point (i, j), starting from $i = 0$. So the cost of the optimal cut is $\min_{j\in[0,w]}(E_{hj})$. Where h is the height of the intersection area, w is the width of the intersection area. The optimal cut will start from line 0 and end at the final line.

2.5 Clarify the concept of optimal cut on the following example

e =

1	3	2	1
2	1	2	3
1	3	4	2
2	4	3	1
4	3	1	2

E =

1	3	2	1
3	2	3	4
3	5	6	5
5	7	8	6
9	8	7	8

E=

1	3	2	**1**
3	2	**3**	4
3	5	6	**5**
5	7	8	**6**
9	8	**7**	8

C=

3
2
3
3
2

And the minimum cost total is 7.

2.6 Estimate the complexity of a naive recursive solution for computing the optimal cut.

Call $new_h, new_w; h, w; h', w'$ respectively is height, width of new image; height, width of original image; height, width of intersection. w' is input, $nb\_block = racineNombreBlocs^2$, $h' = block_h = h/nb\_block, block_w = w/nb\_block$. $verti = new_h/(block_h - 2 \times w'), hori = new_w/(block_w - 2 \times w')$.

At each point, it calls to 3 other points while the height equals 0. For each i in the range of w', we need to calculate $E_{ih'}$. For each j in the range of h', we call to 3 next points. So it costs $O\left(3^{h'}\right)$ to calculate $E_{ih'}$ for each point i in the range of w'. Totally it costs $O\left(w' \times 3^{h'}\right)$ in time complexity.

Here, space complexity has two ways of looking at it. According to programming languages, we usually don't use any variables when calling recursively (shortest way) so $O(1)$. However, when viewed at the assembly level, for each recursive call a next pointer, parameters, and return value are added to the stack. There is a maximum h' time stack recursive all the time so it is $O(h')$ pointers in stack very easy stack overflow. So the space complexity is $O(h')$.

But we also need to calculate the cutline. There are two ways to calculate the cut line. One is to create a new array to save the cutline for each recursive call. This way increases the space complexity to $O\left(h'^2\right)$, and time is the same. The second way is, after getting min we call that point again to set the cutline. This way increases the time complexity to $O\left(w \times 4^{h'}\right)$, and space is the same.

Combine with question 2.9 to get the total complexity.

2.7 Implement a recursive solution without redundant computation.

Done in file **raccordeur_recursif.cpp**.

2.8 Implement an iterative solution.

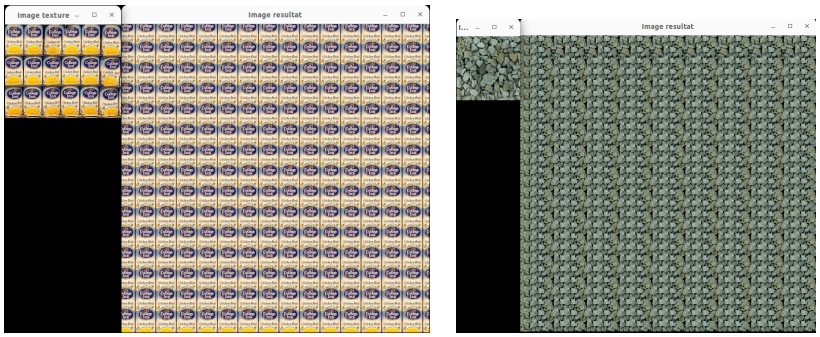
Done in file **raccordeur_iteratif.cpp**.

2.9 Additional question: what is the overall complexity of the algorithm (c) taking into account all the parameters?

Note: (time, space)
Compute between 2 blocks (recursive, naif, travel again): $O\left(w' \times 4^{h'}\right)$, $O(h')$
Compute between 2 blocks (recursive, naif, new arrays): $O\left(w' \times 3^{h'}\right)$, $O\left(h'^2\right)$
Compute between 2 blocks (recursive, memory): $O(w' \times h')$, $O(w' \times h')$
Compute between 2 blocks (iterative): $O(w' \times h')$, $O(w' \times h')$
For each algorithm, call O_1, O_2 respectively is the time, and space complexity for 2 blocks.
Compute all without cache: $O(verti \times hori \times nb\_block \times O_1)$, O_2
Compute all with cache cost: $O\left(O_1 \times min\left(verti \times hori \times nb\_block, nb\_block^2\right)\right)$, $O\left(O_2 + nb\_block^2\right)$
Compute all with cache cost, cut: $O\left(O_1 \times min\left(verti \times hori \times nb\_block, nb\_block^2\right)\right)$, $O\left(O_2 + nb\_block^2 \times h'\right)$

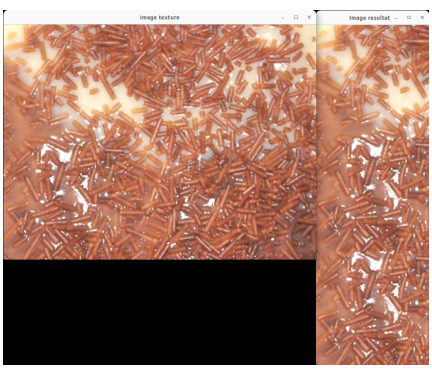
3 Compare

3.1 Experiment with algorithm (c)

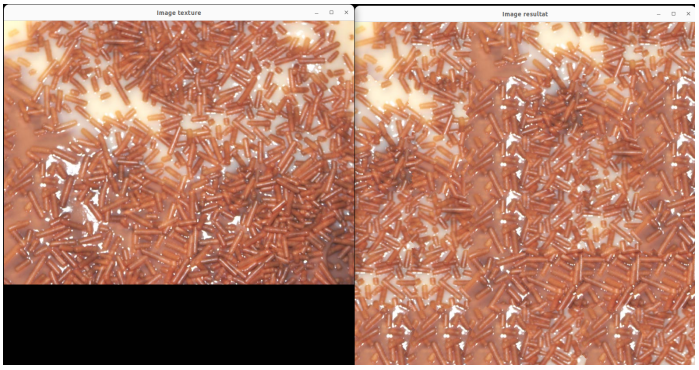


(a) ./tp images/canettes2.tif 5 3 20 600 600 (b) ./tp gravier.tif 5 3 20 600 600

Figure 3: Experiment with algorithms (c) provided with different images.



(a) ./tp images/13.jpg.tif 5 2 10 500 1000



(b) ./tp images/13.jpg.tif 5 4 20 800 800 1 1

Figure 4: Experiment with algorithms (c) provided with different parameters

3.2 Time comparison of algorithm implementation versions (c)

	Test1	Test2	Test3	Test4	Test5	Test6
Recursivenaif,noroute	801ms±13.7ms	12.4s±102ms	>1min	>1min	>1min	>1min
Recursivenaif,route	756ms±4.37ms	3.37s±37.9ms	12.5s±73.6ms	>1min	>1min	>1min
Recursivememo	653ms±659µs	284ms±18.5ms	285ms±232µs	188ms±1.65ms	622ms±25.1ms	7.78s±50.1ms
Iterative	546ms±2.82ms	240ms±12.2ms	238ms±18.7ms	203ms±5.87ms	493ms±23.1ms	7.16s±377ms

Table 1: Time comparison of implementation versions of the algorithm (c). For more information about testcase parameters see **Compare algo.ipynb**

	Normal	Cache cost	Cache cut	Cache cost, cut
Algo 3, test 7	9.61 s ± 21.7 ms	6.57 s ± 8.15 ms	9.91 s ± 241 ms	7.48 s ± 155 ms
Algo 6, test 7	3.87 s ± 32.8 ms	2.65 s ± 32.2 ms	3.79 s ± 66.6 ms	3.03 s ± 104 ms
Algo 4, test 8	9.6 s ± 2.01 ms	611 ms ± 98.5 µs	10.3 s ± 85.6 ms	218 ms ± 1.84 ms
Algo 5, test 8	7.43 s ± 84.6 ms	542 ms ± 65.2 ms	7.83 s ± 40.1 ms	204 ms ± 8.04 ms

Table 2: Time comparison of with and without cache implementation versions of the algorithm (c). For more information about testcase parameters see **Compare cache.ipynb**

In the cache cut version, it's not much different than without cache, because we don't cache costs so we need to calculate the cost and coupe again for each pair of blocks. Only faster when the index of the best block is found, but slower at assigning a useless cut between pair of blocks (not best).

4 Advanced Optimize

4.1 Fix some bugs

Line 28^{th} file **image_4b.cpp** changed to

```
if(tif) // fix bug
```

Line 579^{th} and 593^{th} in new **textureur.cpp** (Line 433^{th} and 439^{th} in old **textureur.cpp**) changed from

```
Coupe_GD(raccordeur, &table_blocs[ind], &table_blocs[res_bindex-¿get(c-1, l)], coupe);
...
Coupe_HB(raccordeur, &table_blocs[ind], &table_blocs[res_bindex-¿get(c, l-1)], coupe);
```

to

```
Coupe_GD(raccordeur, &table_blocs[res_bindex-¿get(c-1, l)], &table_blocs[ind], coupe);
...
Coupe_HB(raccordeur, &table_blocs[res_bindex-¿get(c, l-1)], &table_blocs[ind], coupe);
```

Because the previous block is the front parameter.

4.2 Stored cut to a new array

For a recursive algorithm, there are two ways to save the path. One is to travel again, the other is to create a new array to save the way each time you travel. The first way is slower but saves memory. The second way is faster but takes more memory

4.3 Cache cost

Modified "texture" to be able to save the distance of a pair of blocks. Because every time we want to add a block, we have to go through all the blocks to calculate the cost with the previous block. There will be 2 computations, one **vertically** and one **horizontally** (ignored if there is no neighbor block). To minimize these computations we will create two 2-dimensional arrays of size $[nb\_bloc \times nb\_bloc]$. One to save cost vertically and one horizontally. All will be initialized to -1, then if $A[i,j] == -1$, we need to calculate and save it in $A[i,j]$. Next time we need to use it, just get $A[i,j]$.

4.4 Cache cut

Same as cache cost 4.3. We create two 3d-arrays (one **vertically** and one **horizontally**) to save cut between two blocks.