UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

"Algoritmo di codifica e decodifica in linguaggio C per il formato di immagine Quite OK"

Relatore: Prof. Antonio Giunta

Laureando/a: Fabio Minorello

ANNO ACCADEMICO 2021 – 2022

Data di laurea 21/09/2022

SOMMARIO

1. Introduzione al formato QOI

Dalla sua pubblicazione nel 1996, il formato di immagine png si è fatto largo come lo standard per la compressione di immagini senza perdite. Lo scopo di una compressione lossless è quello di ridurre la ridondanza presente nell'informazione e codificarla in un minor numero di bit, senza però perdere parte di quell'informazione. Png riesce in questo scopo, ma al costo di un algoritmo di encoding complesso e costoso dal punto di vista computazionale. Negli anni sono comparse alcune ottimizzazioni dell'algoritmo di compressione/decompressione png come TinyPNG, OptiPNG, PNGOUT che hanno ridotto il tempo necessario per la elaborazione e la dimensione dei file compressi generati. Inoltre, sono stati sviluppati formati di immagini lossless alternativi quali Lossless WebP, AVIF, JPEG XL che però attualmente non hanno la stessa copertura di mercato di png.

Tra le alternative, una molto interessante è il formato Quite OK Image Format (.qoi) sviluppato da Dominic Szablewski e presentato a fine 2021. Il progetto si fonda sull'idea di creare un algoritmo di compressione semplice e intuitivo che possa competere con png in determinate casistiche con un fattore di compressione simile ma ottenibile in un tempo ridotto.

2. Specifiche formato QOI

Un file QOI consiste in un header di 14 bytes seguito da una serie di "chunks" di dimensione multipla di 1 byte identificati da un tag iniziale di 2 o 8 bit e concluso da un marker costituito da 7 byte 0x00 e un ultimo byte 0x01.

L'header ha la seguente struttura:

~ magic[4] tag iniziale di 4 bytes che identifica un'immagine in formato qoi

~ 4 bytes per rappresentare la larghezza dell'immagine da comprimere in pixel

~ 4 bytes per rappresentare l'altezza dell'immagine da comprimere in pixel

~ 1 byte per rappresentare il numero dei canali colore dell'immagine.

I canali sono 3 nel caso di immagine RGB (un canale per rappresentare ogni colore).

I canali diventano 4 se è presente anche il canale alfa che viene utilizzato per gestire la trasparenza. Gli 8 bit alfa, infatti, rappresentano 256 livelli di grigio da 0 (bianco) a 255 (nero), quindi considerando solo questo canale otterremo una versione dell'immagine in scala di grigi che funziona come una sorta di stencil. I pixel con canale alfa bianco verranno mostrati

combinati nell'immagine finale; saranno sempre più trasparenti all'aumentare della presenza del nero dentro il canale alfa.

~ 1 byte per rappresentare lo spazio colore:

  ~ 0 → Spazio di colore sRGB con canale alfa lineare

  ~ 1 → Tutti i canali lineari

Lo spazio colore definisce un intervallo fisso e misurabile di possibili colori e valori di luminanza che un dispositivo può catturare (fotocamera) o riprodurre (display); quindi, è un'informazione importante da mantenere all'interno dell'immagine in modo che sia correttamente mostrata sugli schermi o sia correttamente elaborata se importata in un qualsiasi software di modifica foto. Nella codifica qoi, lo spazio colore è a solo scopo informativo, cioè non cambia il modo in cui i chunk sono codificati.

Lo spazio colore più comune per canali a 8 bit è quello sRGB. È uno spazio non lineare; più precisamente, l'intensità di ogni pixel è calcolata applicando una funzione di trasferimento esponenziale dove l'esponente è un parametro, chiamato gamma. L'intensità viene modificata in questo modo per rappresentare più dettagliatamente le zone d'ombra, dove l'occhio umano riesce a cogliere maggiori dettagli, rispetto alle zone di alte luci. Lavorare in uno spazio colore non lineare però implica che le operazioni di addizione e sottrazione tra le intensità dei pixel non diano risultati validi, per questo nell'ambito della modellazione, animazione 3D e nello sviluppo di giochi si utilizzano ancora spazi colore lineari, dove l'intensità del colore viene salvata senza l'applicazione di una curva di gamma, e le operazioni matematiche sono facilitate.

Tornando alla compressione qoi, le immagini vengono codificate scorrendo i pixel riga per riga, da sinistra a destra, e poi da sopra a sotto. Il codificatore e decodificatore salvano il pixel precedente per un confronto con quello attuale, e lo inizializzano ai seguenti valori: rosso, verde e blu a 0 mentre il canale alfa a 255. Un'immagine è completata quando tutti i pixel (width * height) sono stati coperti.

Per la compressione, l'algoritmo utilizza un array di 64 pixel (inizializzati a 0) dove salva i pixel già incontrati nella codifica o decodifica fino a quel momento. I pixel vengono inseriti in una posizione determinata dalla seguente funzione di hash:

$$index\_position = (r * 3 + g * 5 + b * 7 + a * 11) \% 64$$

Se in quella posizione si trova già un pixel, questo viene sovrascritto in modo che sia salvato sempre l'ultimo pixel incontrato corrispondente a quella posizione (funziona come una sorta di cache di pixel).

Dopo questa serie di premesse, ecco nel dettaglio come funziona l'encoder qoi:

Se il pixel incontrato è uguale al precedente, incrementa di uno il counter run dei pixel uguali consecutivi.

Quando si incontra un pixel diverso oppure run vale 62 (valore di run massimo), si codifica il tutto nel chunk QOI_OP_RUN che ha la seguente struttura:

QOI_OP_RUN							
Byte[0]							
7	6	5	4	3	2	1	0
1	1	run					

I primi due bit valgono 1, per identificare il tipo di chunk

Gli altri bit memorizzano il numero di pixel uguali consecutivi incontrati; questo numero viene salvato con un bias di -1, perché altrimenti non utilizzerei mai 00000. Inoltre, non posso salvare numeri di pixel pari a 63 o 64 perché otterrei 11111110 e 11111111 che però sono tag di altri due chunck.

Se il pixel è diverso dal precedente, dopo aver salvato QOI_OP_RUN, se ne calcola la funzione di hash. Se nella cella corrispondente del vettore index trova un pixel uguale, si utilizza il chunk QOI_OP_INDEX che ha la seguente struttura:

QOI_OP_INDEX							
Byte[0]							
7	6	5	4	3	2	1	0
0	0	index					

I primi due bit valgono 0 per identificare il tipo di chunk

Gli altri bit memorizzano l'ìndice del pixel nel vettore index di 64 pixel.

Un buon encoder non utilizza più QOI_OP_INDEX consecutivi, perché in quel caso è consigliabile usare QOI_OP_RUN

Se il pixel che si trova nella cella corrispondente è diverso, allora si confronta il canale alfa del pixel con quello del pixel precedente; se questi due canali sono differenti, allora si codifica il tutto con il chunk QOI_OP_RGBA che salva ogni canale del pixel con un tag davanti di 8 bit.

QOI_OP_RGBA											
Byte[0]								Byte[1]	Byte[2]	Byte[3]	Byte[4]
7	6	5	4	3	2	1	0	7 … 0	7 … 0	7 … 0	7 … 0
1	1	1	1	1	1	1	1	Red	Green	Blue	Alfa

I primi otto bit valgono 1 per identificare il tipo di chunk

8 bit memorizzano il valore del canale rosso

8 bit memorizzano il valore del canale verde

8 bit memorizzano il valore del canale blu

8 bit memorizzano il valore del canale alfa

Se il canale alfa non cambia e se la variazione di rosso, verde e blu con il pixel precedente è compresa tra -3 e 2 allora si codifica il pixel con il chunk QOI_OP_DIFF che ha la seguente struttura:

QOI_OP_DIFF							
Byte[0]							
7	6	5	4	3	2	1	0
0	1	dr		dg		db	

2-bit tag 01 per identificare il tipo di chunk

2 bit per scrivere la variazione del canale rosso rispetto al pixel precedente (-2, …, 1)

2 bit per scrivere la variazione del canale verde rispetto al pixel precedente (-2,…,1)

2 bit per scrivere la variazione del canale blu rispetto al pixel precedente (-2,…,1)

I valori sono salvati come interi senza segno con un bias di 2, per esempio -2 è salvato come 0 mentre 1 come 3. Questo per evitare di gestire numeri negativi.

Dato che i canali sono salvati come interi senza segno, si ha un problema nel rappresentare differenze negative, per questo si usa una wraparound operation, quindi 1 – 2 = -1 risulterà in 255 mentre 255 + 1 risulterà in 0.

Se invece la variazione del canale verde è compresa tra -33 e 32, si calcola la differenza tra la variazione del canale rosso e la variazione del verde vg_r e la differenza tra la variazione del canale blu e la variazione del verde vg_b. Se vg_r e vg_b sono comprese tra -9 e 8 allora il pixel può essere codificato con il chunk QOI_OP_LUMA

<table>
<tr><td colspan="16">QOI_OP_LUMA</td></tr>
<tr><td colspan="8">Byte[0]</td><td colspan="8">Byte[1]</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td colspan="6">Variazione verde vg</td><td colspan="4">vg_r</td><td colspan="4">vg_b</td></tr>
</table>

2-bit tag 10 per identificare il tipo di chunk

6 bit per salvare la variazione del canale verde rispetto al pixel precedente -32, …, 31

4 bit per salvare la differenza tra variazione canale rosso e variazione canale verde -8, …, 7

4 bit per salvare la differenza tra variazione canale rosso e variazione canale verde -8, …,7

La variazione del canale verde serve a indicare la direzione generale di come stanno cambiando i pixel; poi il canale rosso e quello blu hanno una loro ulteriore variazione che però rappresentiamo rispetto a quella del verde.

$$vg\_r = vr - vg = (cur\_px.r - prev\_px.r) - (cur\_px.g - prev\_px.g)$$

$$vg\_b = vb - dg = (cur\_px.b - prev\_px.b) - (cur\_px.g - prev\_px.g)$$

I valori sono salvati come interi senza segno con un bias di 32 per la variazione del canale verde e un bias di 8 per la variazione del canale rosso e blu rispetto al verde. Questo per evitare di gestire numeri negativi.

Anche in questo caso lavoriamo con interi senza segno; quindi, per calcolare la differenza, viene utilizzata una warparound operation.

Il creatore non spiega perché sceglie il verde come canale di riferimento ma ipotizzo sia legato a due fattori: il canale verde in condizioni normali ha la maggior quantità di luce; infatti, per una rapida conversione in bianco e nero (monocanale) di un'immagine, si considera proprio il canale verde. Quindi è sensato utilizzarlo come metro di riferimento per capire se l'intensità dei colori sta variando in una determinata direzione. In secondo luogo, la maggior parte dei sensori fotografici utilizzati per scattare foto hanno gli elementi sensibili distribuiti secondo lo schema Bayer, dove le celle sensibili al verde sono il doppio rispetto a quelle degli altri colori; quindi il canale verde avrà meno rumore rispetto agli altri due canali.

Se invece le variazioni dei 3 canali rispetto al pixel precedente superano tutte gli intervalli nei due punti precedenti, il pixel viene codificato nel chunk QOI_OP_RGB che salva tutti i valori dei canali non alfa del pixel con un tag di 8 bit davanti.

QOI_OP_RGB										
Byte[0]								Byte[1]	Byte[2]	Byte[3]
7	6	5	4	3	2	1	0	7 … 0	7 … 0	7 … 0
1	1	1	1	1	1	1	0	Red	Green	Blue

8 bit tag 11111110 per identificare il tipo di chunk

8 bit per il valore del canale rosso

8 bit per il valore del canale verde

8 bit per il valore del canale blu

3. Dominio applicativo del programma

Il programma in appendice è stato sviluppato in ANSI C, versione standard del linguaggio di programmazione C, ed implementa la codifica e decodifica di immagini nel formato .qoi. Il programma elabora file immagine di tipo .bin contenenti soltanto i pixel grezzi dell'immagine senza alcun tipo di header o compressione pregressa. Le immagini devono essere True Color cioè ogni pixel ha 3 o 4 canali i cui valori sono salvati in 8 bit ciascuno.

I file .bin che ho utilizzato per testare il programma sono stati ricavati decomprimendo le immagini png di test fornite dall'autore con un programma che ho implementato (anch'esso in appendice nel file pngTobin.c) sfruttando la libreria open source stb_image, che è comunemente usata nell'industria dei videogiochi perché permette di gestire più formati di immagine con poche semplici funzioni. La libreria non è compatibile con ANSI C, quindi ho dovuto utilizzare C99 per questo breve programma.

4. Struttura e organizzazione dei file nel programma

La cartella del programma contiene i seguenti file:

- qoi.h, che contiene:
    - La definizione del tipo strutturato qoi_desc, che è parametro sia della funzione di codifica e decodifica e contiene la descrizione dell'immagine da comprimere o decomprimere, in particolare: larghezza (uint32), altezza (uint32), numero di canali (uint8), se i pixel sono Lineare o SRGB (uint8)
    - La definizione del tipo strutturato pixel, che rappresenta un pixel dell'immagine ed ha 4 campi uint8, uno per ogni canale
    - La dichiarazione della funzione di compressione
    - La dichiarazione della funzione di decompressione
- qoi.c, che contiene la definizione delle funzioni di compressione e decompressione. All'inizio del file ci sono due costanti INPUT_BUFFER_SIZE e PRINT_BUFFER_SIZE in cui è possibile settare la dimensione in byte del buffer di input e di output. Più grandi saranno i valori, più la compressione/decompressione sarà veloce, perché il tempo perso per la lettura/scrittura dei file sarà ridotto; questo, però, occupa più RAM. I 2 valori di dimensione dei buffer devono essere multipli di 4.
- qoi_tester.c, implementazione del programma che un utente può utilizzare per comprimere e decomprimere un file, dopo averlo compilato e linkato con il file qoi.o, può esser richiamato da riga di comando scrivendo:

    qoi_tester *percorso_file_da_comprimere*

    Il programma, osservando l'estensione del file, capirà se vogliamo comprimere o decomprimere l'immagine. Se in ingresso c'è un file .qoi, il programma otterrà il file .bin corrispondente. Se invece in ingresso c'è un file .bin, il programma, prima di comprimere il file, chiederà all'utente le dimensioni dell'immagine e se il formato dei pixel è SRGB o Lineare. Attenzione: se in input si inserisce il percorso del file da elaborare, il file risultante si troverà nello stesso percorso.

- qoi_benchmark.c, che confronta le prestazioni di codifica e decodifica dell'algoritmo qoi con quelle di png su un piccolo set di immagini di test. Per png ho scelto come implementazione quella della libreria stbi_image (la stessa utilizzata per generare i file bin) per la sua facilità di utilizzo. Il programma lavora con 2 cartelle:
    - Immagini di test, cartella contenente immagini di test fornite dell'autore in tre formati: qoi, png e bin. Il programma utilizza le immagini per comprimere e decomprimerle e legge la descrizione di ciascuna immagine dall'header del formato png sfruttando la funzione stbi_info della libreria stbi. Le immagini in formato bin contenenti solo i pixel raw senza header sono ottenute decomprimendo i png forniti dall'autore con il programma pngTObin descritto precedentemente.
    - output_bench, cartella dove il programma scrive i file ottenuti codificando/decodificando file png e qoi

  La compressione/decompressione per ognuno dei due algoritmi viene eseguita per ogni immagine 5 volte (il numero di ripetizioni è settabile modificando una costante) sullo stesso file di partenza e se ne calcola la media. Viene salvata la dimensione finale del file compresso e il relativo fattore di compressione. Al termine dell'elaborazione viene restituito il file Benchmark_results.txt contente i risultati del test.

5. Differenze rispetto all'implementazione dell'autore del formato

La libreria per comprimere e decomprimere nella versione dell'autore è disponibile su github e se la si confronta con la mia implementazione si possono notare alcune scelte diverse.

La prima differenza è che l'autore ha scritto la sua libreria in unico file header (header-only); in questo modo, chi intende utilizzare la libreria può semplicemente includere il file qoi.h, che contiene sia dichiarazione che definizione delle funzioni, senza dover prima compilare il file qoi.c. Io ho scelto invece di mantenere il metodo visto in aula di creare sia un file di header .h, contenente la definizione delle strutture e la dichiarazione delle funzioni, e un file .c contente la definizione delle funzioni.

La seconda differenza è che ho voluto migliorare la gestione della memoria utilizzata dall'algoritmo. L'autore ha scelto di caricare l'intera immagine da comprimere/decomprimere dal file prima di elaborarla e mantenere, allo stesso tempo, in memoria anche il risultato della compressione/decompressione. Questa scelta sicuramente ha dei vantaggi:

- aumenta le prestazioni dell'algoritmo perché riduce al minimo il tempo perso per la lettura e scrittura nel file
- potrebbe aiutare nel caso si elaborino le immagini direttamente in memoria senza salvarle in un file.

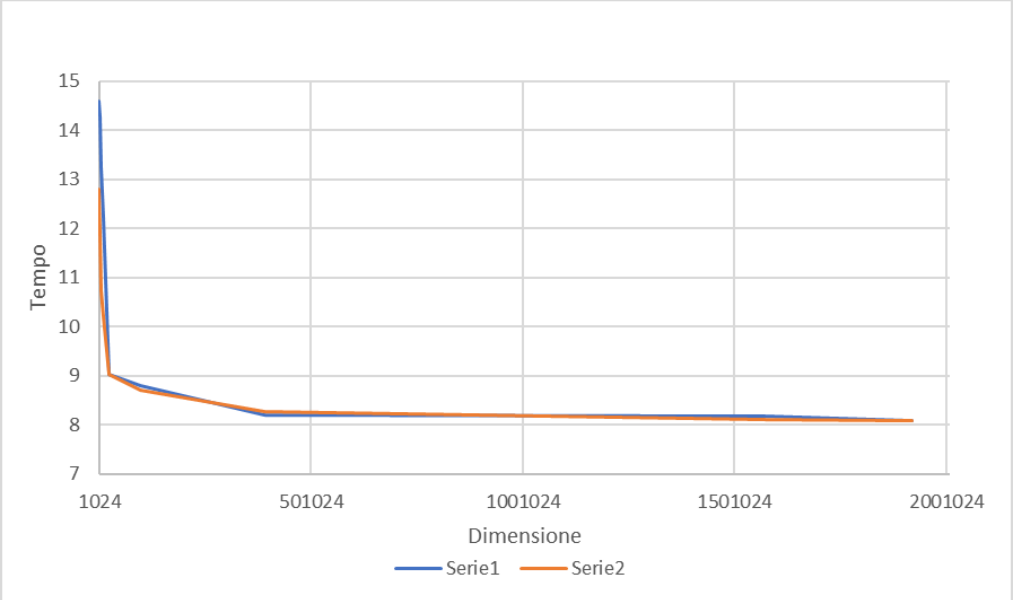
Per migliorare l'utilizzo della memoria, come accennato nel capitolo precedente, ho introdotto due buffer, cioè due vettori di uint8 allocati dinamicamente la cui dimensione può essere scelta dall'utente tramite due costanti: INPUT_BUFFER_SIZE e PRINT_BUFFER_SIZE. Se lavorassimo senza utilizzare RAM dovremmo leggere/scrivere un byte per volta, massimizzando il numero di accessi al file e di conseguenza massimizzando il tempo perso per le operazioni di I/O. Utilizzare un buffer permette di leggere/scrivere un numero di byte pari alle dimensioni del buffer per ogni lettura/scrittura in modo di ridurre il numero di accessi e di conseguenza il tempo perso. Più si

aumenta la dimensione del buffer più aumenteranno le performance del programma; quindi, se è possibile utilizzare molta memoria RAM senza problemi, si possono impostare i buffer alla dimensione dell'immagine più grande da comprimere (per decomprimere è più complesso, bisognerebbe conoscere la dimensione della foto originale) e si possono ottenere le stesse prestazioni dell'algoritmo che legge e scrive tutto il file in una volta.

Le dimensioni dei buffer hanno un solo vincolo: devono essere multiple di 4. È una scelta che deriva dal fatto che nella fase di encoding il programma deve caricare dei pixel che possono essere di 3 o 4 byte a seconda del numero di canali; quindi, una dimensione del buffer non multipla di questi due numeri significherebbe lasciare sempre delle celle libere. Se i pixel non hanno il canale alfa, allora il programma alloca un buffer di dimensione pari al più grande multiplo di 3 minore del numero inserito come INPUT_BUFFER_SIZE.

Osservando queste misurazioni ottenute comprimendo e decomprimendo una delle immagini di test (dice.bin/dice.qoi) presenti nella cartella di immagini fornite dall'autore e variando le dimensioni del buffer, si può osservare quanto sia importante ridurre il numero di accessi al file con buffer più grandi. I dati sono molto approssimativa in quanto i tempi sono molto variabili da una misurazione all'altra ma può dare un'idea dell'incremento prestazionale ottenibile.

Dimensione (bytes)	Encode (ms)	Decode (ms)
1024	14,6	12,8
3072	14,28	11,87
6144	13,24	10,73
12288	12,22	10,2
24576	9,03	9,03
98304	8,81	8,71
393216	8,21	8,27
1572864	8,18	8,12
1920000	8,08	8,09



6. Confronto prestazioni con png stb

Per confrontare le prestazioni del formato qoi con quelle di png, ho utilizzando il programma qoi_benchmark.c descritto nel capitolo 4 compilato ed eseguito su una macchina portatile con le seguenti caratteristiche e impostazioni:

- Intel core i5 7200u 2,5 GHz e TDP max 25 W
- 8 GB di RAM LPDDR4
- Windows 10 settato con profilo energetico a massime prestazioni
- Portatile collegato all'alimentazione per avere le massime prestazioni

In modo da ridurre al minimo l'influenza del tempo speso per lettura e scrittura dal file ho impostato le dimensioni dei buffer di input e di output nel file qoi.c a 3 MB in modo che possano contenere completamente l'immagine più grande del set, cioè wikipedia_008 (1152 x 858 x 3 = 2.965.248 bytes).

I risultati ottenuti sono i seguenti:

Nome file:	dice	**Dimensioni**	800 x 600 x 4	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	9,80	9,05	519653	27,07
PNG	84,18	14,33	349827	18,22
Nome file:	kodim10	**Dimensioni:**	512 x 768 x 3	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	12,70	10,11	652383	55,30
PNG	117,92	16,36	850199	72,07
Nome file:	kodim23	**Dimensioni:**	768 x 512 x 3	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	11,08	9,51	675251	57,24
PNG	110,65	13,16	824064	69,86
Nome file:	qoi_logo	**Dimensioni:**	448 X 220 X 4	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	4,86	4,40	16488	4,18
PNG	11,93	1,51	20210	5,13
Nome file:	testcard	**Dimensioni:**	256 x 256 x 4	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	4,99	4,84	21857	8,34

PNG	8,84	1,26	14227	5,43
Nome file:	testcard_rgba	**Dimensioni:**	256 x 256 x 4	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	5,59	5,42	24167	9,22
PNG	11,92	1,79	18371	7,01
Nome file:	wikipedia_008	**Dimensioni:**	1152 x 858 x 3	
	Encode (ms)	**Decode (ms)**	**Size (bytes)**	**Comp. Rate (%)**
QOI	44,52	22,89	1521134	51,30
PNG	289,37	32,07	1924586	64,90

Calcolando il delta medio tra i risultati ottenuti da png e qoi otteniamo:

Encode (ms)	**Decode (ms)**	**Comp. Rate (%)**
77,32	2,04	4,28

Osservando i dati possiamo trarre le seguenti conclusioni:

- L'algoritmo di compressione qoi in media è circa 6 volte più veloce di quello di png implementato nella libreria stbi ed è più veloce in 7 casi su 7
- L'algoritmo di decompressione qoi invece è in media 1,5 volte più veloce in 4 casi su 7 ma sembra rallentare sulle immagini con un basso numero di pixel.
- Il fattore di compressione di qoi invece è maggiore in 3 casi su 7 ed in ognuno dei casi l'immagine ha pixel a 4 canali. Perde invece parecchio terreno con file contenenti scatti fotografici come kodim23 , kodim10 e wikipedia_008.

L'analisi appena effettuata è molto limitata in quanto:

- Png è stato implementato in varie versioni e probabilmente alcune avranno un encoder più veloce o più efficiente
- il set di sole 7 immagini è molto ridotto (ma è possibile osservare misurazioni più ampie nel sito del creatore del formato)

Tuttavia si possono trarre alcune rapide conclusioni: l'algoritmo ottiene risultati paragonabili a quelli di png nell'implementazione della libreria stbi ma con un algoritmo di codifica più semplice e in generale più veloce. Il formato qoi sembra essere più efficiente nel comprimere immagini grafiche con trasparenza di grandi dimensioni (come le texture nei videogiochi) rispetto a grandi fotografie dove png primeggia.

7. Limiti del lavoro

Un primo limite è il fatto che il programma non riesca a gestire immagini di dimensione superiore ai 2 GB; questo perché nelle specifiche di fseek e ftell è indicato che non possono lavorare con offset per la posizione dei byte così grandi. Esisterebbero dei modi per superare questo limite ma ho scelto di non implementarli dato che è molto raro lavorare con immagini di tali dimensioni.

In secondo luogo, il programma non riconosce se si sta lavorando su una piattaforma che utilizza l'ordinamento big-endian o little-endian. Questo è un problema perché una particolarità del formato qoi è quella di usare l'ordinamento big-endian dei bytes; quindi, per rispettare tale specifica quando si lavora su una macchina Intel (che utilizza invece l'ordinamento little-endian), ho implementato la seguente funzione per riordinare i bytes delle variabili uint32 (da scrivere nell'header dell'immagine):

```
void write_32_bigendian(unsigned char *print_buffer, int *indBuff,
unsigned int w) {
      print_buffer[(*indBuff)++] = (0xff000000 & w) >> 24;
      print_buffer[(*indBuff)++] = (0x00ff0000 & w) >> 16;
      print_buffer[(*indBuff)++] = (0x0000ff00 & w) >> 8;
      print_buffer[(*indBuff)++] = (0x000000ff & w);
}
```

Questa funzione legge i bytes compressi in formato qoi:

```
unsigned int read_32_bigendian(const unsigned char *bytes, int *p) {
      unsigned int a = bytes[(*p)++];
      unsigned int b = bytes[(*p)++];
      unsigned int c = bytes[(*p)++];
      unsigned int d = bytes[(*p)++];
      return a << 24 | b << 16 | c << 8 | d;
}
```

Queste funzioni però riordinano i byte in ogni caso, ignorando la possibilità in cui la macchina su cui si stia lavorando utilizzi già l'ordinamento big-endian. Di per sé non è un grosso limite, dato che le architetture più comuni, come x86 Intel e AMD, utilizzano l'ordinamento little-endian, mentre ARM supporta entrambi.

8. Sitologia

- https://qoiformat.org/
- https://phoboslab.org/log/2021/11/qoi-fast-lossless-image-compression
- https://github.com/phoboslab/qoi
- https://blog.frame.io/2020/02/03/color-spaces-101/
- https://en.wikipedia.org/wiki/Bayer_filter
- https://github.com/nothings/stb
- https://www.ibm.com/docs/en/zvse/6.2?topic=SSB27H_6.2.0/fl2cp_fseek_ftell_text_files_asa_nonasa.html
- https://qoiformat.org/benchmark/
- https://developer.mozilla.org/en-US/docs/Glossary/Endianness?retiredLocale=it

9. Appendice

9.1 pngTObin.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

int main(int args, char **argv) {
    int width, heigth, channels;
    FILE* outF;

    if(args<2) {
        printf("Inserire nome file png da decomprimere senza estensione\n");
        return -1;
    } /* if */

    char nome_file_out[50] = "";
    char nome_file[50] = "";
    strcat(nome_file, argv[1]);
    strcat(nome_file, ".png");

    printf("%s\n", nome_file);

    /*Carico immagine*/
    unsigned char *img = stbi_load(nome_file, &width, &heigth, &channels,
      STBI_rgb_alpha);
    if (img == NULL) {
        printf("Error in loading the image\n");
        return 1;
    }
    printf("Immagine caricata con larghezza %d px, altezza %d px e %d canali\n",
      width, heigth, channels);

    int img_size = width * heigth * channels; /*dimensione dell'immagine*/
    /*printf("%d",img_size);*/

    strcat(nome_file_out, "");
    strcat(nome_file_out, argv[1]);
    strcat(nome_file_out, ".bin");

    printf("%s\n", nome_file_out);

    outF = fopen(nome_file_out, "wb");
    if (outF == NULL) {
        return -2;
    } /* if */

    fwrite(img, 1, img_size, outF); /*sizeof(unsigned char)=1*/

    stbi_image_free(img); /*Libera l'area di memoria in cui ho caricato
      l'immagine*/
}
```

9.2 qoi.h

```
/*  Minorello Fabio - Decoder/Encoder formato di immagine .qoi
      Derivato dal lavoro di Dominic Szablewski */
#include <stdio.h>

/* colorspace */
#define QOI_SRGB   0 /* Solo canale alfa e' lineare */
#define QOI_LINEAR 1 /* Tutti i canali RGBA sono lineari */

/* Struttura contenente informazioni sull'immagine*/
typedef struct {
      unsigned int width; /*Larghezza in pixel*/
      unsigned int height; /*Altezza in pixel*/
      unsigned char channels; /*Numero di canali colore*/
      unsigned char colorspace; /*Spazio colore*/
} qoi_desc;

typedef struct {
      unsigned char r, g, b, a;
} pixel;

int qoi_encode(const char nameFileIn[], const char nameFileOut[], const qoi_desc
*desc);

int qoi_decode(const char nameFileIn[], const char nameFileOut[], qoi_desc
*desc);

int sizeFile(FILE* inF);
```

9.3 qoi.c

```
/*  Minorello Fabio - Decoder/Encoder formato di immagine .qoi
      Derivato dal lavoro di Dominic Szablewski */

#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>#include "qoi.h"

/* DIMENSIONE BUFFER IMPOSTABILE (Deve essere un multiplo di 4)*/
#define PRINT_BUFFER_SIZE 1024 /* Dimensione in byte */
#define INPUT_BUFFER_SIZE 1024 /* Dimensione in byte */

#define N_CH_ALFA 4
#define N_CH_NOALFA 3

#define MAX_RUN 62 /*= 2^6 - 2 */
/*Estremi variazione canali blocco QOI_OP_DIFF*/
#define LOW_DIFF -3
#define HIGH_DIFF 2
/*Estremi variazione canali blocco QOI_OP_LUMA*/
#define VGBR_LOW -9
#define VGBR_HIGH 8
#define VG_LOW -33
#define VG_HIGH 32
/*Offset*/
#define DIFF_OFF 2
#define VG_OFF 32
#define VGB_OFF 8

/* Tag dei possibili blocchi */
#define QOI_OP_INDEX  0x00 /* 00xxxxxx */
#define QOI_OP_DIFF   0x40 /* 01xxxxxx */
#define QOI_OP_LUMA   0x80 /* 10xxxxxx */
#define QOI_OP_RUN    0xc0 /* 11xxxxxx */
#define QOI_OP_RGB    0xfe /* 11111110 */
#define QOI_OP_RGBA   0xff /* 11111111 */

#define QOI_MASK_2    0xc0 /* 11000000 */

#define QOI_HEADER_SIZE 14
#define QOI_PADDING_SIZE 8
#define QOI_MAGIC_SIZE 4
#define QOI_MAX_BLOCK_SIZE 5
#define QOI_INDEX_SIZE 64


/* 2GB e' la dimensione massima gestibile con questa implementazione, in quanto
fseek e ftell non posso lavorare con offset più grandi
Considerando che, nel caso peggiore, l'immagine compressa per ogni pixel occupa
5 bytes, si possono rappresentare un totale di 400 milioni di pixel
che sono sufficienti per chiunque.*/
#define QOI_PIXELS_MAX ((unsigned int)400000000) /*unsigned per coerenza nei
calcoli*/

static const unsigned char qoi_padding[QOI_PADDING_SIZE] =
{0,0,0,0,0,0,0,1};

static const unsigned char qoi_magic[QOI_MAGIC_SIZE] = {'q','o','i','f'};
```

```
/* Dato un pixel calcola l'hash corrispondente

IP a pixel di cui calcolare hash corrispondente
OR risultante della funzione di hash
*/
unsigned int qoi_hash(const pixel* a) {
    return (a->r*3 + a->g*5 + a->b*7 + a->a*11);
} /* qoi_hash */

/* Dato un file ne restituisce la dimensione in bytes
IP inF File di cui calcolare la dimensione
OR size Dimensione del file in bytes
*/
int sizeFile(FILE* inF) {
    int size;
    fseek(inF, 0L, SEEK_END);
    size = ftell(inF);
    fseek(inF, 0L, SEEK_SET);
    return size;
} /* sizeFile */

/*----------------------------------ENCODER--------------------------------*/


/* Inserisce nel buffer di stampa un unsigned int con byte riordinati
   da little a bigendian
IOP print_buffer Array uint8 in cui salvare i byte riordinati di w
IOP indBuff Indice che punta alla prima cella vuota del vettore print_buffer
IP w unsigned int di cui salvare i byte riordinati
*/
void write_32_bigendian(unsigned char *print_buffer, int *indBuff, unsigned int
w) {
      print_buffer[(*indBuff)++] = (0xff000000 & w) >> 24;
      print_buffer[(*indBuff)++] = (0x00ff0000 & w) >> 16;
      print_buffer[(*indBuff)++] = (0x0000ff00 & w) >> 8;
      print_buffer[(*indBuff)++] = (0x000000ff & w);
} /* write_32_bigendian */

/* Scrive nel buffer di stampa l'header del file qoi
IP desc variabile strutturata contenente le info sull'immagine da comprimere
IOP print_buffer print_buffer Array uint8 in cui inserire header qoi
IOP indBuff Indice che punta alla prima cella vuota del vettore print_buffer
*/
void qoi_writeHeader(const qoi_desc *desc, unsigned char* print_buffer, int*
indBuff) {
    int i;
    for (i=0; i<4; i++) {
        print_buffer[i]=qoi_magic[i];
    } /* for */
    (*indBuff) += 4;
    write_32_bigendian(print_buffer, indBuff, desc->width);
    write_32_bigendian(print_buffer, indBuff, desc->height);
    print_buffer[(*indBuff)++] = desc->channels;
      print_buffer[(*indBuff)++] = desc->colorspace;
} /* qoi_writeHeader */

/* Scrive nel buffer di stampa il padding che individua fine del file qoi
IOP print_buffer print_buffer Array uint8 in cui inserire padding qoi
IOP indBuff Indice che punta alla prima cella vuota del vettore print_buffer
*/
void qoi_writePadding(unsigned char* print_buffer, int* indBuff) {
    int i;
    for (i=0; i<8; i++) {
        print_buffer[(*indBuff)+i]=qoi_padding[i];
```

```
    } /* for */
    (*indBuff) += 8;
} /* qoi_writePadding */

/* Determina se due pixel sono uguali oppure no
IP p1 primo pixel da confrontare
IP p2 secondo pixel da confrontare
OR bool true se i due pixel sono uguali, false in caso contrario
*/
bool isPixelEqual(pixel* p1, pixel* p2) {
   return (p1->r == p2->r) && (p1->g == p2->g) && (p1->b == p2->b) &&
            p1->a == p2->a);
} /* isPixelEqual */

/* Inserisce nel buffer per la stampa su file un blocco QOI_OP_RUN
IOP buff_out Array uint8 in cui inserire il blocco QOI_OP_RUN
IOP run Numero di pixel uguali consecutivi incontrati, da azzerare dopo stampa
IOP indBuff Indice che punta alla prima cella vuota del vettore $buff_out
*/
void printQoiRUN(unsigned char* buff_out, int* run, int* indBuff) {
    buff_out[(*indBuff)++] = QOI_OP_RUN | (*run - 1);
    *run = 0;
} /* printQoiRUN */

/* Inserisce nel buffer per la stampa su file un blocco QOI_OP_RGB
IP px pixel contente i valori dei canali da stampare nel blocco QOI_OP_RGB
IOP buff_out Array uint8 in cui inserire il blocco QOI_OP_RGB
IOP indBuff Indice che punta alla prima cella vuota del vettore $buff_out
*/
void printQoiRGB(const pixel* px, unsigned char* buff_out, int* indBuff) {
    buff_out[(*indBuff)++] = QOI_OP_RGB;
      buff_out[(*indBuff)++] = px->r;
      buff_out[(*indBuff)++] = px->g;
      buff_out[(*indBuff)++] = px->b;
} /* printQoiRGB */

/* Inserisce nel buffer per la stampa su file un blocco QOI_OP_RGBA
IP px pixel contente i valori dei canali da stampare nel blocco QOI_OP_RGBA
IOP buff_out Array uint8 in cui inserire il blocco QOI_OP_RGBA
IOP indBuff Indice che punta alla prima cella vuota del vettore $buff_out
*/
void printQoiRGBA(pixel* px, unsigned char* buff_out, int* indBuff) {
    buff_out[(*indBuff)++] = QOI_OP_RGBA;
      buff_out[(*indBuff)++] = px->r;
      buff_out[(*indBuff)++] = px->g;
      buff_out[(*indBuff)++] = px->b;
    buff_out[(*indBuff)++] = px->a;
} /* printQoiRGBA */

/* Comprime i pixel nei blocchi QOI_OP_DIFF o QOI_OP_LUMA
IP px Pixel appena letto dall'immagine da comprimere
IP px_prev Pixel precedente a $px nell'immagine da comprimere
OP buff_out Buffer di stampa in cui scrivere i blocchi compressi
IOP indOut Indice della prima cella libera di $buff_out
*/
void qoiAlfaNoChange(const pixel* px, const pixel* px_prev, unsigned char*
buff_out, int* indOut) {
    /* Uso signed char in modo da maneggiare numeri negativi, altrimenti
      wraparound*/
      signed char vr = px->r - px_prev->r;
      signed char vg = px->g - px_prev->g;
      signed char vb = px->b - px_prev->b;
    /*Calcolo quanto aumenta variazione blu e rosso rispetto a variazione
```

```
      verde*/
      signed char vg_r = vr - vg;
      signed char vg_b = vb - vg;
    /*se variazioni dei canali e' compresa tra -3 e 2*/
      if (vr>LOW_DIFF && vr<HIGH_DIFF &&
          vg>LOW_DIFF && vg<HIGH_DIFF &&
          vb>LOW_DIFF && vb<HIGH_DIFF    ) {
            buff_out[(*indOut)++] = QOI_OP_DIFF | (vr + DIFF_OFF) << 4 |
                                    (vg + DIFF_OFF) << 2 | (vb + DIFF_OFF);
        } /*if*/
    /* se vale -33 < vg < 32 e -9 < vg_r,vg_b < 8 */
    else if ( vg_r>VGBR_LOW && vg_r<VGBR_HIGH &&
              vg > VG_LOW && vg < VG_HIGH &&
              vg_b > VGBR_LOW && vg_b < VGBR_HIGH) {
        /* Differenza rappresentata in 6 bit per il verde con offset di 32
           per rosso e blu calcolata rispetto a verde e rappresentata in 4 bit
           con offset di 8*/
          buff_out[(*indOut)++] = QOI_OP_LUMA | (vg + VG_OFF);
          buff_out[(*indOut)++] = (vg_r + VGB_OFF) << 4 | (vg_b + VGB_OFF);
    } /* else if */
      else
        printQoiRGB(px, buff_out, indOut); /*Salvo intero pixel se differenza
                                             troppo grande*/
} /* qoiAlfaNoChange */

/* Inizializza le variabili run e px_prev ai valori indicati dalle specifiche
IOP px_prev variabile contenente pixel precedente da inizializzare
IOP run Variabile da inizializzare che conta numero di pixel consecutivi uguali
*/
void initPixelsAndRun(pixel* px_prev, int* run) {
    *run = 0;
      px_prev->r = 0;
      px_prev->g = 0;
      px_prev->b = 0;
      px_prev->a = 255;
} /* initPixelsAndRun */

/* Controlla se la descrizione dell'immagine da comprimere fornita e' valida
IP desc Variabile di tipo desc_qoi di cui controllare validità
OR false se la descrizione non e' valida, true se corretta
*/
bool isDescValid(const qoi_desc *desc) {
    if (
            desc == NULL || desc->width == 0 || desc->height == 0 ||
            desc->channels < N_CH_NOALFA || desc->channels > N_CH_ALFA ||
            desc->colorspace > 1 ||
            desc->height >= QOI_PIXELS_MAX / desc->width
      ) {
            printf("Descrizione immagine da comprimere non valida.\n");
            return false;
      } /* if */
    else
        return true;
} /* isDescValid */

/* Alloca dinamicamente un generico buffer e inizializza indice per scorrerlo
IP bufferSize Dimensione buffer da allocare
IOP indBuff Indice che punta alla prima cella libera del buffer
OR buffer Puntatore alla prima cella del buffer appena allocato
*/
unsigned char* allocateBuffer(int bufferSize) {
    unsigned char* buffer = malloc(bufferSize);
    assert(buffer != NULL);
```

```
    return buffer;
} /* allocateBuffer */

/* Alloca dinamicamente il buffer dove caricare bytes da comprimere
OP bufferSize Dimensione finale del buffer allocato
IP channels Numero di canali dei px dell'immagine da leggere da file
OR Puntatore al buffer appena allocato
*/
unsigned char* allocateInputBufferEnc(int* bufferSize, int channels) {
    *bufferSize = INPUT_BUFFER_SIZE;
    if (channels==N_CH_NOALFA)
        *bufferSize = INPUT_BUFFER_SIZE - INPUT_BUFFER_SIZE % 3;
    return allocateBuffer(*bufferSize);
} /* allocateBuffer */

/* Libera buffer output stampandone il contenuto
IOP buffer da stampare su file $outF
IOP indBuff Indice della prima cella libera del buffer
OF outF File in cui stampare contenuto del buffer $buffer
*/
void flushBufferOut(unsigned char* buffer, int* indBuff, FILE* outF) {
    fwrite(buffer, 1, *indBuff, outF);
    *indBuff = 0;
} /* flushBufferOut */

/* Funzione che legge pixel dal buffer di input
IP input_buffer Buffer di input uint8 da cui leggere valori canale
IOP px Pixel in cui inserire valori dei canali appena letti
IOP indIn Indice che punta alla prossima cella da leggere del buffer di input
IP channels Numero di canali dei pixel dell'immagine in input
*/
void readPixel(const unsigned char* input_buffer, pixel* px, int* indIn , int
channels) {
      px->r = input_buffer[(*indIn)++];
      px->g = input_buffer[(*indIn)++];
      px->b = input_buffer[(*indIn)++];
    if (channels == N_CH_ALFA)
            px->a = input_buffer[(*indIn)++];
} /* readPixel */

/* Encoder che comprime un'immagine raw di soli pixel (.bin) in un immagine .qoi
IP nomeFileIn nome del file immagine da comprimere
IP nomeFileOut nome del file contenente l'immagine compressa
IP desc Variabile strutturata contenente informazioni sull'immagine quali
   dimensioni, numero canali, spazio colore
OF File chiamato nomeFileOut contente immagine .qoi compressa
*/
int qoi_encode(const char nameFileIn[], const char nameFileOut[], const qoi_desc
*desc) {
    FILE *inF, *outF;
    unsigned char* input_buffer;
    unsigned char* output_buffer;
    int last_px, px_pos, channels;
    int img_size, run, indOut, bufferInSize;
      pixel px, px_prev;
    pixel* index;

    /* Controllo se descrizione immagine da comprimere valida */
    if (!isDescValid(desc))
        return -4;
    /*Apertura file di output e di input*/
    inF = fopen(nameFileIn, "rb");
    if (inF == NULL) {
```

```
    printf("Errore apertura file input.\n");
    return -1;
} /* if */
outF = fopen(nameFileOut, "wb");
if (outF == NULL) {
    fclose(inF);
    printf("Errore apertura file output.\n");
    return -2;
} /* if */

/* Alloco e inizializzo buffer di input contenente pixel immagine da
  comprimere */
channels = desc->channels;
input_buffer = allocateInputBufferEnc(&bufferInSize, channels);
fread(input_buffer, 1, bufferInSize, inF); /*sizeof(unsigned char)=1*/

output_buffer = allocateBuffer(PRINT_BUFFER_SIZE);
indOut = 0;

qoi_writeHeader(desc, output_buffer, &indOut);

index = calloc(QOI_INDEX_SIZE, sizeof(pixel));
assert(index != NULL);

/*Inizializzo pixel precedente come da specifiche*/
initPixelsAndRun(&px_prev,&run);
  px = px_prev;

img_size = desc->width * desc->height * desc->channels; /*dimensione
                                                          dell'immagine*/
  last_px = img_size - desc->channels;

  for (px_pos = 0; px_pos < img_size; px_pos += channels) {
    int indIn = px_pos % bufferInSize;
    readPixel(input_buffer,&px,&indIn,channels);

    if (isPixelEqual(&px,&px_prev)) {
              run++;
              if (run == MAX_RUN || px_pos == last_px)
            printQoiRUN(output_buffer, &run, &indOut);
        } /* if */
        else {
              int index_pos;
              if (run > 0)
                 printQoiRUN(output_buffer, &run, &indOut);

              index_pos = qoi_hash(&px) % QOI_INDEX_SIZE;

        /*Se trovo il pixel gia' nell'indice blocco index*/
              if(isPixelEqual(&index[index_pos], &px))
                    output_buffer[indOut++] = QOI_OP_INDEX | index_pos;
              else {
                    index[index_pos] = px; /*sovrascrivo il pixel con quello
                                             appena trovato*/
                    if (px.a == px_prev.a)
                          qoiAlfaNoChange(&px, &px_prev, output_buffer,
                                          &indOut);
                    else /*Se alfa cambia*/
                          printQoiRGBA(&px, output_buffer, &indOut);
                    } /* else */
        } /* else */
        px_prev = px;
```

```
        /*Controllo se ho posto per blocco piu' grande, se no stampo e svuoto*/
        if ( (indOut + QOI_MAX_BLOCK_SIZE) > PRINT_BUFFER_SIZE)
            flushBufferOut(output_buffer, &indOut, outF);
        /*Controllo se non ho piu' pixel da leggere da buffer di input*/
        if (indIn == bufferInSize)
            fread(input_buffer, 1, bufferInSize, inF);
    } /* for */
    /*Controllo se ho posto per scrivere padding, se no stampo e svuoto*/
    if (indOut+QOI_PADDING_SIZE > PRINT_BUFFER_SIZE)
        flushBufferOut(output_buffer, &indOut, outF);

    qoi_writePadding(output_buffer, &indOut);
    fwrite(output_buffer, 1, indOut, outF);

    fclose(inF);
    fclose(outF);
    free(input_buffer);
    free(output_buffer);
    printf("Immagine codificata.\n");
    return 0;
} /* qoi_encode */


/*----------------------------------DECODER----------------------------------*/

/* Legge da un vettore di unsigned char (uint8) un unsigned int bigendian
IP bytes vettore di unsigned char
IOP p puntatore con cui avanzare nella lettura del vettore bytes
OR unsigned int letto dal vettore bytes mantenendo ordine bigendian
*/
unsigned int read_32_bigendian(const unsigned char *bytes, int *p) {
      unsigned int a = bytes[(*p)++];
      unsigned int b = bytes[(*p)++];
      unsigned int c = bytes[(*p)++];
      unsigned int d = bytes[(*p)++];
      return a << 24 | b << 16 | c << 8 | d;
} /* read_32_bigendian */

/* Confronta i primi 4 bytes di header per vedere se corrispondono ai primi 4
   bytes qoi_magic che identificano un'immagine compressa in formato qoi
IP header vettore di unsigned char (uint8) header del file immagine da
controllare
OR True se l'header corrisponde a quello di qoi, false altrimenti
*/
bool isQoiHeader(const unsigned char* header) {
    int i;
    for (i = 0 ; i < sizeof(qoi_magic) ; i++) {
        if (header[i] != qoi_magic[i])
            return false;
    } /* for */
    return true;
} /* isQoiHeader */

/* Controlla se la dimensione del file da decomprimere e' troppo piccola
IF inF Immagine da decomprimere
OP size Dimensione file
OR false se dimensione troppo piccola, true altrimenti
*/
bool isSizeValid(FILE* inF, int* size) {
    *size = sizeFile(inF);
    if (*size < QOI_HEADER_SIZE + sizeof(qoi_padding)) {
        printf("Dimensione immagine non conforme.\n");
        return false;
```

```
    } /* if */
    return true;
} /* isSizeValid */

/* Legge la descrizione dal file qoi e ne ricava la descrizione qoi_desc
IP input_buffer Buffer di input da cui leggere i valori da inserire in qoi_desc
IOP indIn Indice della prima cella non letta di $input_buffer
IOP desc Variabile qoi_desc in cui scrivere valori letti dall'header del file
qoi da decomprimere
*/
void readDescQoi(unsigned char* input_buffer, int* indIn, qoi_desc* desc) {
    *indIn = QOI_MAGIC_SIZE; /* Salto i bytes di qoi_magic */
      desc->width = read_32_bigendian(input_buffer, indIn);
      desc->height = read_32_bigendian(input_buffer, indIn);
      desc->channels = input_buffer[(*indIn)++];
      desc->colorspace = input_buffer[(*indIn)++];
} /* readHeader */

/* Funzione che decodifica il blocco QOI_OP_DIFF
IOP px Variabile dove scrivere pixel decodificato
IP c1 Variabile contente il chunck QOI_OP_DIFF da decodificare
*/
void decodeQoiDIFF(pixel* px, int c1) {
    /*0x03 = 00000011*/
      px->r += ((c1 >> 4) & 0x03) - DIFF_OFF;
      px->g += ((c1 >> 2) & 0x03) - DIFF_OFF;
      px->b += ( c1 & 0x03) - DIFF_OFF;
} /* decodeQoiDIFF */

/* Funzione che decodifica il blocco QOI_OP_LUMA, gestisce il caso in cui
   il secondo chunck non sia nel buffer che va aggiornato con nuovi valori
IOP input_buffer Buffer da dove leggere i byte del blocco da decodificare
IOP indIn Indice della prima cella non letta di $input_buffer
IOP px Variabile dove scrivere pixel decodificato
IP c1 Variabile contente il chunck QOI_OP_DIFF da decodificare
IF inF File dell'immagine da cui eventualmente trasferire i byte sul buffer
*/
void decodeQoiLUMA(unsigned char* input_buffer, int* indIn, int c1, pixel* px,
FILE* inF) {
    int c2, vg;
    if (*indIn==INPUT_BUFFER_SIZE) {
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
    } /* if */
      c2 = input_buffer[(*indIn)++];
    /*0x3f = 00111111*/
      vg = (c1 & 0x3f) - VG_OFF;
    /*Altre variazioni su secondo chunck c2, -8 per rimuovere offset*/
      px->r += vg - VGB_OFF + ((c2 >> 4) & 0x0f); /*00001111*/
      px->g += vg;
      px->b += vg - VGB_OFF + (c2 & 0x0f);
} /* decodeQoiLUMA */

/* Funzione che decodifica il blocco QOI_OP_RGB, che potrebbe contenere bytes
che stanno fuori dal buffer e quindi può chiamare una nuova lettura
IOP input_buffer Buffer da dove leggere i byte del blocco da decodificare
IOP indIn indice per scorrere i bytes di $input_buffer
IOP px Pixel in cui scrivere i valori dei canali corrispondenti
IP inF File da cui eventualmente leggere i nuovi bytes
*/
void decodeQoiRGB(unsigned char* input_buffer, int* indIn, pixel* px, FILE* inF)
{
    int diff = INPUT_BUFFER_SIZE-*indIn;
```

```
    if (diff==0) {
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->r = input_buffer[(*indIn)++];
            px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
    } /* if */
    else if (diff==1) {
        px->r = input_buffer[*indIn];
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
    } /* else if */
    else if (diff==2) {
        px->r = input_buffer[(*indIn)++];
        px->g = input_buffer[(*indIn)++];
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->b = input_buffer[(*indIn)++];
    } /* else if */
    else {
        px->r = input_buffer[(*indIn)++];
            px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
    } /* else */
} /*decodeQoiRGB*/

/* Funzione che decodifica il blocco QOI_OP_RGBA, che potrebbe contenere bytes
che stanno fuori dal buffer e quindi può chiamare una nuova lettura
IOP input_buffer Buffer da dove leggere i byte del blocco da decodificare
IOP indIn indice per scorrere i bytes di $input_buffer
IOP px Pixel in cui scrivere i valori dei canali corrispondenti
IP inF File da cui eventualmente leggere i nuovi bytes
*/
void decodeQoiRGBA(unsigned char* input_buffer, int* indIn, pixel* px, FILE*
inF) {
    int diff = INPUT_BUFFER_SIZE-*indIn;
    if (diff==0) {
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->r = input_buffer[(*indIn)++];
            px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
        px->a = input_buffer[(*indIn)++];
    } /* if */
    else if (diff==1) {
        px->r = input_buffer[*indIn];
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
        px->a = input_buffer[(*indIn)++];
    } /* else if */
    else if (diff==2) {
        px->r = input_buffer[(*indIn)++];
        px->g = input_buffer[(*indIn)++];
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->b = input_buffer[(*indIn)++];
        px->a = input_buffer[(*indIn)++];
    } /* else if */
    else if (diff==3) {
```

```
        px->r = input_buffer[(*indIn)++];
        px->g = input_buffer[(*indIn)++];
        px->b = input_buffer[(*indIn)++];
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        *indIn=0;
        px->a = input_buffer[(*indIn)++];
    } /* else if */
    else {
        px->r = input_buffer[(*indIn)++];
            px->g = input_buffer[(*indIn)++];
            px->b = input_buffer[(*indIn)++];
        px->a = input_buffer[(*indIn)++];
    } /* if */
} /*decodeQoiRGBA*/

/* Funzione che inserisce un pixel nel buffer per la stampa su file
IOP buff_out Buffer in cui inserire il pixel
IP px Pixel da inserire nel buffer di stampa $buff_out
IOP indOut Indice della prima cella vuota in $buff_out
IP channels Numero di canali del pixel
*/
void printPixel(unsigned char* buff_out, const pixel* px, int* indOut , int
channels) {
    buff_out[(*indOut)++] = px->r;
      buff_out[(*indOut)++] = px->g;
      buff_out[(*indOut)++] = px->b;
      if (channels == N_CH_ALFA)
                  buff_out[(*indOut)++] = px->a;
} /* printPixel */

/* Decoder che decomprime un'immagine .qoi in un immagine raw di soli pixel
(.bin)
IP nomeFileIn nome del file immagine da decomprimere
IP nomeFileOut nome del file contenente l'immagine decompressa
OF File chiamato nomeFileOut contente immagine decompressa
*/
int qoi_decode(const char nameFileIn[], const char nameFileOut[], qoi_desc
*desc) {
    FILE *inF, *outF;
    int size, indIn, run, indOut;
    unsigned char* input_buffer;
    unsigned char* output_buffer;
    int num_px_img, num_chunks, px_pos;
    pixel px;
    pixel* index;

    /*Apertura file di output e di input*/
    inF = fopen(nameFileIn, "rb");
    if (inF == NULL) {
        printf("Errore apertura file input.\n");
        return -1;
    } /* if */
    outF = fopen(nameFileOut, "wb");
    if (outF == NULL) {
        fclose(inF);
        printf("Errore apertura file output.\n");
        return -2;
    } /* if */

    if(!isSizeValid(inF, &size))
        return -3;

    input_buffer = allocateBuffer(INPUT_BUFFER_SIZE);
```

```
fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF); /*sizeof(unsigned char)=1*/

readDescQoi(input_buffer, &indIn, desc);
if (!isDescValid(desc)) {
    fclose(inF);
    fclose(outF);
    free(input_buffer);
        return -4;
} /* if */

index = calloc(QOI_INDEX_SIZE, sizeof(pixel));
assert(index != NULL);

output_buffer = allocateBuffer(PRINT_BUFFER_SIZE);
indOut=0;

initPixelsAndRun(&px,&run); /*pixel che stampero' nel file decompresso*/

num_px_img = desc->width * desc->height * desc->channels;
  num_chunks = size - sizeof(qoi_padding); /*Numero di chunk da
                                            decodificare*/

  for (px_pos = 0; px_pos < num_px_img; px_pos += desc->channels) {
        if (run > 0) {
              run--; /* Riscrivo px precedente */
        } /* if */
        else if (indIn < num_chunks) {
              int c1 = input_buffer[indIn++];
        /*Prima verifico i tag piu' grandi*/
              if (c1 == QOI_OP_RGB)
            decodeQoiRGB(input_buffer,&indIn,&px,inF);
              else if (c1 == QOI_OP_RGBA)
                    decodeQoiRGBA(input_buffer,&indIn,&px,inF);
              else if ((c1 & QOI_MASK_2) == QOI_OP_INDEX)
            /*QOI_OP_INDEX=00 quindi c1 corrisponde a indice hash*/
                    px = index[c1];
              else if ((c1 & QOI_MASK_2) == QOI_OP_DIFF)
                    decodeQoiDIFF(&px,c1);
              else if ((c1 & QOI_MASK_2) == QOI_OP_LUMA)
            decodeQoiLUMA(input_buffer, &indIn, c1, &px, inF);
              else if ((c1 & QOI_MASK_2) == QOI_OP_RUN)
            /*run = per quante volte ripeto lo stesso pixel*/
                    run = (c1 & 0x3f); /*0x3f = 00111111*/

        /*Aggiorno sempre indice con gli ultimi pixel incontrati,
        sovrascrivendo i vecchi.*/
              index[qoi_hash(&px) % QOI_INDEX_SIZE] = px;
        } /* else if (p < num_chunks) */

    printPixel(output_buffer, &px, &indOut , desc->channels);

    if (indIn==INPUT_BUFFER_SIZE) {
        fread(input_buffer, 1, INPUT_BUFFER_SIZE, inF);
        indIn=0;
    } /* if */

    if ((indOut+N_CH_NOALFA > PRINT_BUFFER_SIZE &&
              desc->channels == N_CH_NOALFA) ||
    (indOut+N_CH_ALFA > PRINT_BUFFER_SIZE && desc->channels == N_CH_ALFA)) {
        fwrite(output_buffer, 1, indOut, outF);
        indOut = 0;
    } /* if */
  } /* for */
```

```
    /*Stampo quello che e' rimasto nel buffer*/
    fwrite(output_buffer, 1, indOut, outF);

    fclose(inF);
    fclose(outF);
    free(input_buffer);
    free(output_buffer);
    printf("Immagine decodificata.\n");
    return 0;
} /* qoi_decode */
```

9.4 qoi_tester.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#include "qoi.h"

/* Dato un file ne restituisce la dimensione in bytes
IP pathFile Percorso del file di cui calcolare la dimensione
OR size Dimensione del file in bytes
*/
int sizeFileIn(const char *pathFile) {
    FILE* f = fopen(pathFile, "rb");
    if (f == NULL) {
        printf("Errore apertura file per controllo dimensione\n");
        return -2;
    } /* if */
    return sizeFile(f);
} /* sizeFileIn */

/* argv[1] nome file da comprimere/decomprimere con estensione */
int main(int args, char **argv) {
    qoi_desc d;
    char* fileNameIn;
    int sizeName;
    char* fileNameOut;

    if(args<2) {
        printf("Inserire nome file con estensione bin o qoi");
        return -1;
    }
    fileNameIn=argv[1];
    sizeName = strlen(argv[1]);

    fileNameOut = malloc(sizeName+1);
    assert(fileNameOut != NULL);

    strncpy(fileNameOut,fileNameIn,sizeName-3); /*copio nome senza estensione*/
    fileNameOut[sizeName-3]='\0';

    if(strcmp(&fileNameIn[sizeName-3],"bin") == 0) {
        char t;
        printf("Per comprimere l'immagine servono altezza e larghezza in
                  px:\n");
        printf("Larghezza: ");
        scanf("%u",&d.width);
        printf("Altezza: ");
```

```
        scanf("%u",&d.height);
        printf("Se lineare scrivi l, se SRGB scrivi s: ");
        fflush(stdin);
        scanf("%c",&t);
        if (t == 's')
            d.colorspace=0x00;
        else
            d.colorspace=0x01;
        d.channels = sizeFileIn(fileNameIn)/(d.width*d.height);
        printf("Dimensioni immagine: %d %d %d\n", d.width, d.height,
                  d.channels);
        strcat(fileNameOut,"qoi");
        qoi_encode(fileNameIn,fileNameOut,&d);
    } /* if */
    else if(strcmp(&fileNameIn[sizeName-3],"qoi") == 0) {
        strcat(fileNameOut,"bin");
        qoi_decode(fileNameIn,fileNameOut,&d);
    } /* else if */
    else
        printf("Formato file non compatibile");

    free(fileNameOut);
    free(argv[1]);

    return 0;
} /* main */
```

9.5 qoi_benchmark

```
/* Minorello Fabio 1216499 */

/* Programma per confrontare prestazioni png e qoi*/

/* Comando per compilare: gcc qoi_benchmark.c qoi.c -std=c99 -O3 -o
qoi_benchmark */

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "qoi.h"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

typedef int (*fCompress)(const char* pathFileIn, const void* pathFileOut,
qoi_desc* desc);

#define NUM_FOTO 7
#define NUM_REP 5

static const char dirInfo[15] = "Immagini_test/";
static const char dirIN[15] = "Immagini_test/";
static const char dirOUT[14] = "output_bench/";

/* Dato un file ne restituisce la dimensione in bytes
IP pathFile Percorso del file di cui calcolare la dimensione
OR size Dimensione del file in bytes
*/
int sizeFileOut(const char *pathFile) {
    FILE* f = fopen(pathFile, "rb");
```

```
    if (f == NULL) {
        printf("Errore apertura file per controllo dimensione\n");
        return -2;
    } /* if */
    return sizeFile(f);
} /* sizeFile */

/* Comprime immagine con algoritmo png sfruttando la libreria stb
IP pathFileIn percorso del file immagine .bin da comprimere
IP pathFileOut percorso dove scrivere il file immagine png compresso
IOP desc variabile qoi_desc dai cui campi leggere la descrizione immagine
OF File png contente immagine compressa salvato in pathFileOut
*/
int stb_png_encode(const char *pathFileIn, const char *pathFileOut, qoi_desc*
desc) {
    FILE* inF;
    unsigned char *data;
    int w = desc->width;
    int h = desc->height;
    int Nch = desc->channels;
    int img_size = w * h * Nch;
    /* Alloco memoria per caricare file bin su ram */
    data = malloc(img_size);
    assert(data != NULL);

    inF = fopen(pathFileIn, "rb");
    if (inF == NULL) {
        printf("Errore apertura file\n");
        return -2;
    } /* if */
    fread(data, 1, img_size, inF); /* Carico file su ram*/
    /* Comprimo in jpg */
    stbi_write_png(pathFileOut, w, h, Nch, data, w * Nch);
    free(data);
} /* stb_png_encode */

/* Decomprime immagine con algoritmo png sfruttando la libreria stb
IP pathFileIn percorso e nome del file immagine png da decomprimere
IP pathFileOut percorso e nome del file bin dove salvare file bin decompresso
IOP desc variabile qoi_desc nei cui campi inserire la descrizione immagine
OF File bin contente immagine decompressa salvato in pathFileOut
*/
int stb_png_decode(const char *pathFileIn, const char *pathFileOut, qoi_desc*
desc) {
    FILE* outF;
    int Nch;
    unsigned char *img = stbi_load(pathFileIn, &desc->width, &desc->height,
                                   &Nch, 0);
    int img_size = desc->width * desc->height * Nch;
    if (img == NULL) {
        printf("Error in loading/decoding the image\n");
        return 1;
    } /* if */

    outF = fopen(pathFileOut, "wb");
    if (outF == NULL) {
        return -2;
    } /* if */

    fwrite(img, 1, img_size, outF);
    stbi_image_free(img);
} /* stb_png_decode */
```

```
/* Funzione che calcola il tempo medio per compressione/decompressione in un
   determinato formato a seconda del parametro funzione $compress passato
IP pathFileIn percorso e nome del file immagine da elaborare
IP pathFileOut percorso e nome del file con cui salvare file ottenuto
IOP desc variabile qoi_desc nei cui campi leggere/scrivere descrizione immagine
IP compress Funzione con cui comprimere/decomprimere l'immagine in ingresso
*/
double benchOneFormat(const char *pathIn, const char *pathOut, qoi_desc* desc,
fCompress compress) {
    LARGE_INTEGER frequency; /* ticks per second */
    LARGE_INTEGER t1, t2; /* ticks */
    double elapsedTime;
    double avgTime=0;
    int i;
    for (i=0; i<NUM_REP; i++) {
        QueryPerformanceFrequency(&frequency); /* get ticks per second */
        QueryPerformanceCounter(&t1); /* start timer */

        compress(pathIn,pathOut,desc);

        QueryPerformanceCounter(&t2); /* stop timer */
        elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
        printf("Time to encode the image: %f ms \n",elapsedTime);
        avgTime += elapsedTime;
    } /* for */
    return avgTime/NUM_REP;
} /* benchOneFormat */

/* Funzione che mette insieme stringhe di cartelle e estensioni per ottenere
   il percorso finale del file desiderato
IOP path Contiene percorso aggiornato
IP dir percorso della cartella dove il file e' inserito
IP name Nome del file che voglio utilizzare nella cartella
IP extension Estensione del file che voglio utilizzare nella cartella
*/
void pathUpdater(char* path, const char* dir, const char* name, const char*
extension) {
    strcpy(path,"");
    strcat(path,dir);
    strcat(path,name);
    strcat(path,extension);
} /* pathUpdater */

/* Comprime e decomprime una stessa immagine sia in formato qoi e png,
cronometrando
   in modo da confrontare i tempi di elaborazione e salvando le dimensioni
finali dei
   file. Stampa i risultati in un file txt
IP fileName Nome dell'immagine da elaborare senza estensione
OF outF File di testo dove salvare i risultati del test
*/
void benchmarkImage(char* fileName, FILE* outF) {
    LARGE_INTEGER frequency; /* ticks per second */
    LARGE_INTEGER t1, t2; /* ticks */
    double avgTime;
    int ch, size, i;
    /*Creo variabile strutturata per descrizione*/
    qoi_desc d;
    d.colorspace=QOI_SRGB;
    char pathIn[40] = "";
    char pathOut[40] = "";

    pathUpdater(pathIn, dirInfo, fileName, ".png");
```

```
    /*Estraiamo informazioni immagine*/
    stbi_info(pathIn, &d.width, &d.height, &ch);
    d.channels=ch;
    printf("Dimensioni immagine: %d %d %d\n", d.width, d.height, d.channels);

    fprintf(outF, "File name: %s con dimensioni %d X %d X %d\n", fileName,
            d.width, d.height, ch);
    fprintf(outF,"    Encode  Decode   Size   Comp.rate\n");

    /*------------------------- QOI ENCODE BENCHMARK -------------------------*/
    pathUpdater(pathIn, dirIN, fileName, ".bin");
    pathUpdater(pathOut, dirOUT, fileName, "1.qoi");
    avgTime=benchOneFormat(pathIn, pathOut, &d, (fCompress)qoi_encode);
    fprintf(outF, "QOI %6.2f", avgTime);
    size=sizeFileOut(pathOut);
    /*------------------------- QOI DECODE BENCHMARK -------------------------*/
    pathUpdater(pathIn, dirInfo, fileName, ".qoi");
    pathUpdater(pathOut, dirOUT, fileName, "1.bin");
    avgTime=benchOneFormat(pathIn, pathOut, &d, (fCompress)qoi_decode);
    fprintf(outF,"  %6.2f", avgTime);
    fprintf(outF,"  %d", size);
    fprintf(outF,"  %5.2f %%\n", (double)
            size*100/(d.width*d.height*d.channels));
    /*------------------------------ PNG ENCODE BENCHMARK --------------------*/
    pathUpdater(pathIn, dirIN, fileName, ".bin");
    pathUpdater(pathOut, dirOUT, fileName, "1.png");
    avgTime=benchOneFormat(pathIn, pathOut, &d, (fCompress)stb_png_encode);
    fprintf(outF,"PNG %6.2f", avgTime);
    size=sizeFileOut(pathOut);
    /*------------------------------ PNG DECODE BENCHMARK --------------------*/
    pathUpdater(pathIn, dirInfo, fileName, ".png");
    pathUpdater(pathOut, dirOUT, fileName, "_png.bin");
    avgTime=benchOneFormat(pathIn, pathOut, &d, (fCompress)stb_png_decode);
    fprintf(outF,"  %6.2f", avgTime);
    fprintf(outF,"  %d", size);
    fprintf(outF,"  %5.2f %%\n", (double)
            size*100/(d.width*d.height*d.channels));
    fprintf(outF,"\n");
} /*benchmarkImage*/

int main(void) {
    int i;

    FILE* outF = fopen("Benchmark_results.txt", "w");
    if (outF == NULL) {
        printf("Errore apertura file per risultati\n");
        return -2;
    } /* if */

    char** fileNames = malloc(NUM_FOTO * sizeof(char*));
    assert(fileNames != NULL);
    fileNames[0] = "dice";
    fileNames[1] = "kodim10";
    fileNames[2] = "kodim23";
    fileNames[3] = "qoi_logo";
    fileNames[4] = "testcard";
    fileNames[5] = "testcard_rgba";
    fileNames[6] = "wikipedia_008";

    for(i=0; i<NUM_FOTO; i++) {
        benchmarkImage(fileNames[i],outF);
    } /* for */
```

```
    printf("Benchmark completato.");

    fclose(outF);

    return 0;
} /* main */
```