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

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.

```cpp
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 dagli altri è possibile parallelizzarne la computazione. Per procedere con il calcolo in parallelo abbiamo aggiunto la direttiva *# pragma omp parallel* prima del for così da computare la DCT2 sulle sottomatrici quadrate in maniera concorrente.

```
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 [**Lee**] 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 [**LAGERSTRM2001DesignAI**].

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 [**10.2307/2003354**] sono i piu comuni, ma qualunque altro algoritmo FFT è applicabile.

1.5 Confronto

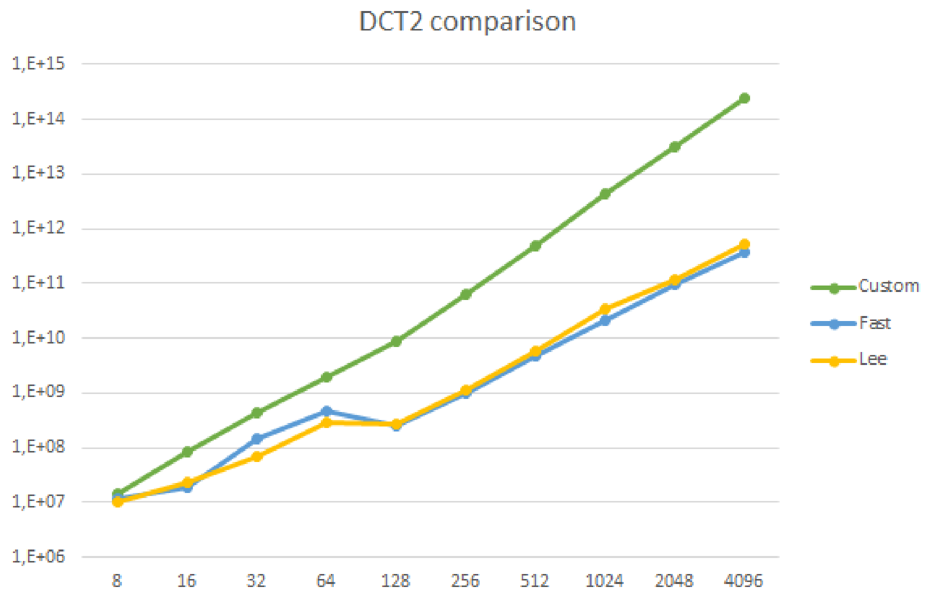

Figure 1.1: *Confronto DCT2 custom, Fast DCT di Lee e Fast DCT FFT*

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.

Test con immagini

Il programma è stato scritto tramite QT, una libreria multipiattaforma per lo sviluppo di programmi che utilizzano un'interfaccia grafica (attraverso l'uso di widget) che utlizza il linguaggio C++ (motivo per il quale abbiamo deciso di utilizzarlo).
Nella seconda parte del progetto, il programma utilizza le seguenti classi scritte da noi:

- **main.cpp**, la quale si occupa di eseguire l'intero corpo del programma;
- **compress.cpp**, la quale data un' immagine x esegue delle iterazioni per creare delle sottomatrici che verrano utilizzate poi dalla libreria Fast DCT. Inoltre, una volta che la libreria dct2 esegue il calcolo, questa classe si occupa di ricomporre l'immagine finale;
- **mainwindow.cpp**, classe che gestisce tutti gli aspetti e i trigger della interfaccia grafica;

Inoltre, viene utilizzata la libreria Fast DCT nella quale ricaviamo la DCT2 operando prima per righe e poi per colonne.

2.1 Tool di testing

Una volta avviato il programma, è possibile caricare l'immagine .bmp tramite 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 che f e d rispettino tutti i vincoli, trasmorma l'immagine (tramite la funzione pixmapToMatrix()) in una matrice e la invia alla funzione DCTcompress. Questa funzione divide l'immagine

in blocchi a ognuno del quale viene applicata la DCT2 restituendo poi una matrice che viene passata alla funzione matrixToPixmap() per poter essere visualizzata in output nell'iterfaccia grafica.
In particolare la funzione compress prende in input una matrice di interi e i parametri f e d restituendo in output una matrice di interi. Al suo interno, la matrice di input viene trasformata nel 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 ritorniamo la matrice nel formato int (arrottondando i valori double all'intero più vicino, mettendo a zero i valori negativi e a 255 quelli maggiori di 255).

2.2 Risultati

Il programma è stato testato sulle immagini di prova fornite e sono stati sperimentati diversi valori dei parametri F e d.

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.

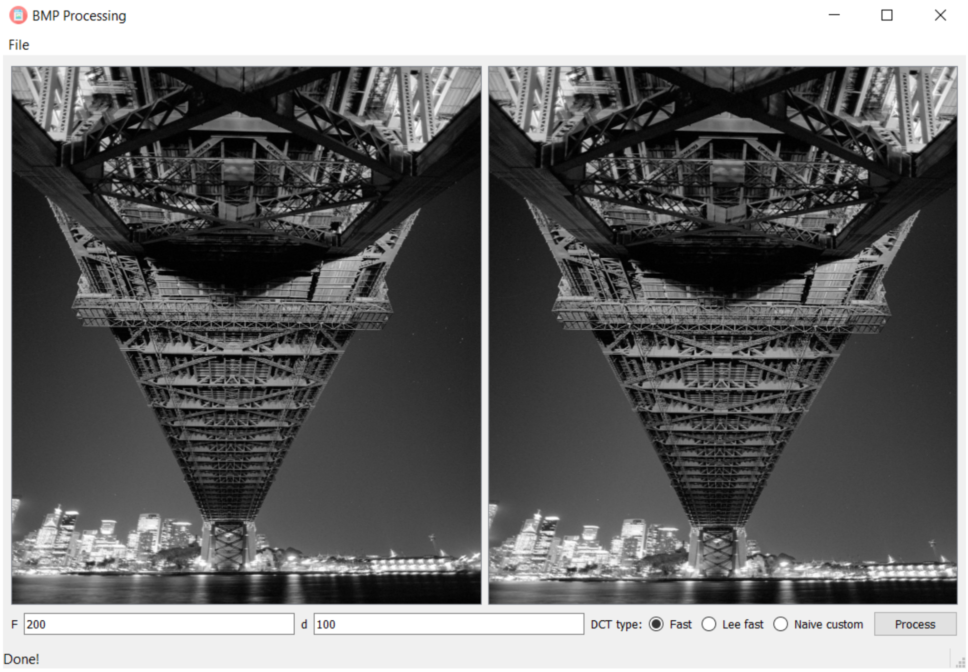

Figure 2.2: *bridge.bmp con F = 200 e d = 100*

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.

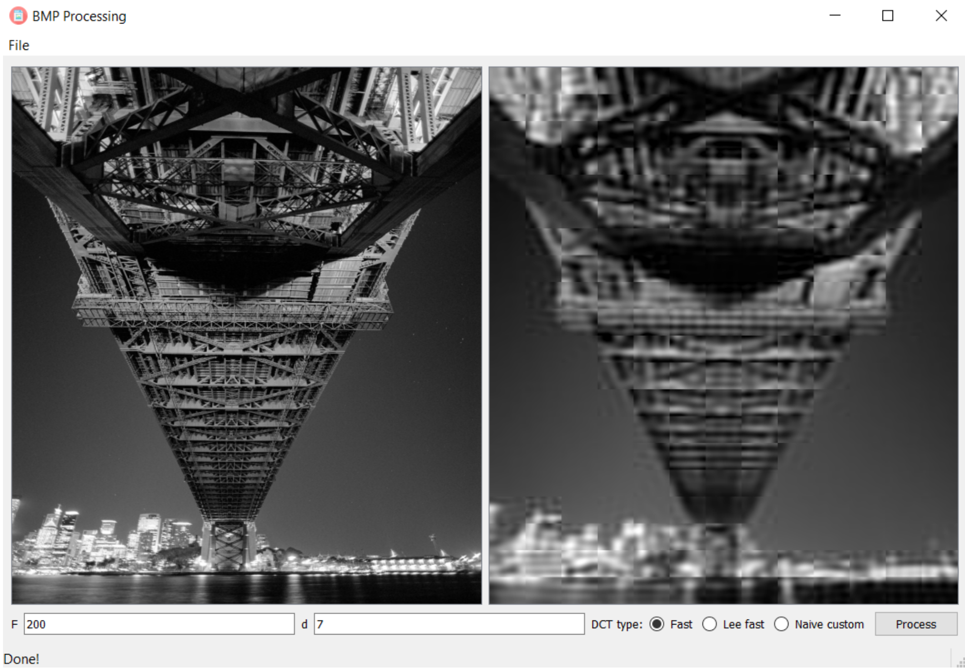

Figure 2.3: *bridge.bmp con* F *= 200 e* d *= 7*

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.

Figure 2.4: *deer.bmp con* F *= 250 e* d *= 50*

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

Figure 2.5: *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.

Figure 2.6: *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.

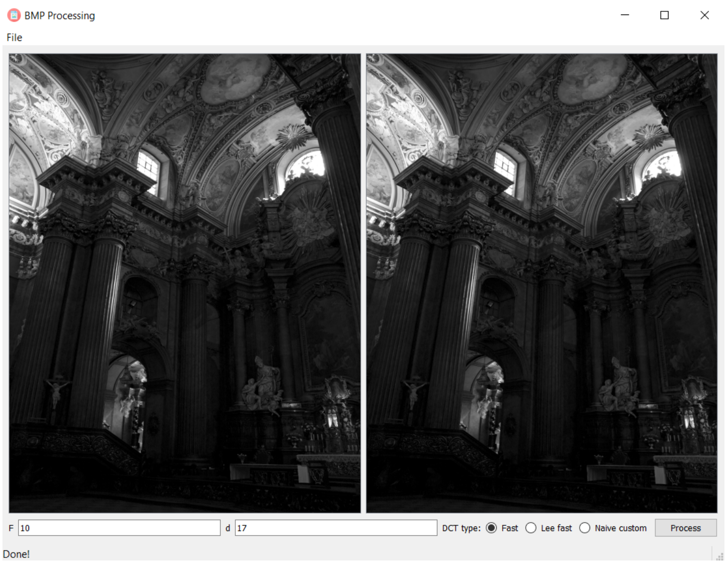

Figure 2.7: *cathedral.bmp con* $F = 10$ *e* $d = 17$