



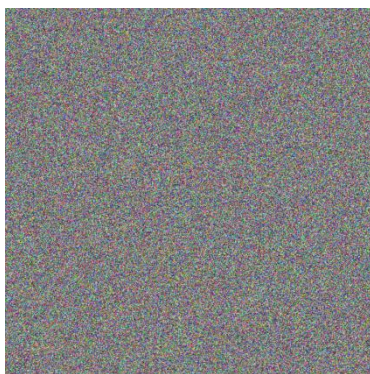
Università degli Studi di Salerno

Dipartimento di Informatica

Corso di laurea Magistrale in Informatica

Relazione per il progetto del corso di Compressione Dati

Schema di Watermarking Robusto per un Bitstream JPEG Cifrato



Il team

Gianmarco Beato 0522500782

Silvia Castelli 0522500758

Silvio Corso 0522500792

Francesco Maria d'Auria 0522500752

Carmine Tramontano 0522500865

Angela Vecchione 0522500814

Il docente

Prof. Bruno Carpentieri

Abstract

JPEG è uno standard di compressione immagine molto popolare su Internet, ed in questo lavoro da noi svolto, per ottenere simultaneamente la protezione del contenuto di una immagine e del copyright, viene fornito uno schema di watermarking robusto per un bitstream JPEG crittografato mediante lo shuffling dei pixel. Il watermark può essere incorporato direttamente nel bitstream JPEG crittografato effettuando delle opportune modifiche ai relativi coefficienti DC quantizzati senza eseguire la decrittografia o la decompressione; Il metodo proposto dal paper di riferimento dimostra un'elevata fedeltà visiva dell'immagine cifrata e con watermark, rispetto all'immagine originale.

Indice

1. Introduzione.....	1
2. Background JPEG.....	3
2.1 Lettura dell'immagine e trasformazione dello spazio cromatico.....	4
2.2 Estrazione di un blocco 8×8 pixel dall'immagine originale	5
2.3 Trasformata discreta del coseno	5
2.4 Quantizzazione dei coefficienti DCT	6
2.5 Codifica entropica	7
2.6 Scrittura del file JPEG	8
3. Progettazione ed implementazione.....	9
3.1 Organizzazione del progetto	9
3.2 Approccio iniziale	10
3.3 Algoritmi di permutazione e cifratura	12
3.3.1 Algoritmi di permutazione e depermutazione dei blocchi MCU.....	12
3.3.2 Algoritmi di cifratura e decifratura.....	16
3.4 Algoritmo di watermarking	21
3.4.1 Estrazione del watermark	23
3.4.2 Algoritmo di verifica del watermark	24
4. Esperimenti ed analisi dei risultati.....	25
4.1 Testing	30
5. Conclusioni e sviluppi futuri.....	33
Sitografia	34

Link GitHub del nostro progetto: https://github.com/carminet94/Robust_Watermarking_JPEG

1. Introduzione

L'elaborazione delle immagini nel dominio della cifratura è un argomento che ha ricevuto di recente una notevole attenzione da parte di numerosi ricercatori. Il nostro lavoro, così come è stato proposto dal paper di riferimento [1], si è focalizzato nell'implementare uno schema di watermarking per un bitstream JPEG cifrato che sia efficiente e robusto. Lato codificatore, dopo aver suddiviso l'immagine in input in blocchi di dimensione 16×16 pixel e dopo averli permutati tra loro in base ad una chiave, si è passato alla cifratura effettuando lo shuffling dei soli pixel corrispondenti agli AC, in base al valore di una chiave fornita, rimanendo in chiaro i pixel corrispondenti ai DC; dopodichè si è passato all'applicazione del watermark e si è compressa l'immagine mediante le varie fasi previste dall'algoritmo JPEG; il watermark è stato incorporato direttamente all'interno dell'immagine effettuando delle modifiche ai coefficienti DC quantizzati senza eseguire la decrittografia o la decompressione. La modifica dei coefficienti DC ha un notevole impatto sulla qualità visiva dell'immagine, infatti effettuando delle modifiche improprie ai coefficienti DC la fedeltà visiva dell'immagine con watermark degenera ampiamente.

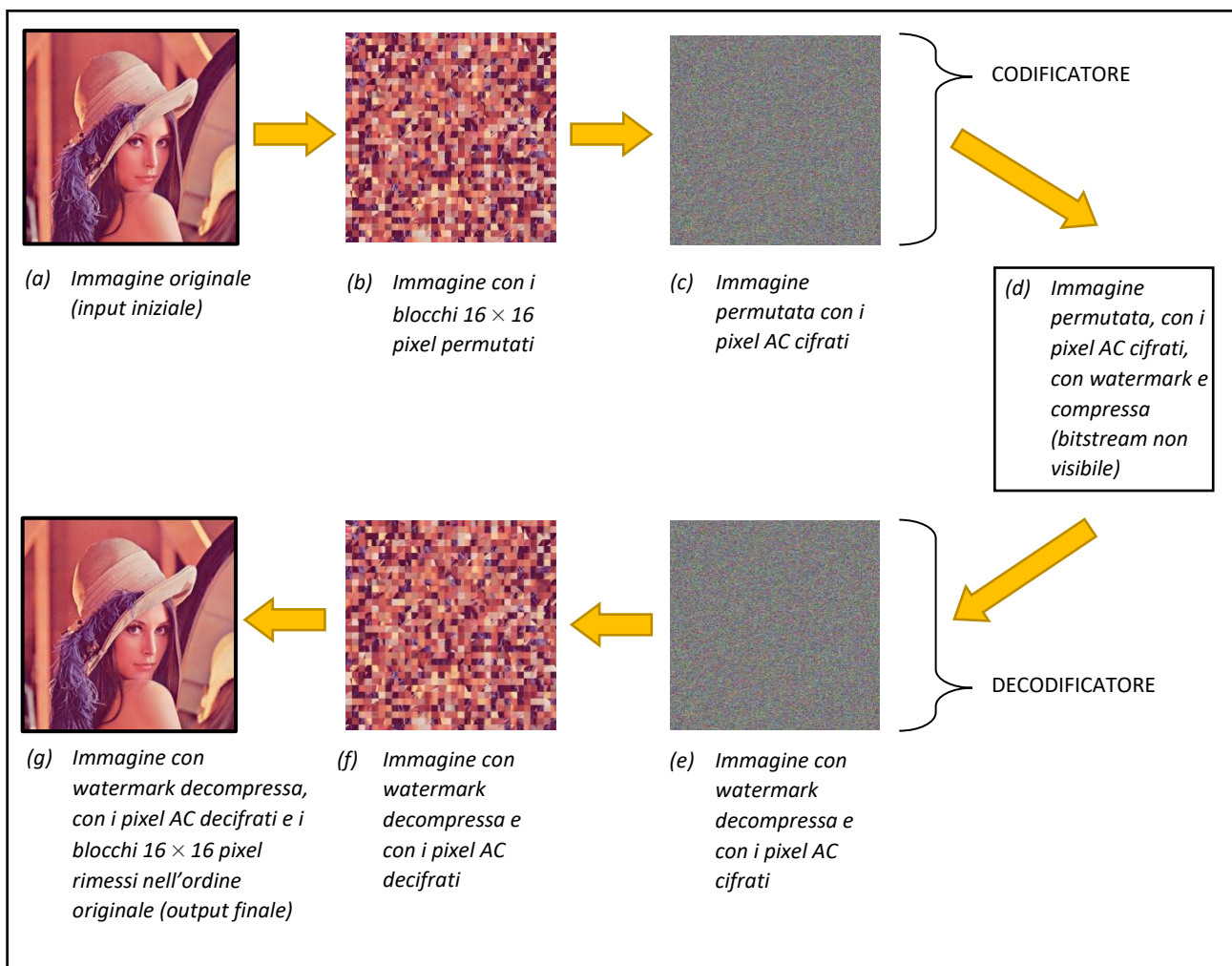


Fig.1. Una panoramica generale del processo implementato.

Dal momento che il comune algoritmo JPEG nelle prime fasi del suo operato suddivide l'immagine in blocchi DCT di dimensione 8×8 pixel, tali blocchi sono stati raggruppati a loro volta in blocchi quadrati di dimensione 16×16 pixel, chiamati *MCU* (Minimum Code Unit), contenente al loro interno 4 blocchi di dimensione 8×8 pixel, così come mostrato nella figura 2.

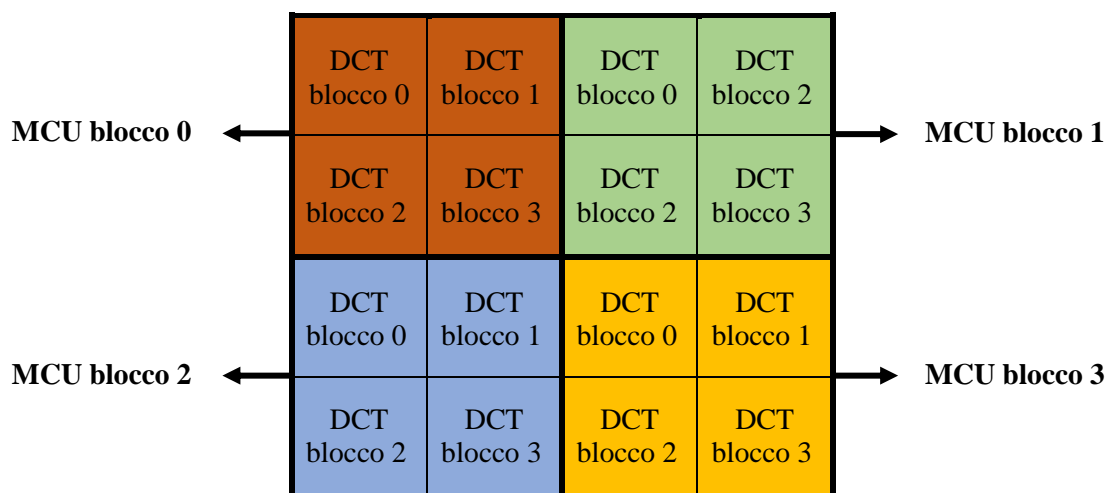


Fig.2. La relazione tra i blocchi DCT di dimensione 8×8 pixel e i blocchi MCU.

Ed è proprio a livello dei blocchi MCU che agisce la strategia di watermarking implementata in questo lavoro, per applicare, appunto, il watermark. Chiunque possiede la chiave di estrazione del watermark può estrarre il watermark stesso dal bitstream decifrato; ma anche quando il bitstream è cifrato e non si possiede la chiave per decifrarlo si può comunque ancora estrarre il watermark incorporato conoscendo la sua chiave di estrazione.

Lato decodificatore, infine, per ottenere l'immagine con watermark decompressa, decifrata e con i blocchi 16×16 pixel rimessi nell'ordine originale, è stato eseguito il processo inverso rispetto a quanto descritto precedentemente.

Il presente documento si articola come segue: nella sezione 2 verrà descritto generalmente l'algoritmo JPEG con tutte le sue fasi. Nella sezione 3 verranno esposti in dettaglio gli algoritmi di permutazione/depermutazione, cifratura/decifratura e di watermarking da noi implementati, oltre all'approccio iniziale e all'organizzazione del progetto. Nella sezione 4 verranno mostrati degli esperimenti e verranno analizzati i risultati e mostrati dei test. Infine nella sezione 5 si passerà alle conclusioni con menzione a degli eventuali sviluppi futuri.

2. Background JPEG^[2]

JPEG (acronimo di **J**oint **P**hoto**t**aphic **E**xpert **G**roup) è uno degli standard più utilizzati e conosciuti per la compressione di immagini a tono continuo, sia a livelli di grigio che a colori. Come è stato già detto in precedenza, nell'articolo analizzato viene proposto uno schema robusto di watermarking per bitstream JPEG crittografati a colori. L'algoritmo JPEG è stato di conseguenza analizzato fase per fase in modo da poter studiarne il comportamento per apportare le eventuali modifiche ai fini dell'implementazione dell'algoritmo di crittografia e di watermarking.

Lo standard JPEG definisce due metodi di compressione di base, uno basato sull'uso della Trasformata Discreta del Coseno (DCT) con compressione di tipo "lossy", ovvero con perdita di informazione, l'altro sull'uso di un metodo predittivo con compressione di tipo "lossless" cioè senza perdita di informazione.

Al fine di implementare l'algoritmo proposto nell'articolo di Chang et Al. è stato analizzato in modo particolare l'algoritmo base di tipo lossy detto "baseline".

L'algoritmo base è anche conosciuto come "sequential encoding", poiché ogni componente dell'immagine è codificata singolarmente dall'alto verso il basso e da sinistra verso destra.

JPEG Baseline definisce un ricco schema di compressione, adeguato alla maggior parte delle applicazioni, ed è composto dai seguenti passi:

- Lettura dell'immagine e trasformazione dello spazio cromatico;
- Estrazione di un blocco 8x8 pixel dall'immagine originale;
- Calcolo della Trasformata Discreta del Coseno (DCT) per ogni elemento del blocco;
- Quantizzazione dei coefficienti DCT;
- Codifica entropica;
- Scrittura del File.

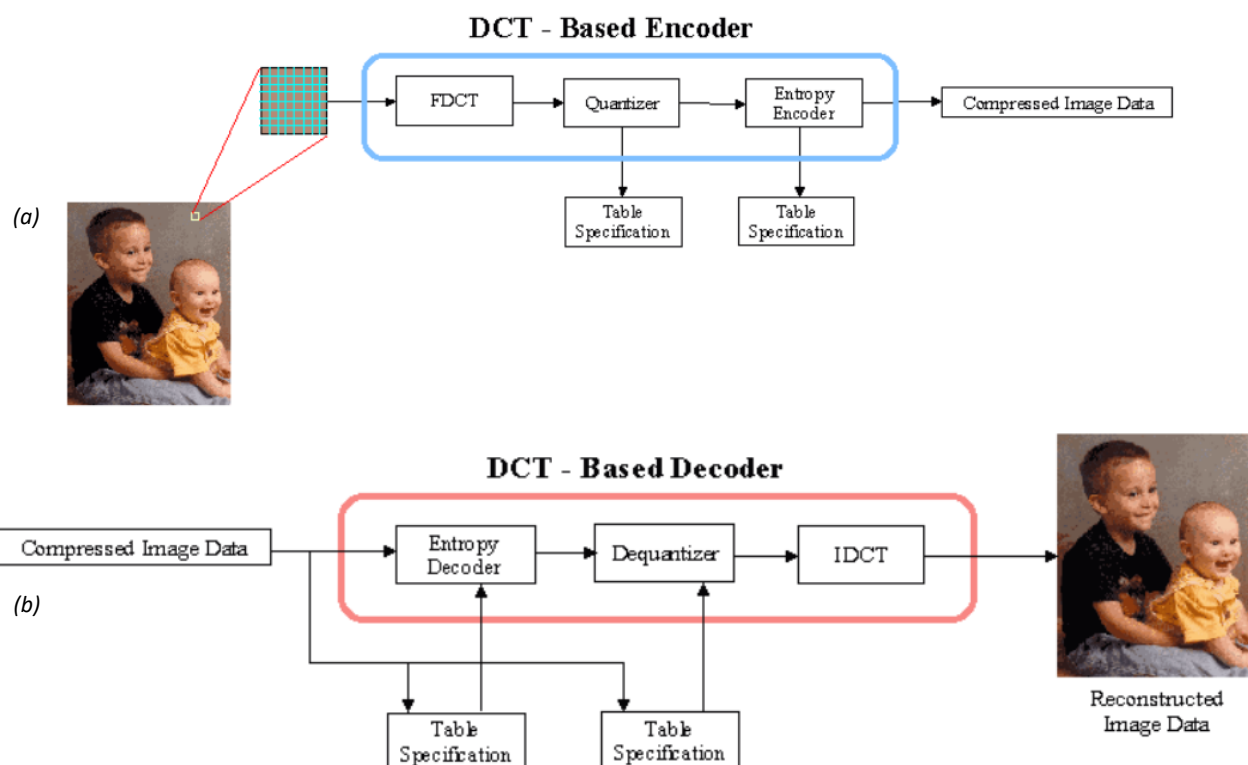


Fig.3. Le fasi di compressione (a) e di decompressione (b) dell'algoritmo JPEG.

Per la fase di decompressione i passi da seguire sono gli stessi seguendo però il percorso inverso rispetto alla compressione.

Nei paragrafi seguenti viene pertanto data in dettaglio una definizione delle singole fasi.

2.1 Lettura dell'immagine e trasformazione dello spazio cromatico

In questa fase prima di tutto viene letta da file l'immagine non ancora compressa. I dati letti dal file sorgente saranno poi rappresentati da una matrice tridimensionale, in cui ogni componente si riferisce alle tre componenti a colori: Red, Green e Blue (RGB).

In seguito, l'informazione RGB viene convertita in un altro spazio di colore detto YCbCr. In pratica l'immagine viene rappresentata tramite la sua luminanza (Y, che rappresenta l'intensità luminosa di ogni pixel) e la sua cromaticanza (CbCr, cioè l'informazione cromatica o di colore).

La luminanza contiene la maggior parte delle informazioni significative sull'immagine stessa. Gli altri due canali contengono invece informazioni cromatiche e hanno importanza minore al fine di ricostruire l'immagine originale. Si può sfruttare questa trasformazione da un dominio ad un altro, per scartare più informazione nelle componenti cromatiche che in quella della luminosità, questo avviene in quanto l'occhio umano è meno sensibile alle variazioni di frequenze cromatiche che a quelle di luminosità. Questa trasformazione introduce degli errori di arrotondamento che però sono ininfluenti rispetto alla quantità di errore introdotta dallo stesso algoritmo JPEG.

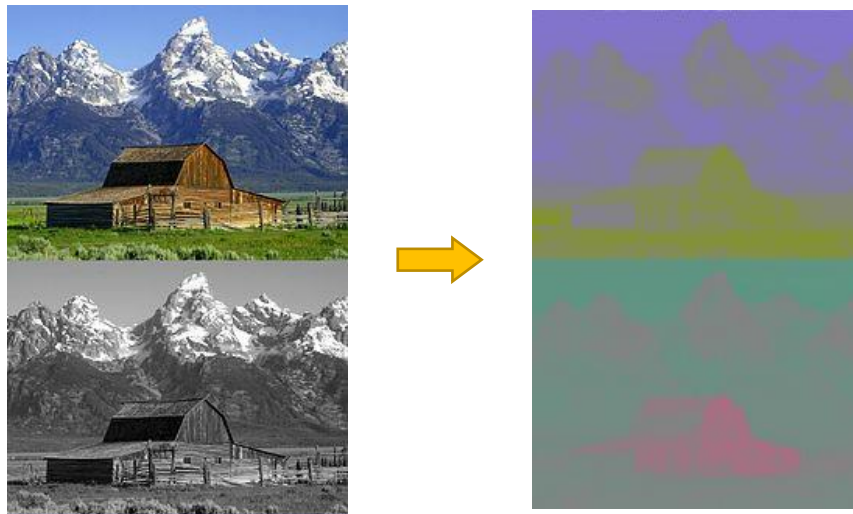


Fig.4. Immagine RGB convertita nelle componenti YCrCb

2.2 Estrazione di un blocco 8×8 pixel dall'immagine originale

Prima della fase di estrazione dei blocchi si effettua innanzitutto un controllo sulla dimensione dell'immagine. Difatti quest'ultima, per essere compressa con l'algoritmo di JPEG deve avere un'altezza e larghezza multipla della dimensione del blocco da estrarre.

Segue poi un'operazione di shift a sinistra di 128 pixel, in questo modo il range dei valori dell'immagine non va più da 0 a 255 ma da -128 a 127. La matrice così ottenuta viene suddivisa in blocchi di 8×8 su cui poi si applicherà la trasformata discreta del coseno.

11	16	21	25	27	27	27	27
16	23	25	28	31	28	28	28
22	27	32	25	30	28	28	28
31	33	34	32	32	31	31	31
31	33	34	32	32	31	31	31
33	33	33	33	32	29	29	29
34	34	33	35	34	29	29	29
34	34	33	33	35	30	30	30

Fig.5. Matrice 8×8 con i relativi valori.

2.3 Trasformata discreta del coseno

Il passo successivo del processo è il più importante al fine della compressione. La trasformata discreta del coseno viene applicata ai valori del blocco di 8×8 pixel, estratto precedentemente. I valori di ciascun blocco vengono quindi trasformati come valori in frequenza.

La DCT è invertibile senza perdita di informazioni, a parte qualche errore di arrotondamento dovuto ai termini del coseno ed è definita come:

$$F(u, v) = \frac{1}{4} C(u)C(v) \left[\sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right]$$

Fig.6. La trasformata discreta del coseno.

Vengono ottenuti per ogni blocco 64 coefficienti che non saranno più a frequenza discreta. In tale modo si potranno eliminare i dati ad alta frequenza senza toccare quelli a bassa frequenza.

In ognuno dei nuovi blocchi 8×8 l'informazione a bassa frequenza si trova nell'angolo in alto a sinistra mentre spostandosi verso l'angolo in basso a destra si va verso i valori di frequenza più alti. Da tale osservazione si può evincere che bastano i valori in alto a sinistra a definire l'intera matrice.

Il primo coefficiente del blocco è chiamato “coefficiente DC” ed ha un valore più grande rispetto ai restanti 63 che sono chiamati “coefficienti AC”. Da notare però che l'applicazione della trasformata non ha effettuato ancora alcuna compressione.

Qui di seguito viene riportato un esempio della modifica di un blocco 8x8 in un blocco 8x8 a cui viene applicata la trasformata discreta del coseno.

235.6250	-2.8073	-9.1411	-4.1851	-0.3750	-3.1184	-1.4903	2.7734
-23.9174	-17.3609	-5.3178	-2.3956	-3.4929	-1.2309	0.8164	0.2551
-11.9060	-7.2085	-4.1668	0.9520	2.5126	-0.1775	-1.6402	-0.6298
-4.3656	-2.5517	-1.0140	0.1197	1.6479	1.9776	-0.5554	-2.2967
1.8750	-1.9223	1.1432	0.6098	-0.1250	0.8327	0.4735	-0.7429
-0.0606	1.3824	0.5347	-0.1134	0.3494	1.0424	0.5913	-0.1803
-3.5922	-0.0371	1.1098	-0.1962	-1.4467	-0.1985	1.6668	1.6578
-2.3359	-0.8359	-0.0301	-0.6404	-1.3320	-0.4814	0.9782	1.1988

Fig.7. Applicazione della trasformata discreta del coseno alla matrice.

2.4 Quantizzazione dei coefficienti DCT

La parte lossy dell'algoritmo JPEG avviene attraverso un processo detto quantizzazione, in cui i coefficienti ad alta frequenza, quindi meno importanti, vengono eliminati, ottenendo così la qualità dell'immagine richiesta al fine di scartare le informazioni meno significative a livello visuale.

La quantizzazione è la fonte principale della perdita di informazione durante la compressione basata su DCT. Ogni elemento del blocco 8x8 viene diviso per un coefficiente di quantizzazione distinto ed il risultato così ottenuto viene arrotondato all'intero più vicino. I 64 coefficienti DCT sono quantizzati uniformemente mediante una tabella di quantizzazione anch'essa a 64 elementi rappresentati da interi nell'intervallo che va da 1 a 255.

La quantizzazione riduce l'ampiezza dei coefficienti DCT il cui contributo è nullo o comunque basso per la qualità dell'immagine. Questo è quindi il processo fondamentale che porta all'eliminazione dell'informazione meno significativa trattandosi di un mapping multi-a-uno.

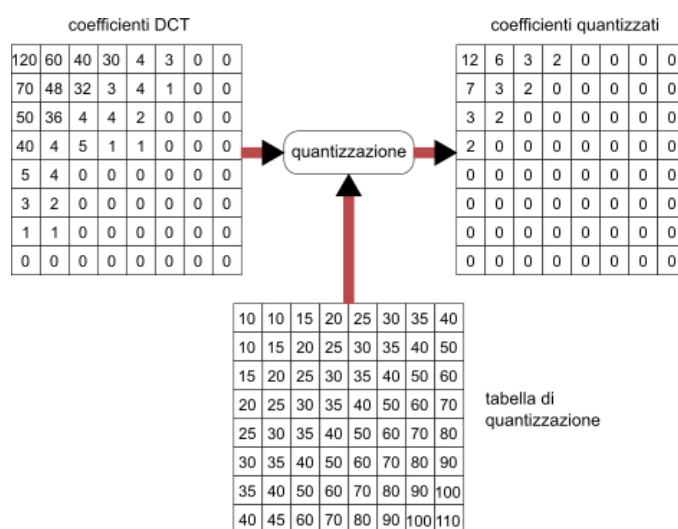


Fig.8. Il processo di quantizzazione.

2.5 Codifica entropica

Nella fase di codifica entropica si realizza un'ulteriore compressione. Questa fase è divisa fondamentalmente in due ulteriori sotto-fasi:

- **Sequenza zig-zag dei coefficienti quantizzati:** osservando i coefficienti quantizzati di un blocco 8x8 i valori diversi da zero sono concentrati nella parte in alto a sinistra della matrice. In questa fase quindi si ordinano i coefficienti in base alla frequenza in modo da poter trovare i valori pari a 0 alla fine della matrice, creando così sequenze di zeri che permettono una più efficace compressione.

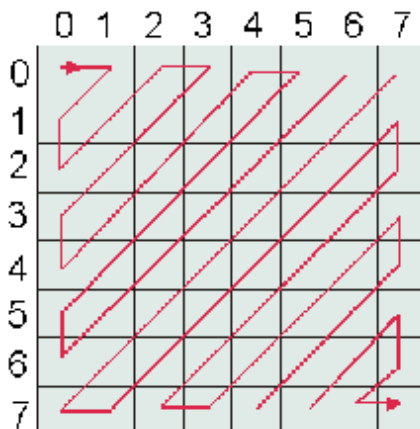


Fig.9. Il percorso a zig-zag sulla matrice

Essendo il coefficiente DC un numero molto elevato si riduce il suo valore sottraendogli il coefficiente DC del blocco precedente. Per tale valore verrà dato in output un simbolo intermedio costituito da: (Size, Ampiezza). “Size” rappresenta il numero di bit necessari a rappresentare il coefficiente DC ridotto e “Ampiezza” rappresenta il valore stesso del coefficiente DC ridotto.

I restanti 63 coefficienti AC vengono invece rappresentati usando i simboli intermedi seguenti: (Runlength, Size) (Ampiezza). Dove “RunLength” è il numero di zeri consecutivi, “Size” mantiene lo stesso significato dato in precedenza per i coefficienti DC, mentre “Ampiezza” rappresenta il primo numero diverso da zero che viene trovato dopo la sequenza di zeri.

- **Codifica di Huffman:** l'algoritmo di Huffman è impiegato per ottenere una rappresentazione più compatta dei dati, utilizza le frequenze statistiche con cui un simbolo compare in una sequenza. I simboli che compaiono più frequentemente vengono rappresentati con sequenze di bit più corte, quelli meno frequenti con sequenze di bit più lunghe. Vengono impiegate due tabelle di codifica di Huffman utilizzate in base al tipo di coefficiente da codificare e in cui viene associato a ciascun simbolo intermedio della sequenza, ottenuto dalla fase di zig-zag, un codice di Huffman. Alla fine di queste due fasi intermedie avremo come risultato una sequenza di bit.

2.6 Scrittura del file JPEG

La fase finale della compressione JPEG Baseline è quella di scrittura del file. Il file viene quindi utilizzato per memorizzare il bitstream JPEG, ovvero l'immagine compressa rappresentata dai coefficienti DC e AC.

Il file in generale è costituito da una serie di segmenti che iniziano con un “marker” che identifica il tipo di segmento memorizzato seguito poi dalla Size e dall'Ampiezza se si tratta dei coefficienti DC o dalla Runlength e dalla Ampiezza se si tratta dei coefficienti AC.

3. Progettazione ed implementazione

In questo capitolo verranno descritte in dettaglio le scelte implementative che sono state adottate e le motivazioni che ci hanno portato ad utilizzarle. Il nostro processo implementativo è passato attraverso diverse fasi di sviluppo, non solo per affrontare separatamente le tecniche proposte dal paper di riferimento, ma anche per verificare se effettivamente fossero valide ed applicabili. Il primo sottoparagrafo riguarderà l'organizzazione dei file del progetto, mentre il secondo paragrafo riguarderà proprio l'approccio iniziale che, come constateremo, è stato accantonato in quanto non utile o non applicabile al nostro contesto.

3.1 Organizzazione del progetto

Il linguaggio di programmazione da noi scelto è Python, diverse sono le motivazioni che ci hanno portato ad utilizzarlo:

- È un linguaggio semplice e veloce.
- È molto leggibile e i suoi moduli sono ben organizzati.
- Ha alcune librerie molto potenti che si adattano perfettamente al nostro contesto. (**numpy** in primis che permette di manipolare facilmente le informazioni relativi alle immagini).

Inoltre, anche l'IDE utilizzato, ovvero PyCharm [3] e il packet manager Anaconda, lavorano in perfetta sincronia con Python e la gestione dei suoi moduli facilitando di molto il lavoro.

Nell'IDE Pycharm in cui sono state condotte le fasi di implementazione e successiva sperimentazione, è possibile trovare vari file python (file con estensione “.py”) che vanno ad individuare tutto ciò che riguarda il progetto in esame. Di seguito viene fornita un'immagine di quello che è il raggruppamento dei file python sviluppati.

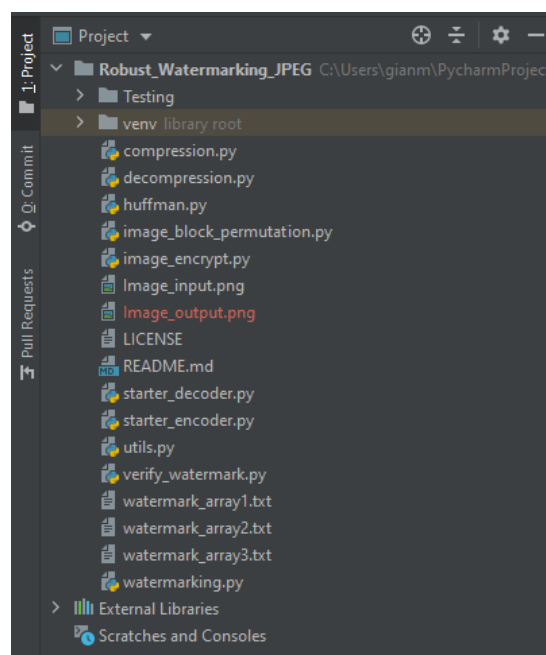


Fig.10. Struttura dei file del progetto

Adesso verranno brevemente analizzati nell'ordine di utilizzo, eccezion fatta per i due file *main* ovvero *starter_encoder.py* e *starter_decoder.py*:

1. La fase di codifica operata dalle funzioni contenute nello script *starter_encoder.py* dopo aver caricato in input l'immagine parte con la permutazione eseguendo le funzioni scritte nello script python *image_block_permutation.py*. Esso implementa la funzione di permutazione dell'immagine e in maniera analoga, nella fase di decodifica, anche la de-permutazione che verrà apportata all'immagine.
2. Successivamente si passa alla fase di cifratura, nella quale viene richiamato lo script *image_encrypt.py*. Tale script è deputato alla cifratura dell'immagine, attualmente permutata, per poi passarla al compressore. Lo script, quindi, implementa una funzione di cifratura e, analogamente a come detto in precedenza, anche una funzione di decifratura che servirà ovviamente in fase di decompressione.
3. Conclusa la fase di cifratura, si passa all'ultimo step, ovvero quello di compressione. In questo caso oltre alla compressione viene applicato anche il watermark. Banalmente per l'applicazione del watermark viene invocato lo script, scritto ad hoc sulla base del paper, *watermarking.py*. L'array rappresentativo del watermark verrà fornito come parametro attraverso file di testo (*watermark_array.txt*). Ed infine per la compressione viene richiamato lo script python inerente alla compressione JPEG *compression.py*. Esso, come da implementazione di JPEG, invocherà anche il processo inerente alla codifica di Huffman, andando a caricare le istruzioni presente nello script *huffman.py*.

Conclusi questi passaggi, si può considerare terminata la fase di compressione. Come già ampiamente accennato, la fase di decodifica (operata dalle funzioni contenute nello script *starter_decoder.py*) fa uso degli stessi script python appena descritti. Quindi l'immagine in output dalla fase di compressione viene data in input al decoder che esegue:

1. La decompressione dell'immagine mediante *decompression.py* che implementa JPEG decompression utilizzando anche *huffman.py*.
2. Successivamente l'immagine viene decifrata.
3. Dall'immagine viene estratto il watermark inserito in fase di compressione, mediante *watermarking.py*.
4. Infine viene de-permutata l'immagine per restituire in output l'immagine originale che corrisponde a *image_input_depermutation_original.png*.

Poi vi è lo script *verify_watermark.py* che si occupa della verifica di correttezza sul watermark. Ovvero se quest'ultimo, inserito in fase di compressione, viene perfettamente conservato anche in fase di decompressione, ritornando all'immagine originale. Infine abbiamo la cartella "Testing" che contiene i vari test effettuati e che verranno argomentati successivamente.

3.2 Approccio iniziale

Inizialmente la nostra attenzione si è focalizzata sulla scelta di una libreria python valida per l'implementazione dell'algoritmo JPEG. Quanto richiesto era presente in una repository github "**jpeg-python**" [4] che implementa esattamente tutti i passi, in maniera chiara, dell'algoritmo. Accertatici, dopo diversi test, del funzionamento della libreria, abbiamo esteso le sue funzionalità con la permutazione e la cifratura dell'immagine prima di comprimerla. Il passo di cifratura è quello che ha richiesto maggior studio e tentativi, in quanto non era ben specificato che tipo di algoritmo fosse necessario per cifrare l'immagine.

Inizialmente, si era pensato di cifrare il bitstream generato in seguito alla compressione invece di manipolare gli AC e i DC dell'immagine dall'interno dell'algoritmo JPEG. Python non offre molte funzioni per poter cifrare bitstream e le poche presenti non ci permettevano poi di utilizzare al meglio il cifrato per poter ritornare a qualcosa di utilizzabile dal nostro "decoder" (funzione per la decompressione). Ciò che si è pensato è stato quello di astrarre la parte di cifratura e di eseguirla esternamente con un tool java ad hoc, da noi creato,

che si occupasse della cifratura e la decifratura dei bitstream. Il tool, ottenibile da github [5], fa utilizzo della libreria *Java Bouncy Castle* [6] per poter implementare la crittografia e presenta un'interfaccia grafica molto semplice come, così mostrato dalla seguente immagine:

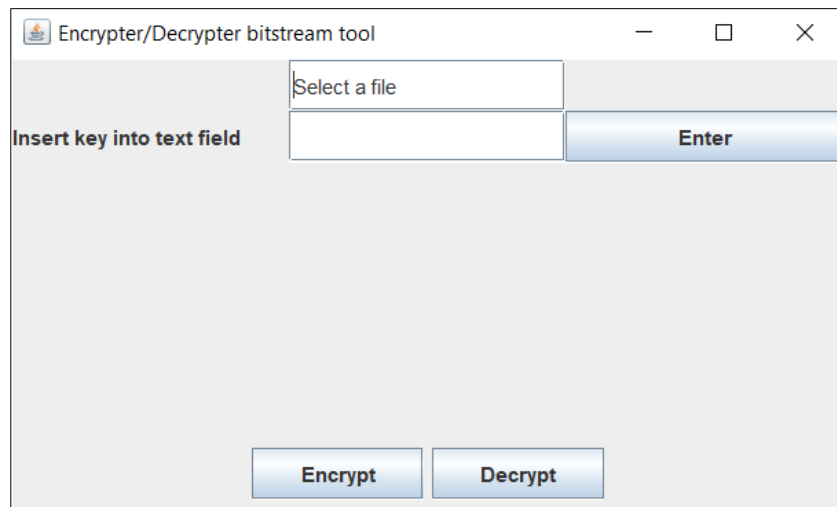


Fig.11. Il tool implementato di cifratura/decifratura di un bitstream

Prendendo in input una chiave e un bitstream rappresentativo dell'immagine vengono generati due file con estensione .txt, uno contenente il bitstream cifrato (output_encrypted.txt) e l'altro contenente il bitstream decifrato (output_decrypted.txt) e quindi uguale a quello passato inizialmente in input. Tuttavia, nonostante l'algoritmo fosse valido per cifrare dei bitstream abbiamo dovuto accantonarlo, questo principalmente per due problematiche riscontrate:

- Eccessivo aumento della dimensione del bitstream cifrato rispetto a quello in chiaro. Questo comportava un appesantimento nella generazione dei file generati e generalmente, nelle computazioni dell'algoritmo jpeg.
- Bitstream cifrato non più interpretabile dal "decoder". Il bitstream presentava una struttura non interpretabile dal nostro decoder per cui non era più possibile decomprimere ed ottenere una rappresentazione grafica dell'immagine cifrata.

Queste due motivazioni ci hanno portato all'allontanarci da questo tool e ad utilizzare altre tecniche che saranno presentate nei prossimi sottoparagrafi.

3.3 Algoritmi di permutazione e cifratura

Seguendo il processo definito dal paper di riferimento, abbiamo deciso di implementare le funzioni di permutazione e depermutazione dei blocchi 16×16 pixel dell'immagine e successivamente le funzioni di shuffling e de-shuffling sui blocchi 8×8 pixel e di applicare il processo di cifratura e decifratura sugli elementi dell'immagine.

3.3.1 Algoritmi di permutazione e depermutazione dei blocchi MCU

Per motivi di implementazione e di gestione delle operazioni successive di shuffling, cifratura e watermarking, è stata effettuata una trasformazione di tipo *raster* dei blocchi MCU (16×16 pixel) da permutare.

Per quanto riguarda la **permutazione**, l'immagine (rappresentata mediante un array tridimensionale di pixel) è stata per prima cosa suddivisa in blocchi di dimensione 16×16 pixel, poi sono stati presi tutti questi blocchi tridimensionali* “quadrati” (chiamati blocchi MCU, aventi, quindi, $16 \times 16 = 256$ pixel) e ciascuno di essi è stato trasformato in un blocco tridimensionale “rettangolare” di dimensione 1×256 pixel ed inserito in un array tridimensionale apposito. In questo modo la tabella risultante avrà un numero di righe pari al numero di blocchi MCU dell'immagine ed un numero di colonne pari a 256. Nella figura 12 è raffigurata concettualmente l'idea di base dell'algoritmo adoperato: se ad esempio l'immagine di partenza ha un numero n di blocchi MCU, evidenziati tramite i vari colori, pari a 4 (da notare che ogni blocco MCU è composto al suo interno da 4 blocchi DCT di dimensione 8×8 pixel), allora l'array tridimensionale risultante sarà composta da 4 righe e 256 colonne, quindi una riga per ogni blocco MCU. Ovviamente il nostro algoritmo lavora generalmente su un arbitrario numero n di blocchi MCU ($n = 4$ è una semplificazione per mostrarne facilmente l'idea).

* Si parla di tridimensionalità poiché ciascuno dei tre livelli dell'array rappresenterà un canale dell'immagine, ad esempio RGB o YCbCr.

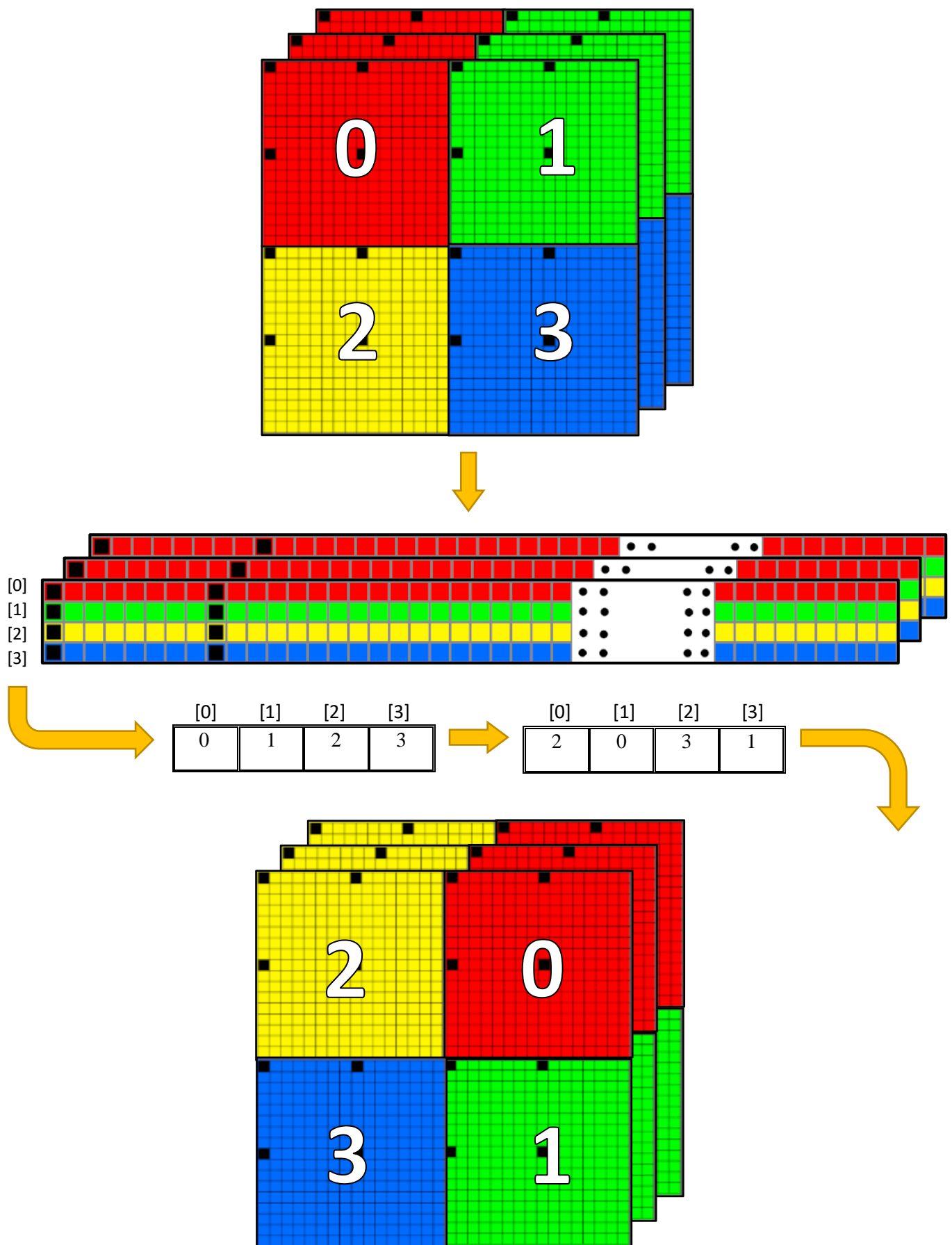


Fig.12 (a). Schema raffigurante l'algoritmo di permutazione dei blocchi MCU.

Successivamente, si è operato come segue:

- Sono stati presi gli indici (numeri) delle righe, le quali corrispondono ad un singolo blocco MCU 1×256 pixel.
- Tali numeri sono stati inseriti in un array.
- I numeri all'interno dell'array sono permutati secondo una funzione **random.sample()** con seed: il seed è generato a partire da un valore chiave (key) e può essere rigenerato a partire dallo stesso valore chiave in altre istanze.

```
random.seed(key)
array_permutation = random.sample(range(blocks_count), blocks_count)
```

Fig.13. Uno snippet di codice del file di progetto "image_block_permutation.py" raffigurante la permutazione dei numeri all'interno dell'array degli indici dei blocchi.

- Infine, sono stati ricostruiti i blocchi "quadrati" di dimensione 16×16 pixel a partire dai blocchi "rettangolari" 1×256 pixel e cioè dalle righe dalla tabella creata in precedenza, secondo l'ordine definito dall'array di indici/numeri permutati. E' stata così ottenuta l'immagine visibile con i blocchi permutati.



Fig.13. L'immagine originale a sinistra e l'immagine di output della permutazione con i blocchi MCU permutati a destra.

Lo stesso accade, in modo simile, con la fase di **depermutazione** (assumiamo che gli array siano tridimensionali):

- L'immagine permutata viene suddivisa in blocchi MCU di dimensione 16×16 pixel.
- Gli n blocchi MCU "quadrati" vengono trasformati ciascuno in blocchi "rettangolari" di dimensione 1×256 pixel ed inseriti in una apposita tabella, che avrà una dimensione $n \times 256$.
- Viene ricreato l'array dei numeri degli indici dei blocchi permutati, riottenendo lo stesso array avuto in fase di permutazione, allo stesso identico modo tramite la funzione random.sample() con seed (fig. 11.).

- Infine, sono stati ricostruiti i blocchi “quadrati” di dimensione 16×16 pixel a partire dai blocchi “rettangolari” 1×256 pixel e cioè dalle righe dall’array creato in precedenza, andando, però, a prelevarli man mano secondo l’ordine crescente degli indici/numeri all’interno dell’array. E’ stata così ottenuta l’immagine originale avente i blocchi rimessi nel loro ordine originale.

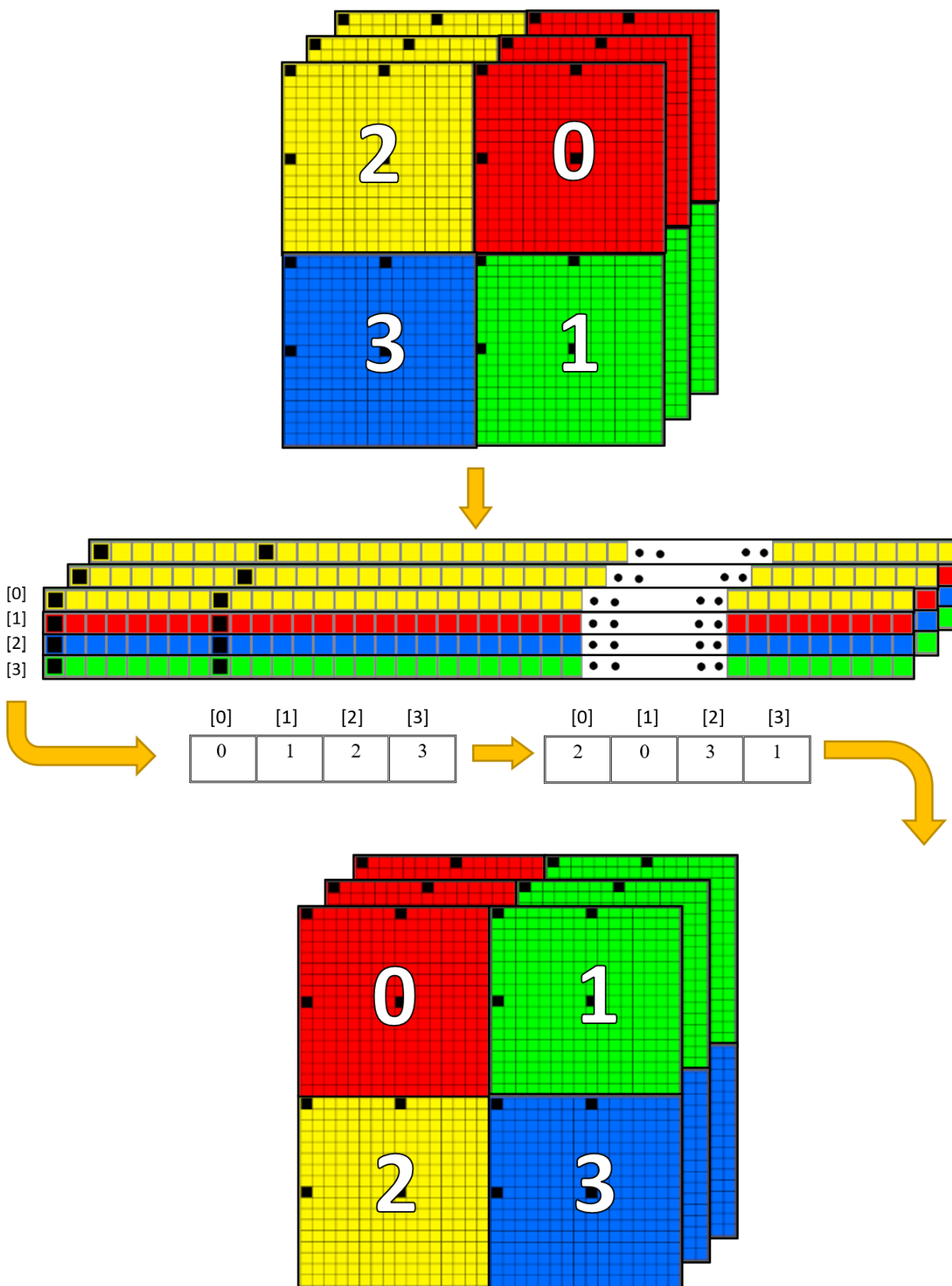


Fig.12 (b). Schema raffigurante l’algoritmo di depermutazione dei blocchi MCU.

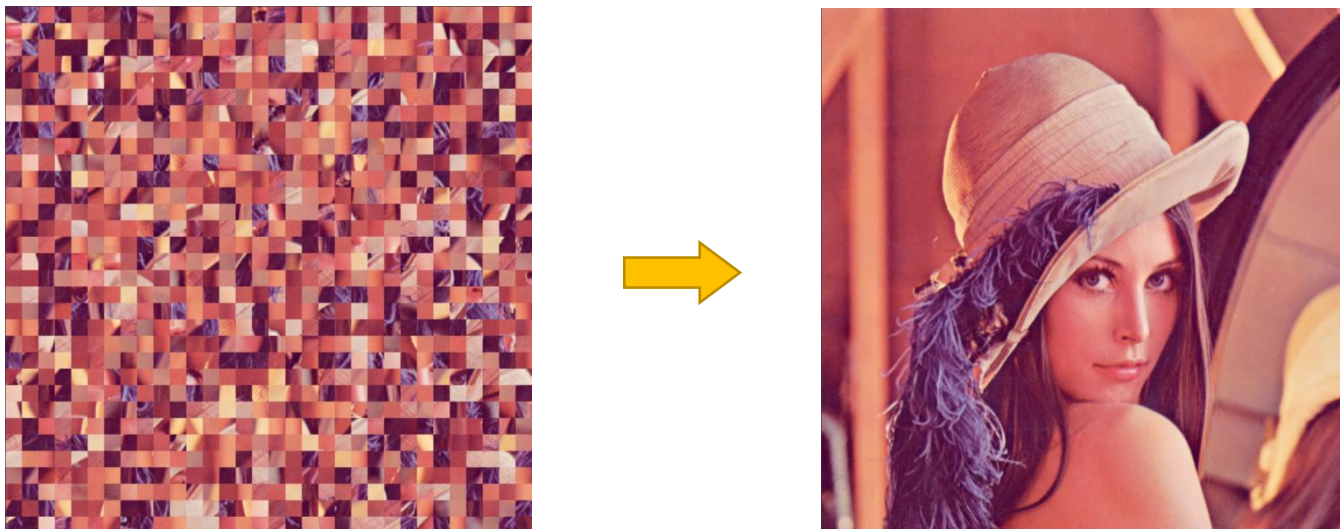


Fig.14. L'immagine permutata a sinistra e l'immagine di output della depermutazione con i blocchi MCU nella loro posizione originale a destra.

3.3.2 Algoritmi di cifratura e decifratura

Il processo di cifratura e decifratura viene effettuato con una libreria python chiamata “**imageshuffle**” disponibile su GitHub [7]. La tipologia di cifratura è basata sulla permutazione dei pixel. La funzione di cifratura prende in input l'immagine da cifrare ed una chiave simmetrica usata per la cifratura e produce in output una immagine avente i pixel shufflati.

Funzionamento generale dell'algoritmo di cifratura:

- L'immagine, espressa in RGB da 8 bit per ciascuna componente, è divisa in blocchi di taglia 8×8 pixel.
- Ogni componente R,G,B da 8 bit viene ulteriormente divisa in 4 bit superiori e 4 bit inferiori. In questo modo, otteniamo 6 canali differenti: R 4-bit-superiori, R 4-bit-inferiori, G 4-bit-superiori, G 4-bit-inferiori, B 4-bit-superiori, B 4-bit-inferiori. Chiameremo queste componenti RS, RI, GS, GI, BS, BI. Ogni componente possiede un range di 2^4 (16) valori. Le intensità di pixel scelti in posizioni casuali sono invertite.
- In ogni blocco 8x8 dell'immagine originale vengono scelti dei pixel in posizioni casuali e il valore d'intensità delle componenti RS, RI, GS, GI, BS, BI viene invertito. Ad esempio, se RS equivale a 10, il suo valore d'intensità verrà invertito e il suo valore diventerà $2^4 - 10 = 6$. Questo processo verrà applicato a tutte le componenti RS, RI, GS, GI, BS, BI dei pixel scelti.
- Si ottiene in output l'immagine cifrata.

Il processo di decifratura è esattamente analogo:

- Viene presa in input l'immagine.
- I pixel vengono permutati nelle loro posizioni originali.
- Vengono ripristinate le intensità dei pixel nei blocchi 8×8 pixel.
- Si ottiene in output l'immagine decifrata.

Nel lavoro da noi svolto, per consentire l'applicazione del watermark ed altre operazioni necessarie al calcolo del watermark stesso, sono stati esclusi i pixel relativi ai DC dell'immagine dal processo di cifratura appena descritto. Per raggiungere ciò sono stati eseguiti i seguenti passi relativi alla cifratura e decifratura.

Per quanto riguarda la **cifratura**:

- L'immagine con i blocchi MCU permutati da cifrare (rappresentata mediante un array tridimensionale di pixel) è stata per prima cosa suddivisa in blocchi di dimensione 8×8 pixel, poi sono stati presi tutti questi blocchi tridimensionali "quadrati" (chiamati blocchi DCT, aventi, quindi, $8 \times 8 = 64$ pixel) e ciascuno di essi è stato trasformato in un blocco tridimensionale "rettangolare" di dimensione 1×64 pixel ed inserito in un array tridimensionale apposito. In questo modo la tabella risultante avrà un numero di righe pari al numero di blocchi DCT dell'immagine ed un numero di colonne pari a 64.
- L'array di blocchi rettangolari appena ottenuto viene cifrato mediante le funzioni disponibili nella libreria python "imageshuffle" precedentemente importata, utilizzando una chiave "key" fornita. Viene così ottenuto un nuovo array contenente i pixel cifrati/shufflati.

```
s = imageShuffle.Rand(key)
encrypt_image_array = s.enc(pixel_array_nocipher)
```

Fig.15. Uno snippet di codice del file di progetto "image_encrypt.py" raffigurante la cifratura di un array data una chiave "key".

Abbiamo, a questo punto, due array: un array di blocchi rettangolari "in chiaro" (fig. 16(a)) ed un altro array rappresentante quest'ultimo in modo totalmente cifrato (fig. 16(b)).

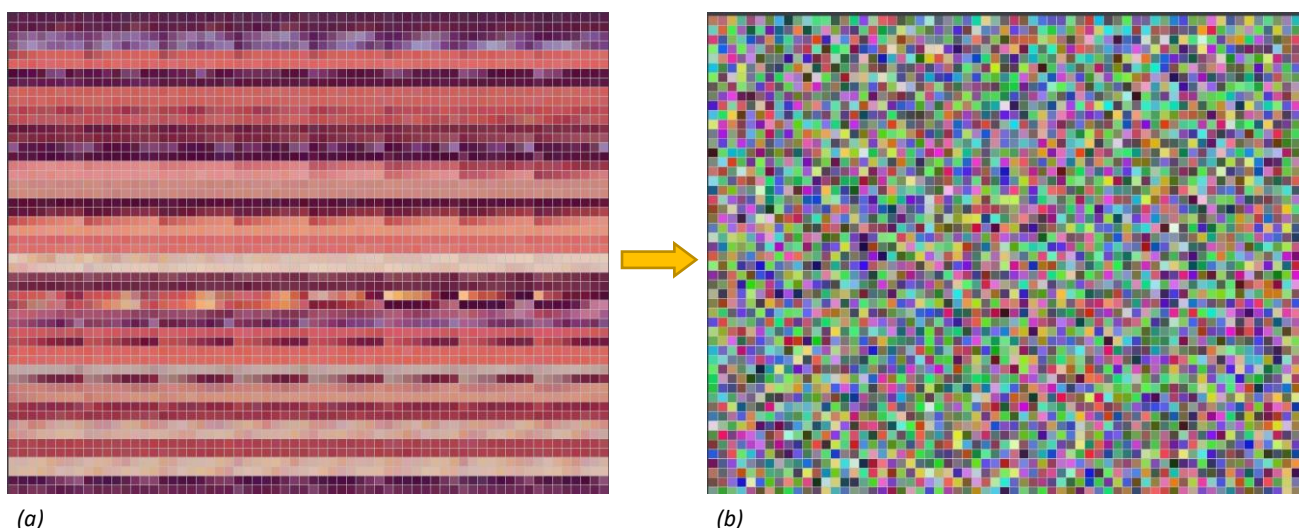


Fig.16. L'immagine (a) a sinistra mostra parte dell'array trasformato in immagine contenente i blocchi rettangolari, mentre l'immagine (b) di sinistra mostra il corrispondente array completamente cifrato.

- Dopodichè, dato che vogliamo avere i pixel corrispondenti ai DC in chiaro, viene effettuato uno swap tra la prima colonna dell'array in chiaro e quella dell'array cifrato (la prima colonna corrisponde proprio ai pixel DC e la colonna cifrata viene memorizzata in un array che servirà in fase di decifratura). Le figure 17 e 18 mostrano chiaramente il processo descritto.

```

#### S W A P ####
#Swapping 3d AC's column of the unmodified image with 3d the AC's column of the encrypted image
swap_array = np.array(encrypt_image_array[:, 0])
encrypt_image_array[:, 0] = pixel_array_nocipher[:, 0]
pixel_array_nocipher[:, 0] = swap_array

```

Fig.17. Uno snippet di codice del file di progetto "image_encrypt.py" raffigurante il processo di swap.

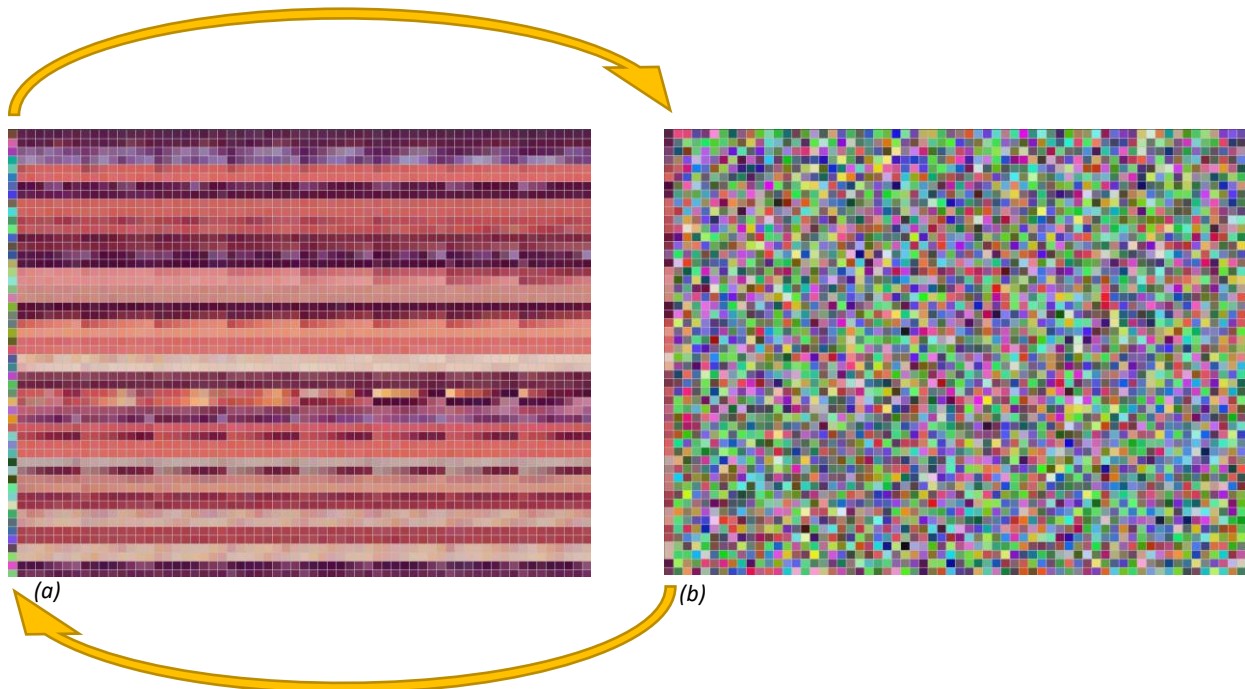


Fig.18. Raffigurazione del processo di swap.

- Dato che vogliamo visualizzare l'immagine in blocchi quadrati con i soli pixel AC cifrati e i pixel DC decifrati, andiamo a ricostruire tali blocchi quadrati di dimensione 8×8 pixel a partire dai blocchi "rettangolari" 1×256 pixel e cioè dalle righe dell'array (raffigurato in figura 18(b)) creato in precedenza contenente i soli pixel DC in chiaro, mentre tutti i restanti pixel sono cifrati. E' stata così ottenuta l'immagine con i soli pixel AC cifrati, mentre i DC sono in chiaro. Il risultato è mostrato nella figura seguente:

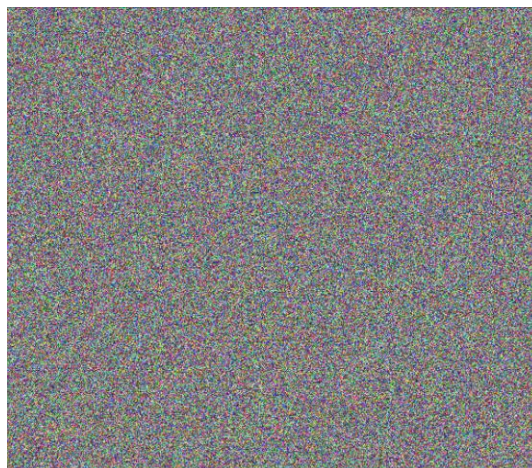


Fig.19. L'immagine cifrata.

E' interessante notare che eseguendo uno zoom profondo sui pixel, sia sull'immagine cifrata che sull'immagine in chiaro con i blocchi permutati, i pixel DC appaiono in chiaro, mentre tutto il resto è cifrato (sovrapponendo le immagini e confrontando i pixel DC nelle stesse posizioni si nota quanto appena detto).

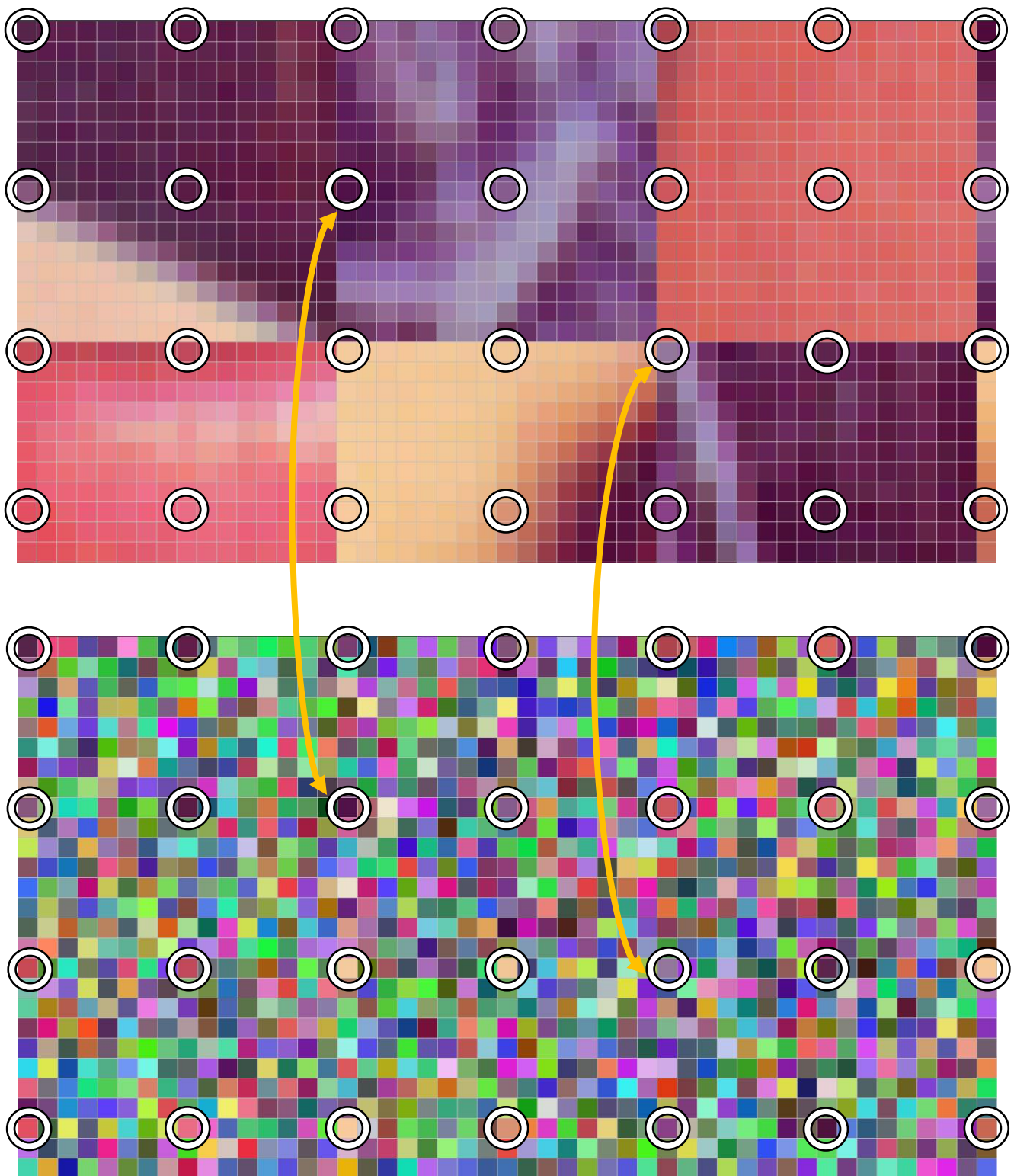


Fig.20. I pixel DC in chiaro su una parte dell'immagine.

Per quanto riguarda la **decifratura**, invece, viene sostanzialmente eseguito il processo inverso (assumiamo che gli array siano tridimensionali); quindi l'immagine cifrata (avente i DC in chiaro e gli AC cifrati) viene suddivisa in blocchi quadrati di dimensione 8×8 pixel e ciascuno di essi viene trasformato in un blocco rettangolare di dimensione 1×64 pixel ed inserito in un array. L'array così ottenuto ha la prima colonna contenente i pixel DC in chiaro. Poi per eseguire la decifratura di questo array, deve essere rimessa al suo posto la colonna avente i pixel DC cifrati, effettuando quindi nuovamente uno swap tra la colonna dei pixel DC in chiaro con la colonna avente i pixel DC cifrati (precedentemente salvati in una variabile array). Una volta fatto ciò avremo tutti i pixel completamente cifrati e a questo punto può essere eseguita la decifratura mediante le funzioni integrate nella libreria python importata "imageshuffle".

```
#### S W A P ####
#Swapping between ACs 3d column of the unmodified image and the same column of the encrypted image
pixel_array_cipher2[:, 0] = swap_pixel_array_nocipher

#Decryption 3d n x 64 table
decrypt_image_array = s.dec(pixel_array_cipher2)
```

Fig.21. Uno snippet di codice del file di progetto "image_encrypt.py" raffigurante il processo di swap e decifratura

Una volta decifrato l'array, ciascuna riga di dimensione 1×64 pixel deve essere riconvertita in un blocco quadrato di dimensione 8×8 pixel ed inserito in un array. Otterremo in questo modo l'immagine con i blocchi MCU permutati completamente in chiaro.

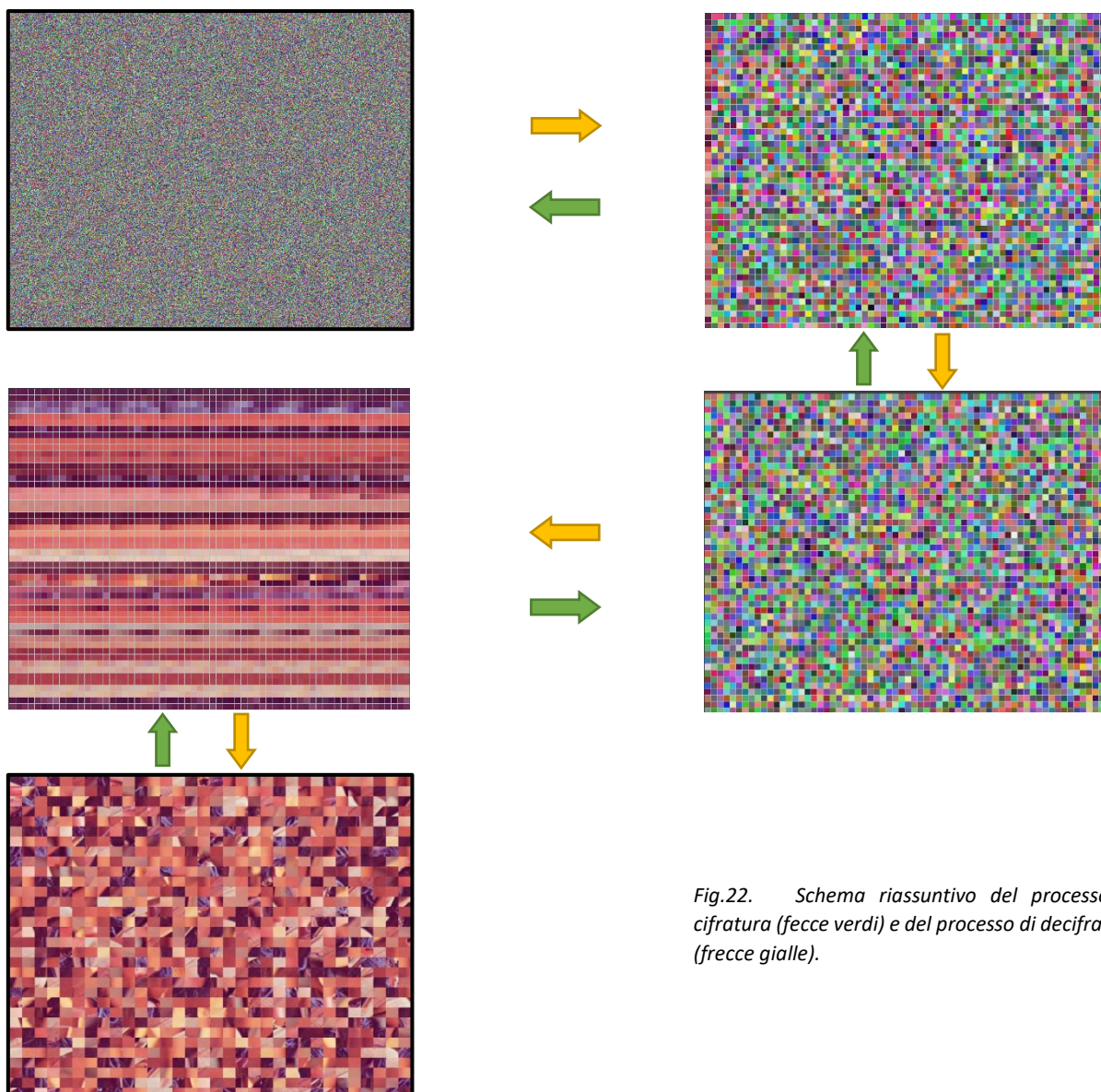


Fig.22. Schema riassuntivo del processo di cifratura (freccie verdi) e del processo di decifratura (freccie gialle).

3.4 Algoritmo di watermarking

Come già detto in precedenza, l'immagine viene convertita da RGB a YCbCr, quindi avremo una componente di luminanza e due componenti di cromaticità. Nel nostro algoritmo di watermark si vanno a modificare solamente le componenti di luminanza, visto che il nostro occhio è più sensibile a quest'ultima che alle componenti di cromaticità. L'idea è quella di introdurre un bit di informazione di watermark nei coefficienti DC quantizzati di una componente di luminanza. L'algoritmo di compressione JPEG divide l'immagine in tanti blocchi DCT 8×8 pixel, dove il primo coefficiente della matrice con coordinate (0,0) sono i coefficienti DC, mentre i restanti 63 coefficienti sono chiamati coefficienti AC. Il nostro algoritmo di watermark è interessato solamente alle componenti DC quantizzate e quindi andremo a salvarle per ogni blocco 8×8 in un file ed andremo ad introdurre un bit di informazione di watermark nelle componenti DC quantizzati attraverso alcune operazioni. Visto che la differenza in valore assoluto tra valori DC di blocchi adiacenti è solitamente piccola, possiamo usare coefficienti DC di blocchi DCT adiacenti per predire il valore DC del blocco DCT corrente. Per garantire che l'immagine possa essere decriptata correttamente dopo l'introduzione del watermark, la grandezza della modifica sui coefficienti quantizzati DC è limitata in un intervallo che va da -3 a 3. Facendo riferimento al paper preso in considerazione, la procedura sviluppata da noi consiste di varie fasi:

1. Nella funzione **compressione** vengono prese le componenti quantizzate dei coefficienti DC di luminanza per ogni blocco DCT 8×8 pixel e vengono salvate nel file di testo "**dc_Y.txt**" per poi passarle alla funzione **watermark**, insieme al file "**watermark_file**" che è il file che viene dato in input che contiene i bit di watermark.
Di seguito vengono riportati alcuni **snippet di codice** relativi a questa fase:

```
# We save the DC for Y component in a file "dc_Y.txt"
if (k == 0):
    dc_Y.write(str(dc[block_index, k]))
    dc_Y.write(" ")
    count += 1
    if (count == (image_width // 8)):
        dc_Y.write('\n')
        count = 0
```

```
##### WATERMARKING #####
dc_mod, block_modified = watermark.watermark("dc_Y.txt", watermark_file)
```

Fig.23. Degli snippet di codice del file di progetto "compression.py".

Nella funzione watermark vengono eseguite tali operazioni:

2. Vengono raggruppati tutti i coefficienti DC quantizzati secondo la loro divisione in blocchi MCU. Ogni blocco MCU contiene quattro coefficienti DC quantizzati dei blocchi DCT adiacenti. Per identificarli, useremo gli indici $i = 0, 1, 2, 3$.
3. Viene calcolato il massimo e il minimo dei quattro valori DC e il valore medio dei tre valori DC. Quindi:

$$A = \frac{DC_1 + DC_2 + DC_3}{3}$$

$$A_{\max} = \max(DC_0, DC_1, DC_2, DC_3)$$

$$A_{\min} = \min(DC_0, DC_1, DC_2, DC_3)$$

4. Vengono scelti due interi T_1 e T_2 che soddisfano la relazione $T_1 < T_2$ per calcolare la grandezza della modifica come:

$$P = \begin{cases} 1, & \text{if } (A_{\max} - A_{\min}) \leq T_1 \\ 2, & \text{else if } T_1 < (A_{\max} - A_{\min}) \leq T_2 \\ 3, & \text{otherwise} \end{cases}$$

5. Viene Incorporato un bit di informazione di watermarking w modificando il coefficiente DC quantizzato.

Quando $w = 1$ viene modificato il DC_0 come segue:

$$DC'_0 = \begin{cases} A + P, & \text{if } DC_0 < A + P \\ \text{no change.} & \text{otherwise} \end{cases}$$

6. Mentre quando $w=0$ viene modificato il valore DC_0 come:

$$DC'_0 = \begin{cases} A - P, & \text{if } DC_0 > A - P \\ \text{no change.} & \text{otherwise} \end{cases}$$

Dove DC'_0 è la modifica candidata di DC_0

7. Se la differenza in valore assoluto tra DC_0 e il nuovo DC'_0 è più grande di 3: Il coefficiente DC non deve essere modificato per aggiungere il watermark, altrimenti DC_0 va rimpiazzato con DC'_0

Il processo seguente va ripetuto per ogni blocco MCU per incorporare il watermark. Inoltre, nel nostro programma ogni volta che viene aggiunto un bit di watermark vengono memorizzati gli indici dei blocchi modificati [riga, colonna] nella variabile ***modified_blocks*** in modo da sapere dove è stato incorporato il watermark.

Di seguito possiamo vedere uno **snippet di codice**:

```
A = (DC11 + DC22 + DC33) / 3
minimum = min([DC00, DC11, DC22, DC33])
maximum = max([DC00, DC11, DC22, DC33])
if ((maximum - minimum) <= T1):
    P = 1
elif (T1 <= (maximum - minimum) < T2):
    P = 2
else:
    P = 3
if (index < len(bitwatermark) and bitwatermark[index] == 1):
    if (DC00 < A + P):
        new_DC0 = A + P
        if (abs(DC00 - new_DC0) <= 3):
            DC0[index_columns] = int(round(new_DC0, 1))
            modified_blocks.append([index_rows, index_columns])
            index += 1
    else:
        continue
elif (index < len(bitwatermark) and bitwatermark[index] == 0):
    if (DC00 > A - P):
        new_DC0 = A - P
        if (abs(DC00 - new_DC0) <= 3):
            DC0[index_columns] = int(round(new_DC0, 1))
            modified_blocks.append([index_rows, index_columns])
            index += 1
```

Fig.24. Uno snippet di codice del file di progetto "watermarking.py".

3.4.1 Estrazione del watermark

Dal lato decompressore, si vuole estrarre i bit di watermark inseriti su un bitstream JPEG cifrato, in quanto su quello non cifrato non siamo in grado di estrarre i bit visto che nel compressore questi bit venivano aggiunti dopo una fase di permutazione. Per prima cosa otteniamo i quattro coefficienti DC quantizzati dei blocchi MCU, denotati con $DC_i = 0,1,2,3$. Si calcola il valore medio degli ultimi tre coefficienti DC quantizzati, il bit estratto nel blocco MCU può essere ottenuto come:

$$\bar{w} = \begin{cases} 1, & \text{if } \tilde{DC}_0 > \tilde{A} \\ 0, & \text{otherwise} \end{cases}$$

← MEDIA DEI DC1,2,3 (7)

Estraendo tutti i bit del watermark dal bitstream JPEG, possiamo ottenere la sequenza di bit contenenti nel "watermark_file".

Di seguito è riportato uno **snippet di codice** della nostra funzione di estrazione:

```

if(modified_blocks[index][0]==index_rows and modified_blocks[index][1]==index_columns):
    if(DC00 > average):
        bitWatermarkExtracted.append(1)
        index += 1
    else:
        bitWatermarkExtracted.append(0)
        index += 1
    if(index == len(modified_blocks)):
        break
if (index == len(modified_blocks)):
    break
print("Watermark extracted: ", bitWatermarkExtracted)

```

Fig.25. Uno snippet di codice del file di progetto "watermarking.py".

3.4.2 Algoritmo di verifica del watermark

L'algoritmo di verifica degli attacchi è utile se si vuole verificare se la nostra foto con watermark ha subito degli attacchi. Esso prende in input la foto con watermark da verificare, salva nel file "*dc_Y_attack.txt*" le componenti quantizzate dei coefficienti DC di luminanza per ogni blocco DCT 8×8 , visto che vengono usati i DC per aggiungere il watermark. Dopo di che avviene l'estrazione del watermark passando alla funzione di estrazione il file dei dc attaccati e l'array che contiene i blocchi con il watermark inserito. Quindi si va a verificare se i bit di watermark estratti coincidono con la foto originale, se la risposta è affermativa significa che la foto non è stata attaccata, in caso di risposta negativa si potrà verificare quanti bit di watermark sono stati modificati e quindi verificare fino a quanto è "robusto" il nostro algoritmo.

```

for i in range(0, rows, 8):
    for j in range(0, cols, 8):
        try:
            block_index += 1
        except NameError:
            block_index = 0

        for k in range(3):
            # split 8x8 block and center the data range on zero
            # [0, 255] --> [-128, 127]
            # "block" is bidimensional array
            block = npmat[i:i + 8, j:j + 8, k] - 128

            dct_matrix = compression.dct_2d(block)
            quant_matrix = compression.quantize(dct_matrix,
                                                'lum' if k == 0 else 'chrom')

            zz = compression.block_to_zigzag(quant_matrix)
            # fills the array with the previously transformed and quantized DCs
            dc[block_index, k] = zz[0]

            # We save the DC for Y component in a file "dc_Y.txt"
            if (k == 0):
                dc_Y.write(str(dc[block_index, k]))
                dc_Y.write(" ")
                count += 1
                if (count == (image_width // 8)):
                    dc_Y.write('\n')
                    count = 0

dc_Y.close()

dc_Y_attack = open("dc_Y_attack.txt", "r")
watermarking_blocks = np.load("watermark_blocks.npy")
wm.extractWatermark(dc_Y_attack, watermarking_blocks)
dc_Y_attack.close()

```

Fig.26. Uno snippet di codice del file di progetto "verify_watermarking.py".

4. Esperimenti ed analisi dei risultati

Per ciò che ne concerne gli esperimenti inerenti all'algoritmo implementato, è stato utilizzato il linguaggio **Python** per lo sviluppo, nell'IDE *Pycharm*, il quale permette di sfruttare tutte le potenzialità di Python stesso. Attraverso la configurazione di un interprete per l'esecuzione dei programmi implementati, è possibile aggiungere e/o rimuovere librerie a piacimento senza alcuna difficoltà, prestandosi con molta flessibilità alle problematiche preposte. Inoltre, si è pensato di separare le due fasi principali del progetto in questione, infatti sono stati creati due file principali:

- *starter_encoder.py*
- *starter_decoder.py*

Affinché questi due file possano funzionare correttamente, occorre che venga impostata una configurazione sull'IDE utilizzato. Infatti attraverso l'impostazione "edit configurations" è possibile fornire dei parametri che verranno catturati dai programmi in questione. Ogni programma ha una sua configurazione personale. Di seguito viene mostrata la configurazione inerente a *starter_encoder.py*:



Fig.27. I parametri in input del codificatore.

Dove è possibile identificare, rispettivamente, il nome dell'immagine che verrà fornita in input al codice che si occupa della compressione (sia se essa sia di piccole misure o analogamente di misure maggiori) e successivamente il nome dell'immagine che verrà creata al termine dell'esecuzione del codice in questione. In questo caso *Image_compressed.png* sarà l'immagine compressa, cifrata e con l'applicazione del watermark, dove quest'ultimo viene fornito attraverso il terzo parametro, ovvero un apposito file *watermark_array.txt*. Segue il parametro rappresentante la chiave di permutazione ed il parametro rappresentate la chiave di cifratura.

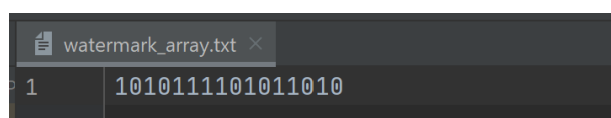


Fig.28. Esempio del contenuto del file "watermark_array.txt" contenente l'array dei bit di watermark.

Viceversa, ma in modalità analoga, è stata stabilita una configurazione per il codice *starter_decoder.py*, dove verrà fornito come primo parametro l'immagine compressa, cifrata e con il watermark; come secondo parametro il nome dell'immagine che verrà creata al termine dell'esecuzione del codice in questione, come terzo parametro l'array dei DC cifrati utilizzato nella fase di cifratura, come quarto parametro l'array degli indici dei blocchi inerenti all'applicazione del watermark. Infine seguono le chiavi di depermutazione e decifrazione, che ovviamente coincidono con quelle inserite nella fase di encoder.

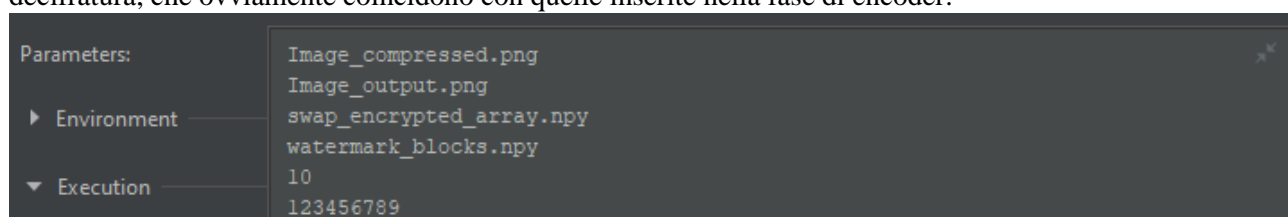


Fig.29. I parametri in input del decodificatore.

La suddetta suddivisione dei due programmi principali ha permesso la sperimentazione delle fasi implementate singolarmente.

```
##### PERMUTATION #####
print("I'm permuting...")
image_permutation = img_permutation.permutation(input_file, 16, key_permutation)

##### ENCRYPTION #####
print("I'm encrypting...")
image_encrypt, swap_array = img_encrypt.encryption(image_permutation, key_cipher)
np.save("swap_encrypted_array.npy", swap_array)

##### WATERMARK AND COMPRESSION #####
print("I'm doing the compression...")
image_compression, blocks_modified = compression.compression(image_encrypt, output_file, watermark_file)
np.save("watermark_blocks.npy", blocks_modified)
```

Fig.30. Snippet di codice del file di progetto "starter_encoder.py"

Per la sperimentazione è necessario che l'immagine caricata sia quadrata di misure, in termini di pixel, multipla di 8 (necessario per la compressione/decompressione) e di 16 (necessario per la permutazione/depermutazione). Se così non fosse il software solleva un'eccezione terminando il programma. Di seguito l'esperimento partendo dalla fase di codifica.

Da come si può notare nella figura 29, vi sono le tre funzioni chiave che sono alla base di *starter_encoder.py*; esse si comportano nel seguente modo:

1. Viene **fornita** l'immagine in input "Image_input.png" su cui voler operare l'algoritmo proposto.



Fig.31. Image_input.png

2. Successivamente vengono **permutati** i blocchi di 16×16 pixel dell'immagine sopra fornita in base al valore di una chiave, secondo la logica implementata e spiegata nel capitolo inerente, dando in output "Image_permutation.png":

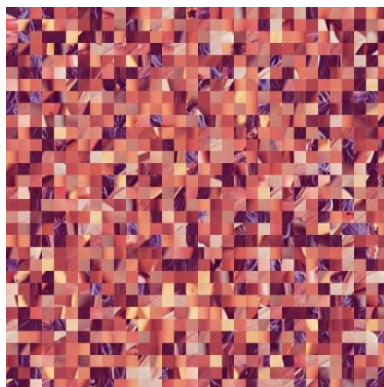


Fig.32. Image_permutation.png

3. A questo punto, avendo l'immagine permutata viene applicata la fase di **cifratura**, la quale attraverso l'uso di una chiave fornisce in output l'immagine con i pixel AC cifrati "Image_AC_encrypt.png":

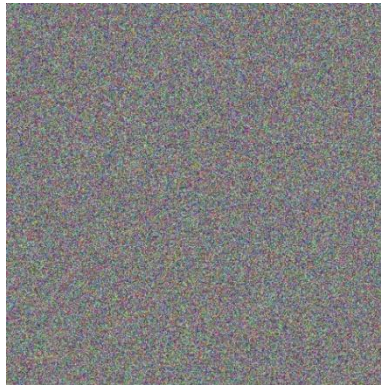
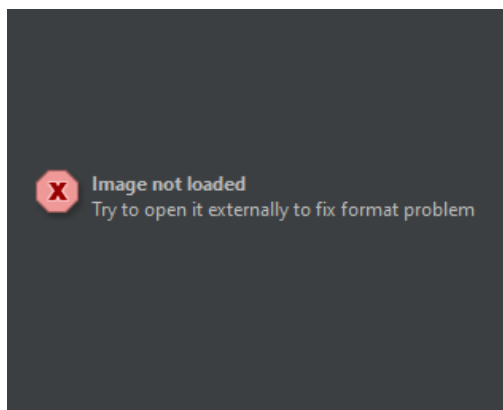
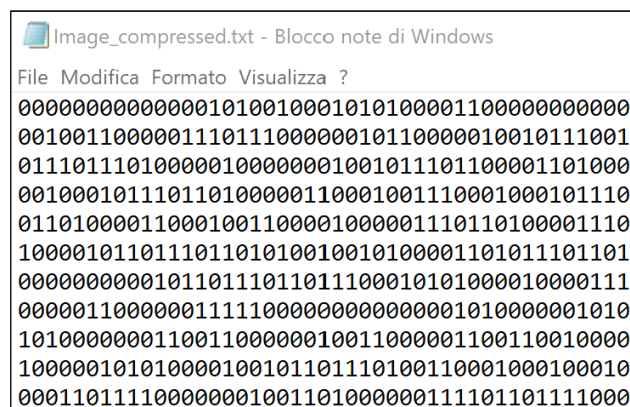


Fig.33. Image_AC_encrypt.png

4. Da come si può notare, si è ottenuta l'immagine originale permutata e cifrata. Essa ora sarà l'input dell'ultimo passaggio della fase di compressione, dove verrà fornita all'**encoder** il quale si occuperà di applicare l'algoritmo JPEG per la compressione, e alla quale verrà applicato il **watermark**. In output si avrà un file compresso, "Image_compressed.png", costituito da caratteri "0" ed "1" testuali. Di conseguenza, l'immagine non è rappresentabile come formato compresso, ma è perfettamente visibile in un formato testuale. *E' doveroso effettuare una precisazione: essendo i caratteri "1" e "0" espressi in forma testuale, il file compresso avrà una taglia maggiore rispetto all'immagine iniziale prima della compressione; per questo motivo è possibile convertire questo file testuale in un file binario, dove i caratteri "1" e "0" diventano bit 1 e 0, di conseguenza la taglia diventerà di circa 8 volte inferiore rispetto al file testuale. Per la conversione possiamo utilizzare i comandi da terminale Linux così come mostrato nella figura 34(c).*



(a) Image_compressed.png



(b) Image_compressed.txt

```
( echo 'obase=16;ibase=2'; sed -Ee 's/[01]{4}/\0/g' binary-text.txt ) | bc | xxd -r -p > instructions.bin
```

(c) Tool di conversione

Fig.34.

Questo segna il termine della sperimentazione della fase di compressione avendo raggiunto il primo punto in esame, ovvero la permutazione, cifratura, compressione e aggiunta del watermark ad un'immagine data in input. Infine come **overhead** vengono forniti in output due file che sono “swap_encrypted_array.npy” e “watermark_blocks”, e che rappresentano, rispettivamente, l'array dei pixel DC cifrati e l'array degli indici dei blocchi inerenti all'applicazione del watermark. Questi ultimi due file serviranno necessariamente alla fase successiva di decodifica.

L'approccio successivo a questa prima fase dell'esperimento (occorre che l'immagine in output fornita dallo *starter_encoder.py*) sarà l'input per l'altro programma principale prima menzionato, ovvero *starter_decoder.py*. Quest'ultimo lavora analogamente ma a procedimento inverso:

```
##### DECOMPRESSION #####
print("I'm doing the decompression...")
image_decompression, dc_Y_mod = decompression.decompression(input_file)

##### DECRYPTION #####
print("I'm decrypting...")
key_cipher = 1234567899
array_encrypt = np.load(array_encrypt_file)
image_decrypt = img_encrypt.deCryption(image_decompression, key_cipher, array_encrypt)

##### EXTRACT WATERMARK #####
print("I'm extracting the watermark...")
watermarking_blocks = np.load(watermarking_blocks_file)
watermark.extractWatermark(dc_Y_mod, watermarking_blocks)

##### DEPERMUTATION #####
print("I'm depermuting...")
key_permutation = 0
image_depermutation = img_permutation.dePermutation(image_decrypt, 16, key_permutation, output_file)
```

Fig.35. Snippet di codice del file di progetto “starter_decoder.py”

In questa seconda parte della sperimentazione, come accennato, parte la fase di decodifica, la quale opera nel seguente modo:

1. Il **decoder** prendendo in input l'immagine permutata, cifrata, compressa e l'array dei DC modificati, esegue l'operazione di decompressione sull'immagine in questione, andando per prima cosa ad effettuare la decompressione, dando in output la seguente immagine "Image_decompressed.png":



Fig.36. Image_decompressed.png

2. L'immagine precedente va ora **decifrata** (vengono decifrati i pixel AC), quindi conoscendo la chiave utilizzata nel processo inverso ed utilizzando il parametro "swap_encrypted_array.npy", viene utilizzata per produrre l'immagine decifrata ("Image_decrypted.png"), la quale corrisponde a:

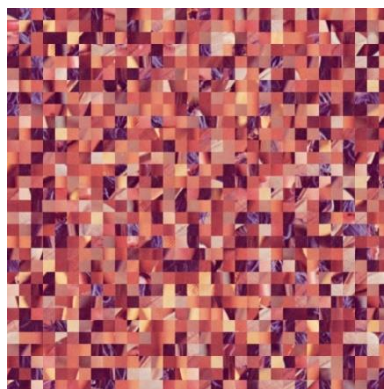


Fig.37. Image_decrypted.png

3. Come si può notare le immagini sono speculari alla fase di compressione, infatti in output è stata restituita l'immagine decompressa e decifrata. Ora non occorre altro che operare la **permutazione inversa** per ricostruire l'immagine originale. Prima di eseguire tale operazione, viene estratto il **watermark** inserito in precedenza utilizzando il parametro "watermark_blocks.npy", andando a costruire "Image_output.png", ovvero quella che era l'immagine originale:



Fig.38. Image_output.png

L'esperimento, a questo punto, è stato portato a termine con ottimi risultati. Le fasi hanno eseguito i compiti preposti riducendo in maniera esponenziale l'overhead dettato dal salvataggio dei vari pixel AC. Inoltre, come accennato la separazione di compressione e decompressione permette di valutare i risultati step-by-step attraverso il salvataggio dei vari valori all'occorrenza. Come hanno mostrato anche le fasi precedentemente descritte e approfondite nei capitoli precedenti, il lavoro ha rispecchiato fedelmente la consegna preposta andando a ridurre anche i costi computazionali sia in termini di complessità di calcolo sia come tempistiche di attesa. Detto ciò si può concludere che il software sviluppato si ritiene molto valido.

4.1 Testing

Per effettuare il testing degli algoritmi da noi implementati è stato preso un dataset di immagini [8] e tre differenti vettori di bit di watermark aventi tre diverse dimensioni: 10-bit, 22-bit e 70-bit. Tali vettori sono contenuti nei rispettivi file “watermark_array1.txt”, “watermark_array2.txt” e “watermark_array3.txt”, così come mostrato nella figura seguente.

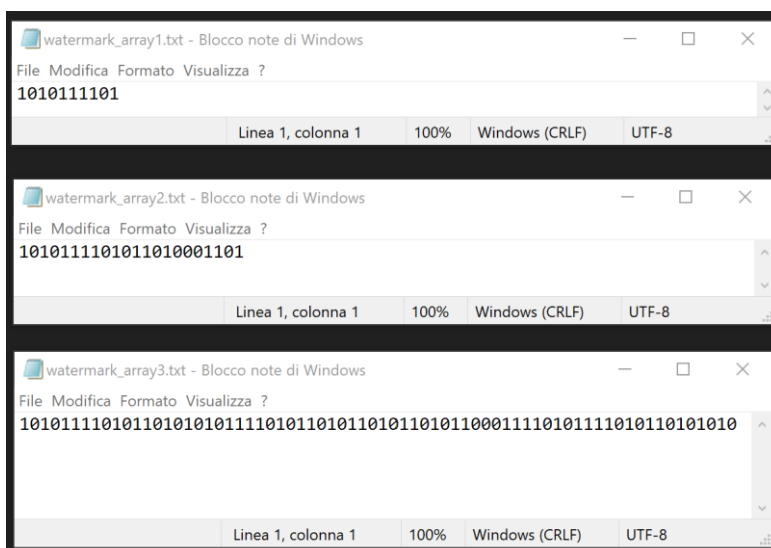


Fig.39. I tre differenti vettori di bit di watermark.

Ad ogni esecuzione è stata data in input una immagine scelta dal dataset in questione ed uno alla volta, ciascuno dei tre vettori di watermark prima citati. E' stato valutato, di esecuzione in esecuzione, se fosse stato possibile applicare il watermark scelto sull'immagine selezionata, oltre a mostrare le immagini delle fasi intermedie quali permutazione, cifratura e compressione. A seguire la tabella dei risultati:


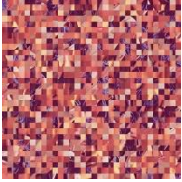
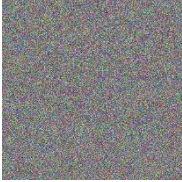

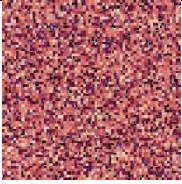


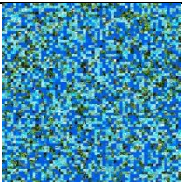
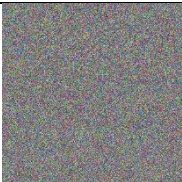




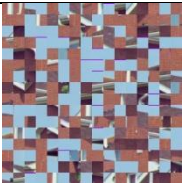
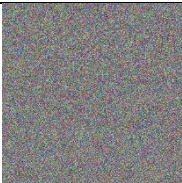
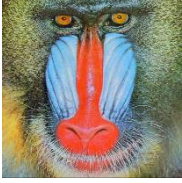



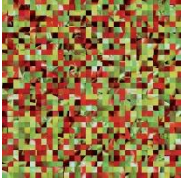

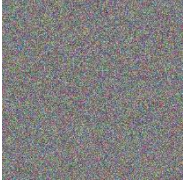
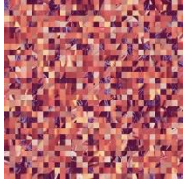

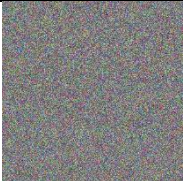
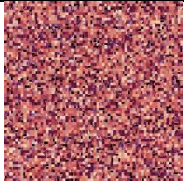

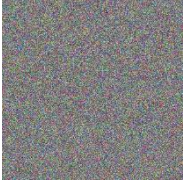
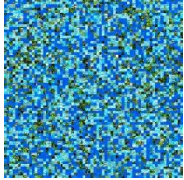

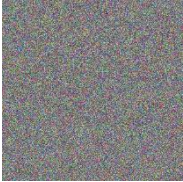


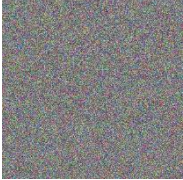
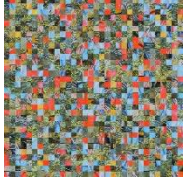
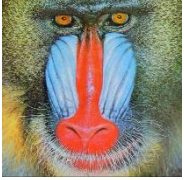
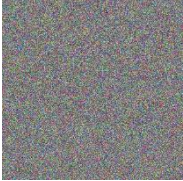
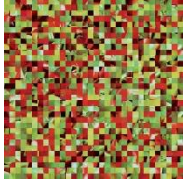

	IMMAGINE INPUT	IMMAGINE PERMUTATA	IMMAGINE CIFRATA	1°= ARRAY DI BIT DI WATERMARK 10-bit	2°= ARRAY DI BIT DI WATERMARK 22-bit	3°= ARRAY DI BIT DI WATERMARK 70-bit
TEST 1	 512 × 512			V	V	X
TEST 2	 1024 × 1024			V	V	V
TEST 3	 1024 × 1024			V	V	V
TEST 4	 256 × 256			V	X	X
TEST 5	 256 × 256			X	X	X
TEST 6	 512 × 512			V	V	X
TEST 7	 512 × 512			V	X	X

IMMAGINE DECOMPRESSA	IMMAGINE DECIFRATA	IMMAGINE OUTPUT
		
		
		
		
X	X	X
		
		

Come possiamo osservare nella tabella, sono stati eseguiti ben 7 test su diverse immagini avente dimensioni differenti in termini di pixel; dai test ne risulta che gli algoritmi di permutazione/depermutazione, cifratura/decifratura e compressione/decompressione hanno avuto esito positivo, mentre l'applicazione del watermark ha avuto due differenti riscontri: inserimento corretto (V) ed inserimento fallito (X). L'inserimento fallito vuol dire che il vettore di bit rappresentante il watermark non è stato possibile incorporarlo nell'immagine in questione poichè i blocchi MCU adiacenti non rispettavano i criteri necessari al suo inserimento. Possiamo notare che nel test 5 nessuno dei vettori di bit di watermark è riuscito ad essere incorporato e di conseguenza non è stato possibile comprimere l'immagine. Nella tabella possiamo osservare tutti i risultati.

5. Conclusioni e sviluppi futuri

In conclusione, il progetto ha raggiunto gli obiettivi prefissati. Infatti, è stato implementato con successo uno schema di watermarking robusto applicabile al contesto della compressione JPEG. Tutte le tecniche descritte dal paper preso in esame sono state implementate suddividendole in appositi moduli così da dare anche la possibilità all'utente di scegliere il grado di sicurezza da impostare (permutazione e cifratura). Inoltre, il tool fornisce la possibilità anche di scegliere diversi parametri in input così da effettuare svariati test come: la key per la cifratura e la permutazione o il numero e il valore dei bit di watermark da inserire.

Un fattore da tenere in considerazione è la size dell'immagine che deve rispettare le dimensioni da noi stabilite ovvero essere multipla di 8×8 e 16×16 , solo in queste condizioni il nostro tool è in grado di lavorare correttamente e di restituire i risultati migliori. La dimensione influisce anche sul numero di bit di watermark da poter inserire, quest'ultimo è un parametro da tenere particolarmente in conto. Di fatti non è stata trovata né sul paper né mediante svariati test una regola da seguire per stabilire il numero di bit massimi da poter inserire, questo perché essi dipendono strettamente non solo dalla risoluzione dell'immagine ma anche dall'intensità di ciascun pixel.

Un possibile sviluppo del tool potrebbe essere quello di prendere in input anche immagini non della corretta size aggiungendo un padding di pixel opportuno per poter far ugualmente lavorare l'algoritmo, chiaramente aggiungere il padding influirebbe sui valori degli AC e DC da prendere in considerazione, quindi una semplice aggiunta non potrebbe bastare ma andrebbero effettuate delle operazioni per mantenere il corretto funzionamento dello strumento.

Un altro sviluppo futuro potrebbe essere quello di convertire i file di appoggio testuali in file di appoggio binari al fine di ridurre la taglia e di mostrare la reale size dell'immagine compressa.

Un ultimo sviluppo potrebbe essere quello di verificare l'algoritmo anche su immagini che hanno subito svariati tipi di attacchi ad esempio: "crop" o "salt and pepper noise attack". Da ipotesi è dato per scontato che l'algoritmo funzioni ugualmente in quanto è in perfettamente in linea con quanto descritto nel paper e nel quale veniva mostrata la sua robustezza ai vari attacchi.

Sitografia

- [1] <https://ieeexplore.ieee.org/document/8887302>
- [2] <http://www.jjsapido.com/iafd/download/jpeg.pdf>
- [3] <https://www.jetbrains.com/pycharm/>
- [4] <https://github.com/ghallak/jpeg-python>
- [5] <https://github.com/s-corso-98/EncDecBit>
- [6] <https://www.bouncycastle.org/>
- [7] <https://github.com/mastnk/imageshuffle>
- [8] <http://sipi.usc.edu/database/database.php?volume=misc&image=39#top>

