Università di Catania

CORSO DI LAUREA MAGISTRALE (LM-18) IN DATA SCIENCE

- REPORT DI INTELLIGENZA ARTIFICIALE -

Tabu search e Iterated Local Search per la ricostruzione di immagini

---

dott. Francesco Grasso,

prof. Mario Francesco Pavone,
prof. Vincenzo Cutello,
a.a. 2023/2024

Sommario

In questo lavoro sperimenteremo uno dei modi più popolari in cui gli algoritmi meta euristici sono stati applicati all'elaborazione delle immagini: la ricostruzione di un'immagine a partire da un insieme di poligoni semitrasparenti.
Inizieremo con una panoramica sull'elaborazione delle immagini in Python e faremo conoscenza con tre librerie utili: *Pillow*, *scikit-image* e *opencv-python*.
Poi scopriremo come si disegnare un'immagine da zero usando i poligoni e come si calcolare la differenza tra due immagini.

Successivamente, svilupperemo due programmi basati su algoritmi metaeuristici, il Tabu Search (TS), utilizzato per la risoluzione di problemi di ottimizzazione combinatoria che prende spunto dalla ricerca locale, ma include un meccanismo aggiuntivo chiamato "lista tabù" e l'*Iterated Local Search (ILS)* che si basa sull'idea di eseguire ripetutamente una ricerca locale su una singola soluzione, aggiungendo elementi di perturbazione per esplorare più ampiamente lo spazio delle soluzioni.

Entrambi verranno utilizzati per ricostruire un ritratto di un famoso dipinto utilizzando i poligoni. Infine, esamineremo e confronteremo i risultati ottenuti.
In questo lavoro verranno trattati i seguenti argomenti:

- Descrizione di diverse librerie di elaborazione delle immagini per Python.
- Capire come disegnare programmaticamente un'immagine utilizzando i poligoni.
- Scoprire come confrontare sistematicamente due immagini date.
- Usare gli algoritmi TS e ILS, in combinazione con le librerie di elaborazione delle immagini, per ricostruire un'immagine usando i poligoni.

Inizieremo prima fornendo una panoramica dell'attività di ricostruzione delle immagini

Elenco delle figure

Indice

Capitolo 1

Descrizione del Problema

Uno degli esempi più diffusi di utilizzo degli algoritmi metaeuristici a singola soluzione nell'elaborazione delle immagini è la ricostruzione di un'immagine data con un insieme di forme semitrasparenti e sovrapposte.

Oltre all'aspetto ludico e all'opportunità di acquisire esperienza nell'elaborazione delle immagini, questi esperimenti offrono una preziosa visione del processo evolutivo e potrebbero potenzialmente contribuire a una migliore comprensione delle arti visive, nonché a progressi nell'analisi e nella compressione delle immagini.

Nell'ambito di tali esperimenti di ricostruzione d'immagini, si utilizza come riferimento un'immagine nota, spesso un dipinto famoso o un suo dettaglio.

L'obiettivo è quello di costruire un'immagine simile, assemblando un insieme di forme sovrapposte, tipicamente poligoni, di colori e trasparenze diverse. In questo caso, affronteremo questa sfida utilizzando l'approccio offerto dagli algoritmi Tabu Search (TS) e Iterated Local Search (ILS).

1.1 Elaborazione delle immagini in Python

Per raggiungere il nostro obiettivo, dovremo eseguire diverse operazioni di elaborazione delle immagini; ad esempio, dovremo creare un'immagine da zero, disegnare forme su un'immagine, tracciare un'immagine, aprire un file d'immagine, salvare un'immagine su un file, confrontare due immagini ed eventualmente ridimensionare un'immagine. Nelle sezioni seguenti esploreremo alcuni dei modi in cui queste operazioni possono essere eseguite utilizzando Python.

Tra le numerose librerie di elaborazione delle immagini disponibili per gli sviluppatori Python, ho scelto di utilizzare tre delle più importanti. Queste librerie saranno discusse brevemente nelle sottosezioni seguenti.

1.1.1 La libreria Pillow

La libreria Pillow[1] è un fork, attualmente mantenuto, dell'originale Python Imaging Library (PIL). Offre supporto per l'apertura, la manipolazione e il salvataggio di file immagine di vari formati. Poiché ci permette di gestire file immagine, disegnare forme, controllare la loro trasparenza e manipolare i pixel, la utilizzeremo come strumento principale per la creazione dell'immagine ricostruita.
La pagina iniziale di questa libreria si trova al seguente link: https://python-pillow.org/.
Una tipica installazione di Pillow utilizza il comando pip, come segue:

```
pip install Pillow
```

La libreria Pillow utilizza il namespace PIL. Se la libreria PIL originale è già installata, occorre prima disinstallarla.

1.1.2 La libreria scikit-image

La libreria scikit-image[2], sviluppata dalla comunità SciPy, estende scipy.image e fornisce una raccolta di algoritmi per l'elaborazione delle immagini, tra cui l'I/O, il filtraggio, la manipolazione del colore e il rilevamento di oggetti. In questa sede utilizzeremo solo il suo modulo metrico, che serve a confrontare due immagini.
La pagina iniziale di questa libreria si trova al seguente link: https://scikit-image.org/.
scikit-image è preinstallata in diverse distribuzioni Python, come **Anaconda** e winPython. Per installarla, utilizzare l'utility pip, come segue:

```
pip install scikit-image
```

Se si utilizza Anaconda o miniconda, utilizzare il seguente comando:

```
conda install -c conda-forge scikit-image
```

1.1.3 La libreria opencv-python

OpenCV è un'elaborata libreria[3] che fornisce numerosi algoritmi relativi alla visione artificiale e all'apprendimento automatico. Supporta un'ampia varietà di linguaggi di programmazione ed è disponibile su diverse piattaforme.
opencv-python è l'API Python per questa libreria. Combina la velocità dell'API C++ con la facilità d'uso del linguaggio Python. In questo lavoro, utilizzeremo questa libreria principalmente per calcolare la differenza tra due immagini, poiché ci permette di rappresentare un'immagine come un array numerico.
La libreria è composta da quattro pacchetti diversi, che utilizzano tutti lo stesso spazio dei nomi (cv2). Solo uno di questi pacchetti deve essere selezionato per essere installato in un singolo ambiente.
Per i nostri scopi, possiamo usare il seguente comando, che installa solo i moduli principali:

```
pip install opencv-python
```

1.2 Disegnare immagini con poligoni

Per disegnare un'immagine da zero, possiamo usare le classi Image e ImageDraw di *Pillow*, come segue:

```
image = Image.new('RGB', (width, height))
draw = ImageDraw.Draw(image, 'RGBA')
```

'RGB' e 'RGBA' sono i valori dell'argomento. Il valore 'RGB' indica tre valori a 8 bit per pixel, uno per ciascuno dei colori Red ('R'), Green ('G') e Blue ('B'). Il valore 'RGBA' aggiunge un quarto valore a 8 bit, 'A', che rappresenta il livello **alpha** (opacità) dei disegni da aggiungere. La combinazione di un'immagine di base RGB e di un disegno RGBA consente di disegnare poligoni con diversi gradi di trasparenza su uno sfondo nero.
Ora, possiamo aggiungere un poligono all'immagine di base utilizzando la funzione polygon della classe ImageDraw, come mostrato nell'esempio seguente. La seguente istruzione disegna un triangolo sull'immagine:

$$draw.polygon([(x_1, y_1), (x_2, y_2), (x_3, y_3)], (red, green, blue, alpha))$$

L'elenco seguente spiega in modo più dettagliato i termini utilizzati nella formulazione precedente:

- Le tuple (x_1, y_1), (x_2, y_2) e (x_3, y_3) rappresentano i tre vertici del triangolo. Ogni tupla contiene le coordinate x e y del vertice corrispondente all'interno dell'immagine;
- red, green e blue sono valori interi nell'intervallo $[0, 255]$, ciascuno dei quali rappresenta l'intensità del colore corrispondente del poligono;
- alpha è un valore intero nell'intervallo $[0, 255]$, che rappresenta il valore di *opacità* del poligono (un valore più basso significa maggiore trasparenza).

In questo modo si possono aggiungere sempre più poligoni, tutti disegnati sulla stessa immagine, eventualmente sovrapposti, come mostrato Figura 1.1:

Figura 1.1: Un grafico di poligoni sovrapposti con colori e valori di opacità diversi.

Una volta disegnata un'immagine utilizzando i poligoni, è necessario confrontarla con l'immagine di riferimento, come descritto nella prossima sottosezione.

1.3 Misurare la differenza tra immagini

Poiché vogliamo costruire un'immagine il più possibile simile a quella originale, abbiamo bisogno di un modo per valutare la somiglianza o la differenza tra due immagini date. Due possibili metodi sono i seguenti:

- Pixel-based Mean Squared Error (MSE)
- Structural Similarity (SSIM)

1.3.1 Errore quadratico medio basato sui pixel (MSE)

Il modo più comune per valutare la somiglianza tra immagini è quello di effettuare un confronto pixel per pixel. Ciò richiede, ovviamente, che le due immagini abbiano le stesse dimensioni. La metrica MSE può essere calcolata come segue:

1. Calcolare il quadrato della differenza tra ogni coppia di pixel corrispondenti di entrambe le immagini. Poiché ogni pixel del disegno è rappresentato con tre valori separati (red, green e blue) la differenza per ogni pixel viene calcolata su queste tre dimensioni;

2. Calcolare la somma di tutti questi quadrati;

3. Dividere la somma per il numero totale di pixel.

Quando entrambe le immagini sono rappresentate con la libreria OpenCV (cv2), questo calcolo può essere eseguito in modo molto semplice, come segue:

```
MSE = np.sum((cv2Image1.astype("float") -
cv2Image2.astype("float")) ** 2) /
float(numPixels)
```

Quando le due immagini sono identiche, il valore MSE sarà pari a *zero*. Di conseguenza, la minimizzazione di questa metrica può essere utilizzata come obiettivo del nostro algoritmo.

1.3.2 Structural Similarity (SSIM)

L'indice SSIM è stato creato per essere utilizzato per prevedere la qualità dell'immagine prodotta da un determinato algoritmo di compressione, confrontando l'immagine compressa con quella originale.
Invece di calcolare un valore di errore assoluto, come avviene ad esempio con il metodo MSE, l'SSIM si basa sulla percezione e considera i cambiamenti nelle informazioni strutturali, nonché effetti come la luminosità e la consistenza delle immagini.
Il modulo **metrics** della libreria scikit-image fornisce una funzione che calcola l'indice di somiglianza strutturale tra due immagini. Quando entrambe le immagini sono rappresentate con la libreria OpenCV (cv2), questa funzione può essere utilizzata direttamente, come segue:

```
SSIM = structural_similarity(cv2Image1, cv2Image2)
```

Il valore restituito è un float nell'intervallo $[-1, 1]$, che rappresenta l'indice SSIM tra le due immagini date. Un valore pari a *uno* indica immagini identiche.
Per impostazione predefinita, questa funzione confronta le immagini in scala di grigi. Per confrontare le immagini a colori, l'argomento opzionale multichannel deve essere impostato su true.

1.4 Utilizzare gli algoritmi TS e ILS per la ricostruzione delle immagini

Come abbiamo detto in precedenza, il nostro obiettivo per questo lavoro è utilizzare un'immagine nota come riferimento e creare una seconda immagine, il più possibile simile a quella di riferimento, utilizzando un insieme di poligoni sovrapposti di colori e trasparenze diverse. Utilizzando l'approccio degli algoritmi di ricerca locale, ogni soluzione candidata è un insieme di tali poligoni e la valutazione della soluzione viene effettuata creando un'immagine con questi poligoni e confrontandola con l'immagine di riferimento.
Come al solito, la prima decisione da prendere riguarda la rappresentazione di queste soluzioni. Tale rappresentazione è discussa nella prossima sottosezione.

1.4.1 Rappresentazione e valutazione della soluzione

Come abbiamo detto in precedenza, la nostra soluzione consiste in un insieme di poligoni all'interno dei contorni dell'immagine. Ogni poligono ha il suo colore e la sua trasparenza. Disegnare un poligono di questo tipo, utilizzando la libreria Pillow, richiede i seguenti argomenti:

- Un elenco di tuple, $[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$, che rappresentano i vertici del poligono. Ogni tupla contiene le coordinate x e y del vertice corrispondente all'interno dell'immagine. Pertanto, i valori delle coordinate x sono nell'intervallo [0, larghezza dell'immagine -1], mentre i valori delle coordinate y sono nell'intervallo [0, altezza dell'immagine -1];
- Tre valori interi nell'intervallo [0, 255], che rappresentano le componenti red, green e blue del colore del poligono;
- Un valore intero aggiuntivo nell'intervallo [0, 255], che rappresenta il valore alpha (o opacità) del poligono.

Ciò significa che per ogni poligono della nostra collezione avremo bisogno di:
[2 × (dimensione del poligono) +4] parametri.
Un triangolo, ad esempio, richiederà 10 parametri, mentre un esagono ne richiederà 16. Di conseguenza, un insieme di triangoli sarà rappresentato con una **lista** nel formato seguente (dove ogni 10 parametri rappresentano un singolo triangolo):

$$[x_{11}, y_{11}, x_{12}, y_{12}, x_{13}, y_{13}, r_1, g_1, b_1, alpha_1,$$

$$x_{21}, y_{21}, x_{22}, y_{22}, x_{23}, y_{23}, r_2, g_2, b_2, alpha_2, \ldots]$$

Per semplificare questa rappresentazione, utilizzeremo numeri float nell'intervallo [0, 1] per ogni parametro. Prima di disegnare i poligoni, espanderemo ogni parametro in modo che rientri nell'intervallo richiesto: larghezza e altezza dell'immagine per le coordinate dei vertici e [0, 255] per i colori e i valori di opacità.
Utilizzando questa rappresentazione, un insieme di 50 triangoli sarà rappresentato come un elenco di 500 valori float compresi tra 0 e 1, in questo modo:

```
[0.1499488467959301, 0.3812631075049196,
0.0004394580562993053, 0.9988170920722447,
0.9975357316889601, 0.9997461395379549,
0.6338072268312986, 0.379170095245514,
0.29280945382368373, 0.20126488596803083,
...
0.4551462922205506, 0.9912529573649455,
0.7882252614083617, 0.01684396868069853,
0.2353587486989349, 0.003221988752732261,
0.9998952203500615, 0.48148512088979356,
0.11555604920908047, 0.08328550982740457]
```

Valutare una determinata soluzione significa dividere questo lungo elenco in pezzi che rappresentano i singoli poligoni: nel caso dei triangoli, il pezzo avrà una lunghezza di 10. Quindi, bisogna creare una nuova immagine vuota e disegnarvi sopra i vari poligoni dell'elenco, uno per uno. Quindi, occorre creare una nuova immagine vuota e disegnarvi sopra, uno per uno, i vari poligoni dell'elenco. Infine, è necessario calcolare la differenza tra l'immagine risultante e l'immagine originale (di riferimento).
Come abbiamo visto nella sezione precedente, esistono due metodi diversi per calcolare la differenza tra le immagini: l'MSE basato sui pixel e l'indice SSIM. Questa procedura (piuttosto elaborata) di valutazione del punteggio è implementata da una classe Python, che verrà descritta nella prossima sottosezione.

1.4.2 Rappresentazione del problema in Python

Per incapsulare la strategia di ricostruzione delle immagini, ho creato una classe Python chiamata TestImage. Questa classe è contenuta nel file image_test.py.
La classe viene inizializzata con due parametri: il percorso del file contenente l'immagine di riferimento e il numero di vertici dei poligoni utilizzati per costruire l'immagine.
La classe fornisce i seguenti metodi public:

- polygonDataToImage(): Accetta come input l'elenco contenente i dati dei poligoni di cui abbiamo parlato nella sottosezione precedente, divide questo elenco in pezzi che rappresentano i singoli poligoni e crea un'immagine contenente questi poligoni disegnandoli uno per uno su un'immagine vuota;

- getDifference(): Accetta come input i dati dei poligoni, crea un'immagine contenente questi poligoni e calcola la differenza tra questa immagine e quella di riferimento utilizzando uno dei due metodi MSE o SSIM;
- plotImages(): Crea un grafico affiancato dell'immagine data accanto all'immagine di riferimento, a scopo di confronto visivo;
- saveImage(): Accetta i dati dei poligoni, crea un'immagine contenente questi poligoni, crea un grafico affiancato di questa immagine accanto all'immagine di riferimento e salva il grafico in un file.

Durante l'esecuzione dell'algoritmo, TS oppure ILS, il metodo saveImage() sarà chiamato ogni 1000 generazioni per salvare un confronto d'immagini *side-by-side* che rappresenta un'istantanea del processo di ricostruzione.

1.4.3 Implementazione base per entrambi gli algoritmi

I passaggi seguenti descrivono le parti principali di questo algoritmo:

1. Si inizia impostando i valori costanti relativi al problema. POL_SIZE determina il numero di vertici per ogni poligono, mentre POL_COUNT determina il numero totale di poligoni che verranno utilizzati per creare l'immagine ricostruita.

2. Continuiamo creando un'istanza della classe ImageTest, che ci permetterà di creare immagini da poligoni e di confrontarle con l'immagine di riferimento, nonché di salvare istantanee dei nostri progressi:

```
imageTest = TestImage = image_test.TestImage("MonaLisa.png",
                POL_SIZE)
```

3. Successivamente, si impostano i limiti superiore e inferiore per i valori float da ricercare. Come abbiamo detto in precedenza, useremo valori float per tutti i nostri parametri e li imposteremo tutti nello stesso intervallo, tra 0,0 e 1,0, per comodità. Durante la valutazione di una soluzione, i valori verranno espansi al loro intervallo effettivo e convertiti in numeri interi quando necessario:

```
BOUNDS_LOW, BOUNDS_HIGH = 0.0, 1.0
```

4. Ora dobbiamo creare una funzione che crei numeri reali casuali distribuiti uniformemente all'interno di un determinato intervallo. Questa funzione presuppone che l'intervallo sia lo stesso per ogni dimensione, come nel caso della nostra soluzione:

```
def randomFloat(low, up):

return [random.uniform(l, u) for l, u in zip([low] * NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

1.4.4 Implementazione dell'algoritmo TS

Vengono eseguiti i passaggi base per tutti e due algoritmi descritti precedentemente:

1. **TABU_LIST_LENGTH**: questo parametro indica la lunghezza della lista tabu utilizzata nell'algoritmo di ricerca tabu. La lista tabu è una struttura dati utilizzata per memorizzare le mosse precedenti che non devono essere ripetute per un certo numero di iterazioni.
2. **POL_SIZE**: rappresenta la dimensione dei poligoni utilizzati nel problema. Ad esempio, se POL_SIZE = 3, i poligoni saranno triangoli (3 vertici).
3. **POL_COUNT**: indica il numero totale di poligoni utilizzati per rappresentare l'immagine. Un valore più alto di POL_COUNT significa che l'immagine verrà approssimata utilizzando un numero maggiore di poligoni, il che potrebbe portare a una maggiore fedeltà nell'approssimazione, ma anche a un aumento del tempo computazionale.
4. **ITER_NUM**: rappresenta il numero totale di iterazioni che l'algoritmo di ottimizzazione eseguirà. Un valore più alto di ITER_NUM significa che l'algoritmo eseguirà più iterazioni, il che potrebbe portare a una migliore approssimazione dell'immagine, ma anche a un aumento del tempo di esecuzione complessivo.

```
TABU_LIST_LENGTH = 200
POL_SIZE = 3
POL_COUNT = 200
ITER_NUM = 1200000
```

5. Infine applichiamo l'algoritmo TS in accordo al seguente schema:

Algorithm 1: Tabu Search.

s_0 = GenerateInitialSolution;
$s*$ = LocalSearch(s_0);
$V_T = \emptyset$;
while *termination condition met* **do**
  s' = Perturbation($s*$, *history*);
  $s*'$ = LocalSearch(s');
  $s*'$ = UpdateTabuList(s');
  $s*$ = AcceptanceCriterion;
end

Le ricerche locali prendono una potenziale soluzione a un problema e controllano i suoi vicini immediati (cioè soluzioni simili tranne che per pochissimi dettagli minori) nella speranza di trovare una soluzione migliore. Di conseguenza essi possono avere la tendenza a bloccarsi in regioni non ottimali o su altipiani dove molte soluzioni sono ugualmente adatte.

La ricerca Tabu migliora le prestazioni della ricerca locale allentando la sua regola di base. Innanzitutto, ad ogni passo si possono accettare mosse peggiorative se non sono disponibili mosse migliorative (come quando la ricerca è bloccata ad uno stretto minimo locale). Inoltre vengono introdotti divieti (da cui il termine tabu) per scoraggiare la ricerca dal ritornare a soluzioni precedentemente visitate.

L'implementazione di tabu search utilizza strutture di memoria che descrivono le soluzioni visitate o insiemi di regole forniti dall'utente. Se una potenziale soluzione è stata visitata in precedenza entro un certo periodo di tempo breve o se ha violato una regola, viene contrassegnata come " tabu " (vietata) in modo che l' algoritmo non consideri ripetutamente tale possibilità.

1. **Inizializzazione**: Genera una soluzione iniziale s0
2. **Ricerca Locale**: Applica una ricerca locale a s0 per ottenere la miglior soluzione locale
3. **Inizializzazione Lista Tabu**: Inizializza la lista tabu V_T come vuota.
4. **Dentro il ciclo principale**, fino a condizione di terminazione:

   Perturbazione: Genera una nuova soluzione applicando un'operazione di perturbazione alla soluzione corrente, utilizzando eventualmente informazioni dalla lista tabu. Dopo vari tentativi si è arrivata alla conclusione che i migliori valori di perturbazione sono i seguenti:

   Per le coordinate dei vertici, si aggiunge un valore casuale nell'intervallo [-0.005, 0.005]. Questo valore controlla la deviazione massima delle coordinate dei vertici

dalla posizione originale.

Per i valori di colore e trasparenza, si aggiunge un valore casuale nell'intervallo [-0.05, 0.05]. Questo valore determina quanto possono variare i valori di colore e trasparenza dei poligoni.

L'obiettivo è introdurre una variazione casuale sufficientemente piccola da esplorare lo spazio delle soluzioni in modo efficiente, senza allontanarsi troppo dalla soluzione attuale.

Se la soluzione perturbata è già presente nella lista tabù, la funzione ricorsivamente applica una nuova perturbazione finché non trova una soluzione non tabù.

Mutazione: viene applicata ai parametri della soluzione, che includono le coordinate dei vertici dei poligoni e i valori di colore e trasparenza.

Per ogni parametro, si aggiunge un valore casuale nell'intervallo [-0.002, 0.002]. Questo valore controlla la deviazione massima dei parametri dalla loro posizione originale.

L'obiettivo della mutazione è introdurre piccole variazioni casuali nei parametri della soluzione per esplorare lo spazio delle soluzioni e cercare di migliorare la soluzione attuale.

Tabu Search: Questa funzione implementa l'algoritmo di tabu search. Viene inizializzata una lista tabù vuota. Per ogni iterazione, viene generata una nuova soluzione perturbando la migliore soluzione corrente e applicando quindi una mutazione.

La nuova soluzione viene valutata e aggiunta alla lista tabù. Se la lista tabù supera la sua lunghezza massima, viene rimossa la soluzione più vecchia.

Se la nuova soluzione è migliore della migliore soluzione finora trovata, diventa la nuova migliore soluzione. In caso contrario, si incrementa il contatore di iterazioni senza miglioramento.

Questo ciclo viene ripetuto fino a quando una determinata condizione di terminazione viene soddisfatta, nel nostro caso un numero massimo di interazioni.

1.4.5 Implementazione dell'algoritmo ILS

Vengono eseguiti i passaggi base per tutti e due algoritmi descritti precedentemente e applicati i seguenti parametri:

```
POL_SIZE = 3
POL_COUNT = 110
BOUNDS_LOW, BOUNDS_HIGH = 0.0, 1.0
```

Infine, applichiamo l'algoritmo ILS in accordo al seguente schema:

Algorithm 1: Iterated Local Search.

s_0 = GenerateInitialSolution;
$s*$ = LocalSearch(s_0);
$V_T = \emptyset$;
while *termination condition met* **do**
  s' = Perturbation($s*$, *history*);
  $s*'$ = LocalSearch(s');
  $s*$ = AcceptanceCriterion;
end

- GenerateInitialSolution: Inizialmente l'algoritmo genera casualmente un punto, tale punto sarà la prima soluzione trovata.
- LocalSearch(): Viene fatta una ricerca locale verificando se il punto è quello ottimale.
- Viene instanziato un ciclo while che termina dopo un numero prefissato di iterazioni. In particolare nel ciclo while:
  - La soluzione attuale viene perturbata con lo scopo di trovare una soluzione vicina migliore di quella attuale. Dopo vari tentativi, si è riscontrato che i migliori valori di perturbazione sono i seguenti:

    ```
    offset=i+random.uniform(-0.1,  0.1)
    casuale=random.randint(1,  128)
    ```

    Per valori troppo alti di perturbazione l'algoritmo risultata sì rapido nella fase iniziale, ma decisamente più lento nella fase finale dove la soluzione risultava molto vicina all'obiettivo.
  - La soluzione candidata viene sottoposta a valutazione mediante la funzione getDifference(), descritta nella sezione precedente.
  - Se la soluzione candidata è migliore rispetto alla soluzione attuale viene operato lo scambio.
- Infine, terminato il ciclo while, restituiamo la soluzione ottimale trovata.

Capitolo 2

Risultati Ottenuti

Per testare il programma, utilizzeremo la seguente immagine, che fa parte del famoso ritratto della Gioconda di Leonardo da Vinci [4], considerato il dipinto più conosciuto al mondo:

Figura 2.1: Ritratto ritagliato del celebre dipinto della Gioconda di Leonardo da Vinci.

Eseguiremo entrami gli algoritmi utilizzando lo stesso tempo di esecuzione, ovvero 1h30min, in modo tale da riuscire a capire il comportamento di entrambi per poterli mettere a confronto, a pari tempo di esecuzione.

Al termine dell'esecuzione, possiamo tornare indietro ed esaminare le immagini salvate per seguire l'evoluzione dell'immagine ricostruita.

2.1 TS - utilizzo dell'errore quadratico medio basato sui pixel

Inizieremo utilizzando la metrica MSE basata sui pixel per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita. Di seguito sono riportate alcune immagini estratte durante la fase di esecuzione e memorizzate *side-by-side*:

Figura 2.2: Dopo 1.000 iterazioni di MSE

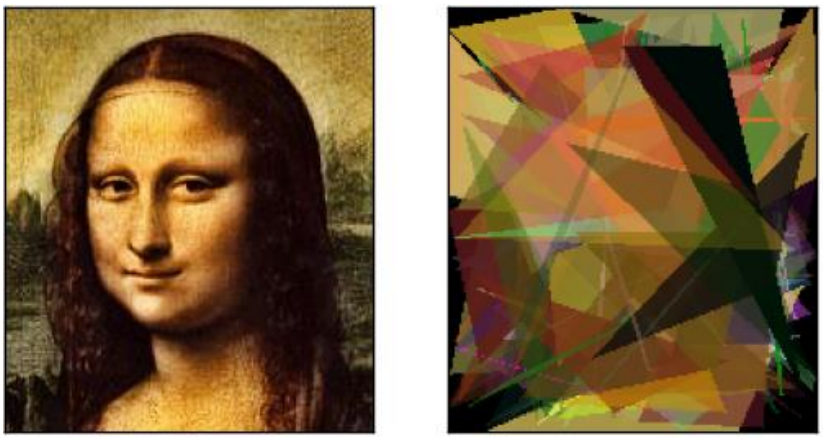

Figura 2.3: Dopo 5.000 iterazioni di MSE

Figura 2.4: Dopo 10.000 iterazioni di MSE

Figura 2.5: Dopo 50.000 iterazioni di MSE

Figura 2.6: Dopo 199.000 iterazioni di MSE

Il risultato finale è simile all'immagine originale, anche se contiene angoli acuti e linee rette, come ci si aspetterebbe da un'immagine basata su poligoni.

Successivamente, proveremo l'altro metodo per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita: l'indice SSIM.

2.2 TS - Utilizzo dell'indice SSIM

Ora ripeteremo l'esperimento, ma questa volta utilizzando la metrica SSIM per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita. Per fare ciò, modificheremo la definizione di getDiff(), come segue:

```
def getDiff(element):
    return imageTest.getDifference(element, "SSIM")
```

Di seguito sono riportate alcune immagini estratte durante la fase di esecuzione

Figura 2.8: Dopo 1.000 iterazioni di SSIM

Figura 2.9: Dopo 5.000 iterazioni di SSIM

Figura 2.10: Dopo 10.000 iterazioni di SSIM

Figura 2.11: Dopo 50.000 iterazioni di SSIM

Figura 2.12: Dopo 199.000 iterazioni di SSIM

Il risultato appare interessante: l'algoritmo ha catturato la struttura dell'immagine ma in modo più grossolano rispetto ai risultati ottenuti con l'MSE. Anche i colori sembrano un pò spenti, dato che l'SSIM si concentra maggiormente sulla struttura e sulla texture.
Di seguito, la Figura 2.13, riporta una versione sfocata delle immagini finali messe a confronto:

2.3 ILS - Utilizzo dell'errore quadratico medio basato sui pixel

Inizieremo utilizzando la metrica MSE basata sui pixel per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita. Di seguito sono riportate alcune immagini estratte durante la fase di esecuzione e memorizzate *side-by-side*:

Figura 2.2: Dopo 1.000 iterazioni di MSE

Figura 2.3: Dopo 5.000 iterazioni di MSE

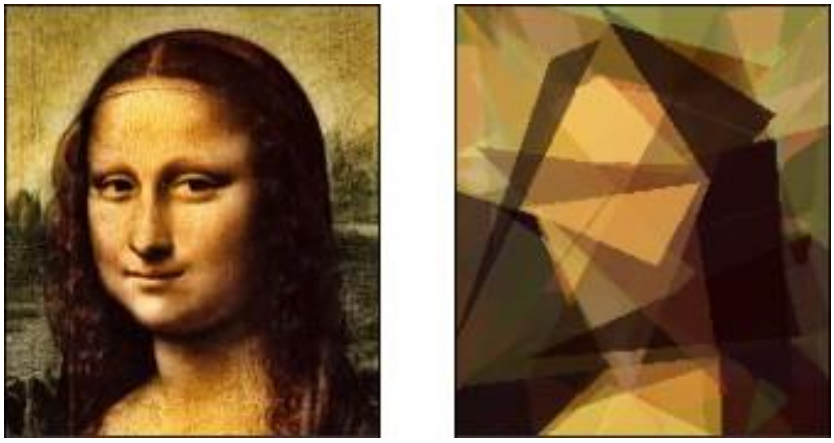

Figura 2.4: Dopo 10.000 iterazioni di MSE

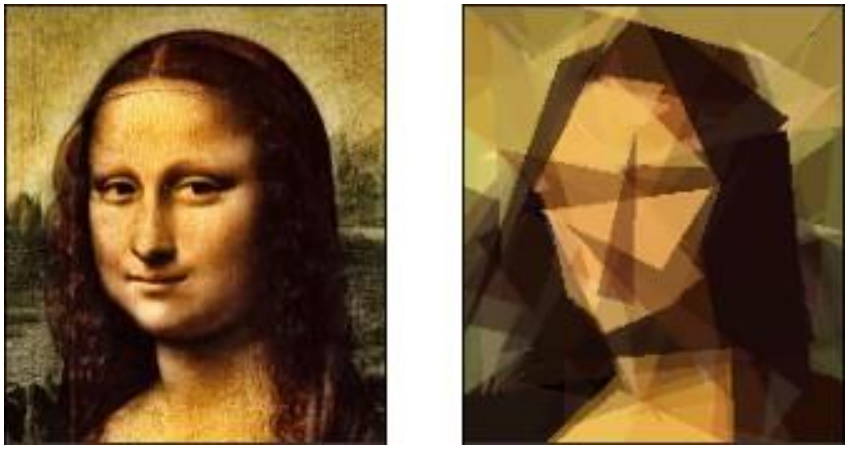

Figura 2.5: Dopo 50.000 iterazioni di MSE

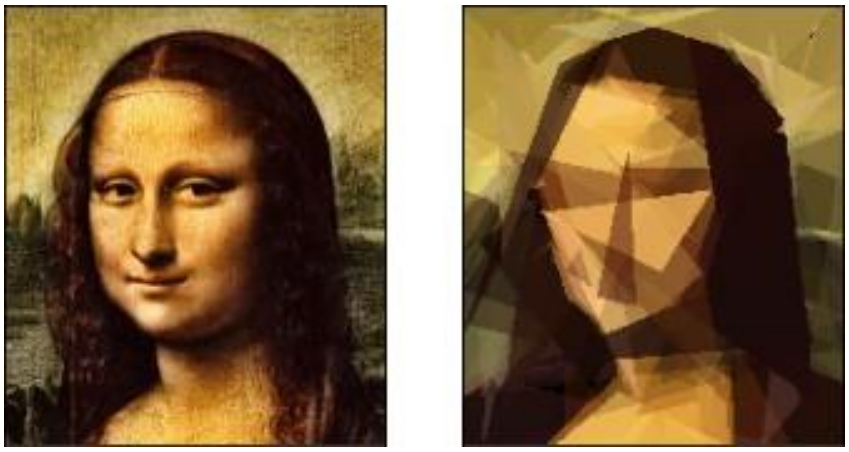

Figura 2.6: Dopo 199.000 iterazioni di MSE

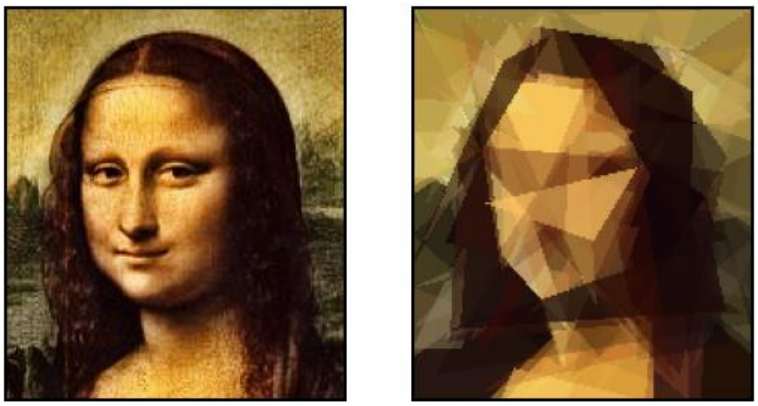

Figura 2.6: Dopo 300.000 iterazioni di MSE

Il risultato finale è molto simile all'immagine originale, anche se contiene angoli acuti e linee rette, come ci si aspetterebbe da un'immagine basata su poligoni.

Successivamente, proveremo l'altro metodo per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita: l'indice SSIM.

2.4 ILS - Utilizzo dell'indice SSIM

Ora ripeteremo l'esperimento, ma questa volta utilizzando la metrica SSIM per misurare la differenza tra l'immagine di riferimento e l'immagine ricostruita. Per fare ciò, modificheremo la definizione di getDiff(), come segue:

```
def getDiff(element):
    return imageTest.getDifference(element, "SSIM")
```

Di seguito sono riportate alcune immagini estratte durante la fase di esecuzione e memorizzate *side-by-side*:

Figura 2.8: Dopo 1.000 iterazioni di SSIM

Figura 2.9: Dopo 5.000 iterazioni di SSIM

Figura 2.10: Dopo 10.000 iterazioni di SSIM

Figura 2.11: Dopo 50.000 iterazioni di SSIM

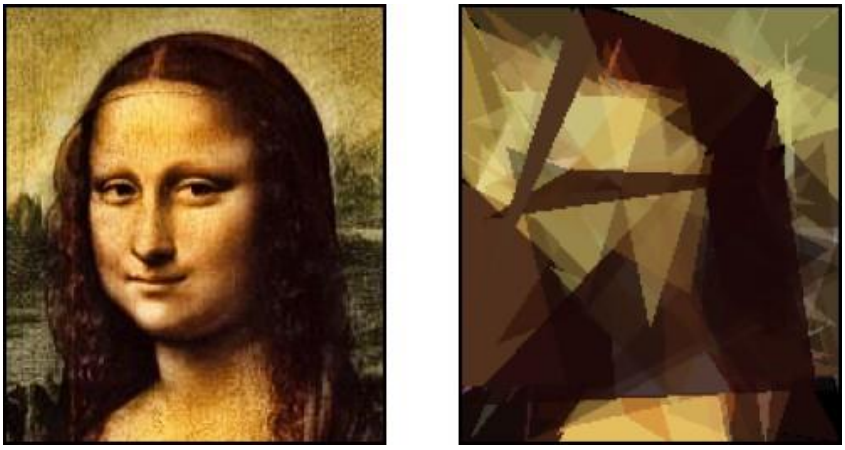

Figura 2.12: Dopo 199.000 iterazioni di SSIM

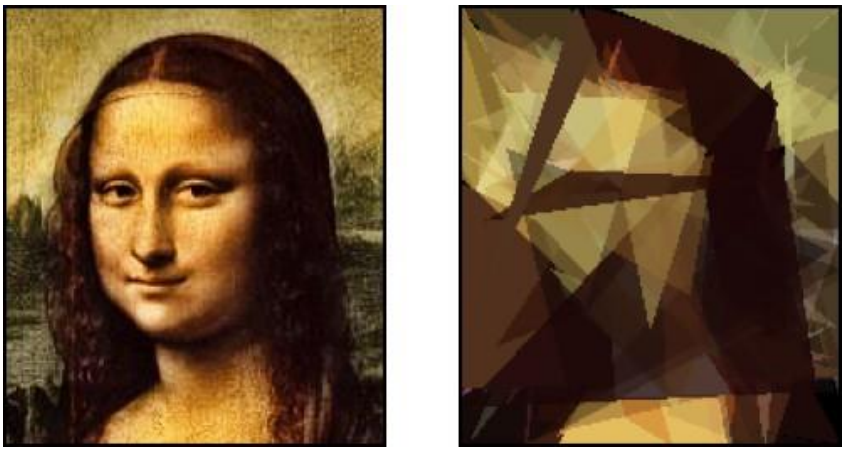

Figura 2.12: Dopo 300.000 iterazioni di SSIM

Il risultato appare interessante: l'algoritmo ha catturato la struttura dell'immagine ma in modo più grossolano rispetto ai risultati ottenuti con l'MSE. Anche i colori sembrano un pò spenti, dato che l'SSIM si concentra maggiormente sulla struttura e sulla texture.

Capitolo 3

Conclusioni

Obiettivo di questo progetto era introdurre il concetto di ricostruzione di immagini esistenti utilizzando un insieme di poligoni sovrapposti e semitrasparenti. Durante la stesura del progetto, si è familiarizzato con diverse librerie di elaborazione delle immagini in Python [1.1] e si è appreso come sia possibile disegnare da zero un'immagine utilizzando dei poligoni [1.2], nonché come calcolare la differenza tra due immagini [1.3]. Successivamente, abbiamo sviluppato due programmi, uno basato sull'algoritmo Tabu Search e uno su un algoritmo meta euristico a singola soluzione, Iterated Local Search, che ricostruisce un segmento di un famoso dipinto utilizzando i poligoni [1.4].
Guardando ai risultati complessivi ottenuti dall'esperimento [2], a prescindere da quello che risulta più ottimizzato, si può sicuramente affermare che gli approcci messi in campo hanno permesso di raggiungere l'obiettivo prefissato.

Cosa abbiamo potuto notare con i parametri utilizzati (i migliori in fase di test):

Il Tabu Search ha impiegato più tempo per convergere rispetto all'Iterative Local Search. Ciò è dovuto alla natura dell'algoritmo che esplora più intensamente lo spazio delle soluzioni.
L'Iterative Local Search, essendo più focalizzato sulla ricerca locale e sull'esplorazione di vicini miglioramenti, tende ad arrivare a una soluzione soddisfacente in meno tempo.

Tabu Search tende a produrre soluzioni più accurate e di qualità superiore. Questo è dovuto alla sua capacità di esplorare più a fondo lo spazio delle soluzioni e di superare i minimi locali attraverso l'utilizzo della lista tabù. È infatti possibile notare che nell'immagine che rappresenta le ultime interazioni la qualità dell'immagine è leggermente superiore strutturalmente.
Infatti Anche se ILS può essere più veloce, la sua ricerca potrebbe essere influenzata dalla presenza di minimi locali, portando a soluzioni di qualità inferiore rispetto a Tabu Search.

Bibliografia

[1] Documentazione ufficilale di Pillow, https://pillow.readthedocs.io/en/stable/index.html

[2] Documentazione ufficiale di scikit-image, https://scikit-image.org/docs/stable/index.html

[3] Documentazione ufficilale di OpenCV, https://docs.opencv.org/master/

[4] Mona Lisa headcrop, https://commons.wikimedia.org/wiki/File:Mona_Lisa_headcrop.jpg

[5] Hands-On Image Processing with Python: Expert techniques for advanced image analysis and effective interpretation of image data (30 novembre 2018), *Sandipan Dey*

[6] Tabu Search, https://en.wikipedia.org/wiki/Tabu_search