Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Metodi del Calcolo Scientifico - Progetto 2

Compressione di immagini tramite la DCT

Alberici Federico - 808058
Bettini Ivo Junior - 806878
Cocca Umberto - 807191
Traversa Silvia - 816435

Anno Accademico 2019 - 2020

Indice

Introduzione **2**

Analisi DCT **3**

1.1 Discrete Cosine Transform 3

1.2 DCT e IDCT 3

1.3 DCT2 e IDCT2 5

1.4 Varianti DCT 8

1.5 Confronto 9

Test con immagini **11**

2.1 Esecuzione del programma 12

2.2 Risultati 13

Introduzione

In questa relazione vengono presentate e discusse le modalità di implementazione della DCT (dall'inglese Discrete Cosine Transform), ovvero la più diffusa funzione che provvede alla compressione spaziale.
Nella prima parte viene confrontata la versione nativa delle DCT implementata in questo progetto con alcune varianti fast (FFT), studiandone il costo computazionale.
Nella seconda parte viene documentato un semplice tool per applicare su immagini in toni di grigio, tramite un approccio di compressione di tipo jpeg (senza utilizzare una matrice di quantizzazione), la funzione DCT2 implemetata.

Analisi DCT

1.1 Discrete Cosine Transform

Una DCT esprime una sequenza finita di punti in termini di una somma di funzioni coseno oscillanti a diverse frequenze. Ad oggi è una delle tecniche di trasformazione più utlizzate nella Teoria dei segnali e nella compressione dei dati, in particolare nei media digitali (audio, video, radio ecc..).
In queste applicazioni infatti la maggior parte delle informazioni significative tendono a essere concentrate in poche componenti a bassa frequenza della DCT. Questo permette di comprimere a piacere il dato scartando le componenti ad alta frequenza (compressione lossy).

1.2 DCT e IDCT

La DCT-II è probabilmente la forma più utilizzata, infatti viene indicata come "la DCT".

$$C_k = \alpha_k \sum_{i=0}^{N-1} V_i \cos\left[\frac{\pi\,(2i+1)\,k}{2N}\right] \quad i = 0, \ldots, N-1 \quad e \quad \alpha_k = \begin{cases} 1/\sqrt{N}, & \text{se } k = 0 \\ \sqrt{2/N}, & \text{se } 1 \leq k \leq N-1 \end{cases}$$

La sua inversa è la DCT-III e per questo viene indicata come "l'inversa della DCT" o "IDCT".

$$V_i = \sum_{k=0}^{N-1} \alpha_k C_k \cos\left[\frac{\pi\,(2i+1)\,k}{2N}\right] \quad k = 0, \ldots, N-1 \quad e \quad \alpha_k = \begin{cases} 1/\sqrt{N}, & \text{se } k = 0 \\ \sqrt{2/N}, & \text{se } 1 \leq k \leq N-1 \end{cases}$$

Entrambe le funzioni effettuano N somme per calcolare la k-esima componente di un vettore di N componenti, determinando un costo computazionale $O(N^2)$.

Implementazione

Per l'implementazione è stato utilizzato C++, sfruttando la libreria open-source Eigen (`https://eigen.tuxfamily.org/`) per semplificare la gestione dei dati.

```
void DCT2::DCT(Eigen::VectorXd &_v)
{

  const Eigen::VectorXd _v_copy = _v;
  const int N = _v.size();
  double ak = 1.0 / sqrt(N);
  double ck = 0;

  for (int k = 0; k < N; k++)
  {
    ck = 0;
    for (int i = 0; i < N; i++)
    {
      ck += cos((2.0 * i + 1.0) * k * M_PI / (2.0 * N)) * _v_copy(i);
    }
    _v(k) = ak * ck;
    if (k == 0)
    {
      ak = sqrt(2.0) / sqrt(N);
    }
  }
}
```

Listato 1.1: Funzione di calcolo DCT

```
void DCT2::IDCT(Eigen::VectorXd &_c)
{
    const Eigen::VectorXd _c_copy = _c;
    const int N = _c.size();
    double ak = 0;
    double vi = 0;

    for (int i = 0; i < N; i++)
    {
        vi = 0;
        ak = 1.0 / sqrt(N);
        for (int k = 0; k < N; k++)
        {
            vi += cos((2.0 * i + 1.0) * k * M_PI / (2.0 * N)) * _c_copy(k) * ak;
            if (k == 0)
            {
                ak = sqrt(2.0) / sqrt(N);
            }
        }
        _c(i) = vi;
    }
}
```

Listato 1.2: Funzione di calcolo IDCT

1.3 DCT2 e IDCT2

La DCT2 è una trasformazione a due dimensioni, ottenuta semplicemente applicando la DCT mono-dimensionale prima per righe e poi per colonne (o viceversa).

La definizione della DCT bi-dimensionale per una matrice A di dimensione m x n in input è:

$$C_{kl} = \alpha_k \alpha_l \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij} \cos\left[\frac{\pi\,(2i+1)\,k}{2m}\right] \cos\left[\frac{\pi\,(2j+1)\,l}{2n}\right],$$

$$con \quad 0 \leq k \leq m-1, \quad 0 \leq l \leq n-1,$$

$$\alpha_k = \begin{cases} 1/\sqrt{m}, & \text{se } i = 0 \\ \sqrt{2/m}, & \text{se } 1 \leq i \leq m-1 \end{cases} \quad e \quad \alpha_l = \begin{cases} 1/\sqrt{n}, & \text{se } j = 0 \\ \sqrt{2/n}, & \text{se } 1 \leq j \leq n-1 \end{cases}$$

L'inversa di tale trasformazione è la IDCT2, ottenuta applicando IDCT alle due dimensioni.

Implementazione

Anche in questo caso Eigen è stato utilizzato per mantenere la struttura dati tramite un oggetto *Eigen::MatrixXd*. L'implementazione non sfrutta direttamente la definizione, ma computa la DCT2/IDCT2 prima sulle righe e poi sulle colonne della matrice in input. Inoltre, essendo l'elaborazione di ogni vettore indipendente dalle altre, è possibile parallelizzarne la computazione. La direttiva *# pragma omp parallel for* prima dei due cicli abilita la computazione concorrente della DCT sulle due dimensioni della matrice in input.
Dunque tale calcolo comporta l'esecuzione di una routine DCT su N righe e M colonne di una matrice, determinando un costo computazionale di:

$$\begin{cases} O(N^2)O(N+M) = O(N^3) & \textit{se DCT di tipo II o III} \\ O(N \log N)O(N+M) = O(N^2 \log N) & \textit{se DCT versione fast (FCT)} \end{cases}$$

```
Eigen::MatrixXd DCT2::DCT2_mt(Eigen::MatrixXd &_m)
{
  Eigen::MatrixXd out = _m;

// DCT su righe
#pragma omp parallel for
  for (int i = 0; i < out.rows(); i++)
  {
    Eigen::VectorXd row = out.row(i);
    DCT(row);
    out.row(i) = row;
  }

// DCT su colonne
```

```
#pragma omp parallel for
  for (int i = 0; i < out.cols(); i++)
  {
    Eigen::VectorXd col = out.col(i);
    DCT(col);
    out.col(i) = col;
  }

  return out;
}
```

Listato 1.3: Funzione di calcolo DCT2

```
Eigen::MatrixXd DCT2::IDCT2_mt(Eigen::MatrixXd &_m)
{
  Eigen::MatrixXd out = _m;

// IDCT su righe
#pragma omp parallel for
  for (int i = 0; i < out.rows(); i++)
  {
    Eigen::VectorXd row = out.row(i);
    IDCT(row);
    out.row(i) = row;
  }

// IDCT su colonne
#pragma omp parallel for
  for (int i = 0; i < out.cols(); i++)
  {
    Eigen::VectorXd col = out.col(i);
    IDCT(col);
    out.col(i) = col;
  }

  return out;
}
```

Listato 1.4: Funzione di calcolo IDCT2

1.4 Varianti DCT

Esistono diverse varianti della DCT che riducono la complessita a $O(NlogN)$. Tali metodi sono conosciuti come *fast DCT* o *FCT* in quanto appunto migliorano notevolmente il costo computazionale.
Di seguito vengono citate due delle più comuni.

Fast DCT di Lee

Descritta da Byeong Gi Lee [1] nel 1984 è uno degli algoritmi fast DCT per 2^m punti più comune. Utilizza una struttura ricorsiva dove la trasformazione DCT è decomposta in una parte pari e una dispari. Queste parti sono a loro volta decomposte nello stesso modo finchè non sono abbastanza piccole (m=1) da essere calcolate tramite valutazione diretta [3].

Fast DCT FFT

Invece di applicare direttamente la formula DCT (o scomporla come mostrato da Lee) è possibile fattorizzare la computazione in modo simile alla *fast Fourier transform* (FFT). Gli algoritmi basati sull'*algoritmo di Cooley-Tukey* [2] per 2^m punti sono i piu comuni, ma qualunque altro algoritmo FFT è applicabile. Per esempio l'*algoritmo FFT di Bluestein* (o *algoritmo chirp-z*) permette di computare per una qualunque dimensione.

Gli algoritmi fast utilizzati nei confronti e test successivi provengono dal Project Nayuki [4], modificati in modo da essere compatibili con gli oggetti Eigen. La versione FFT, in base alla dimensione in input, utilizza i due algoritmi sopra citati.
Come riportato dall'autore "non è difficile utilizzare una FFT per computare una fast DCT che elabori in tempo $O(N \log N)$, ma tale approccio introduce un piccolo fattore costante di penalità in tempo e spazio per memorizzare e computare numeri ridondanti".
Gli algoritmi sono reperibili con una serie di Unit test che ne provano il corretto funzionamento.

1.5 Confronto

I dati utilizzati per generare i grafici seguenti sono stati raccolti applicando i vari algoritmi DCT2 su matrici random di dimensione fissata. Per ogni dimensione il tempo risultante è una media risultante da X esecuzioni degli algoritmi, eseguiti finchè il tempo medio calcolato non si discostasse piu da una certa soglia *dist* per un determinato numero *round* di iterazioni.

Dimensioni potenze di 2

Figura 1.1: *Confronto DCT2 custom, Fast DCT di Lee e Fast DCT FFT*

Vincoli			
Dimensione matrice	dist [ms]	round	time_limit [s]
8 - 256	100	5	3600
512 - 2048	500	5	3600
4096	500	2	3600

Per poter rappresentare graficamente il confronto fra l'algoritmo di DCT2 da noi implementato (*Custom*), e la versione DCT2 dell'algoritmo fast della libreria Lee (*Lee*) e l'algoritmo che sfrutta la fast Fourier transform (*Fast*) è stato utilizzato un grafico a linee

con indicatori, ponendo sull'asse delle ascisse la dimensioni delle matrice quadrate (sono state utilizzate dimensioni in potenza di 2 per poter eseguire il codice di Lee) e sulle ordinate il tempo impiegato, espresso in scala logaritmica.
Si può notare chiaramente che la crescita del tempo impiegato dall'algoritmo Custom è esponenziale rispetto all'andamento similare che hanno gli algoritmi Fast e Lee.

Altre dimensioni

Valori fissi $dist = 100ms$, $round = 5$ e $time_limit = 1800s$.

Figura 1.2: *Confronto DCT2 custom e Fast DCT*

Tale grafico dimostra che, anche senza il vincolo imposto dall'algoritmo di Lee sulla dimensione della matrice, i valori ottenuti con la *DCT2 custom* e la *fast DCT* mantengono lo stesso andamento con matrici di dimensioni incrementali.

Test con immagini

Nella seconda parte del progetto l'interfaccia grafica è stata scritta tramite QT, una libreria multipiattaforma per lo sviluppo di programmi che utilizzano un'interfaccia (attraverso l'uso di widget) che si basa sul linguaggio C++ (motivo per il quale abbiamo deciso di utilizzarla).
Il programma utilizza le seguenti classi da noi implementate:

- **main.cpp**, la quale si occupa di eseguire l'intero corpo del programma;
- **Compress.cpp**, che data una matrice x crea iterativamente delle sottomatrici. Queste saranno poi passate alla libreria *Fast DCT* per eseguire la compressione e ottenere come risultato finale l'immagine compressa. Inoltre, abbiamo aggiunto la possibilità di eseguire questa operazione anche attraverso la *libreria di Lee* e la *custom DCT2*.
- **mainwindow.cpp**, classe che gestisce tutti gli aspetti e i trigger della interfaccia grafica;
- **my_qlabel.cpp**, classe di supporto che permette di visualizzare graficamente la quantità di dettaglio perso dell'immagine tenedo premuto sulla figura. La scelta di implementare questa classe è data dal fatto che con determinati parametri (come si potrà notare nella sezione Risultati) non è possibile notare a occhio nudo la differenza del risultato ottenuto rispetto all'immagine originale.

2.1 Esecuzione del programma

Una volta avviato il programma, è possibile caricare l'immagine .bmp dal filesystem attraverso un apposito tasto ed inserire i valori di F e d. Una volta inseriti i parametri, tramite il pulsante *Process* viene chiamato il metodo *on_parameters_clicked()*, che dopo aver controllato se F e d rispettano tutti i vincoli (F positivo, d compreso fra 0 e 2F - 2), trasforma con la funzione *pixmapToMatrix()* l'immagine in una matrice e la invia alla classe *Compress.cpp*. In essa l'immagine viene divisa in blocchi e a ognuno di questi viene applicata la DCT2, restituendo poi una matrice che viene passata alla funzione *matrixToPixmap()* per poter essere visualizzata in output nell'interfaccia grafica.
In particolare la classe *Compress.cpp* prende in input una matrice di interi e i parametri F e d e restituisce in output una nuova matrice di interi. Al suo interno, la matrice di input viene trasformata in formato double e viene eseguito un troncamento per poter scartare gli "avanzi". Iterativamente, per ogni blocco quadrato F x F applichiamo la DCT2 e poi restituiamo la matrice nel formato int (arrottondando i valori double all'intero più vicino, mettendo a 0 i valori negativi e a 255 quelli maggiori di 255).

Figura 2.3: *Finestra di caricamento immagine da filesystem*

2.2 Risultati

Il programma è stato testato sulle immagini di prova fornite e sono stati sperimentati diversi valori dei parametri F e d, utilizzando la libreria *Fast DCT*.

L'immagine *bridge.bmp* ha dimensione 2749 x 4049, impostando come parametri $F = 200$ e $d = 100$ non è visibile alcuna differenza nell'immagine compressa.

Figura 2.4: *bridge.bmp con $F = 200$ e $d = 100$*

Tenendo fisso il parametro F e diminuendo fortemente il valore del parametro d, portandolo a 7, eliminando un gran numero di frequenze, sono visibili gli artefatti legati alla perdita di dettarglio.

Figura 2.5: *bridge.bmp con $F = 200$ e $d = 7$*

Con l'immagine *deer.bmp*, di dimensione 4043x2641 abbiamo voluto sperimentare con un valore di F maggiore, pari a 250, e con valori di d bassi. Impostando $d = 50$ l'immagine risulta lievemente sgranata.

Figura 2.6: *deer.bmp con $F = 250$ e $d = 50$*

Abbassando il valore di d a 9 sono visibili gli artefatti causa perdita di dettaglio.

Figura 2.7: *deer.bmp con* $F = 250$ *e* $d = 9$

Per l'immagine *cathedral.bmp* (2000x3008) abbiamo voluto testare valori di F piccoli con valori di d relativamente vicini. Ponendo $F = 20$ e $d = 15$ otteniamo un'immagine che non sembra differire dall'originale se non per una luminosità maggiore.

Figura 2.8: *cathedral.bmp con $F = 20$ e $d = 15$*

Abbiamo successivamente impostato $F = 10$ ed abbiamo assegnato a d un valore maggiore, 17. L'immagine risulta pressoché identica.

Figura 2.9: *cathedral.bmp con $F = 10$ e $d = 17$*

Abbiamo testato i casi limite su un'immagine scaricata da noi. Abbiamo assegnato a F un valore casuale (in queso caso 10) e a d il valore 0. In questo modo abbiamo dimostrato che vengono eliminate tutte le frequenze.

Figura 2.10: *bridge2.bmp con $F = 10$ e $d = 0$*

In questo secondo caso, ponendo d = 2F - 2 (in questo caso 18) viene eliminata solo la frequenza più alta.

Figura 2.11: *bridge2.bmp con $F = 10$ e $d = 18$*

Bibliography

[1] Byeong Lee. "A new algorithm to compute the discrete cosine Transform". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32.6 (1984), pp. 1243–1245.

[2] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2003354.

[3] KRISTER LAGERSTRÖM. "Design and Implementation of an MPEG-1 Layer III Audio Decoder". In: 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.3056&rep=rep1&type=pdf#page=34.

[4] Nayuki. *Fast discrete cosine transform algorithms*. Version 0.20.2. Sept. 23, 2018. URL: https://www.nayuki.io/page/fast-discrete-cosine-transform-algorithms.