



# Salienza Strutturale

Rilevamento di linee salienti nelle  
immagini tramite algoritmo di  
Sha'ashua-Ullman

**Alberto Pettini**

Febbraio 2006

# Indice

<b>Indice</b>	<b>I</b>
<b>1 Introduzione</b>	<b>1</b>
<b>2 Salienza Strutturale</b>	<b>3</b>
2.1 Salienza Locale e Globale . . . . .	3
2.2 Elementi di Orientamento . . . . .	4
2.3 Calcolo della Misura di Salienza . . . . .	5
<b>3 Estrazione di linee salienti</b>	<b>7</b>
3.1 Ambiguità . . . . .	8
3.2 Cicli . . . . .	10
<b>4 Implementazione</b>	<b>13</b>
4.1 Approccio . . . . .	13
4.2 Classi sviluppate . . . . .	13
4.3 Esempi di esecuzione . . . . .	15
4.4 Applicazione ad immagini reali . . . . .	18
4.5 Tempi di esecuzione . . . . .	19
4.6 Considerazioni . . . . .	21
<b>Bibliografia</b>	<b>23</b>

# Capitolo 1

## Introduzione

Quando guardiamo una immagine a volte la nostra attenzione si sposta immediatamente su alcune parti di essa. Questo accade istintivamente, senza una complessa analisi strutturale dell' immagine. Il fenomeno di percezione appena descritto è stato oggetto negli anni di numerosi studi sulla psicologia, e coinvolge diverse proprietà di una immagine come colori, orientamento dei singoli elementi, contrasto, curvatura delle linee...

Un semplice esempio di quanto accennato viene proposto in Fig. 1.1. Immediatamente e senza nessuno sforzo notiamo al centro della figura l' immagine di una macchina, su uno sfondo di linee disordinate. Se analizziamo però localmente l' immagine notiamo subito che apparentemente ogni singola linea sembra non avere sostanziali differenze rispetto alle linee che compongono il resto dello sfondo.

Scopo di questa tesina è presentare brevemente il metodo proposto in [SU88] da Ammon Sha'ashua e Shimon Ullman per il calcolo della cosiddetta *mappa di salienza*, utilizzata per mettere in evidenza le linee salienti di una immagine, e la successiva estrazione delle stesse (per una trattazione completa dell' argomento vedere l' articolo originale). Segue poi una implementazione di tale algoritmo in linguaggio Java, volta a dimostrare la validità del metodo studiato ed a meglio comprenderne i dettagli di funzionamento. Vengono quindi presentate la documentazione su struttura e

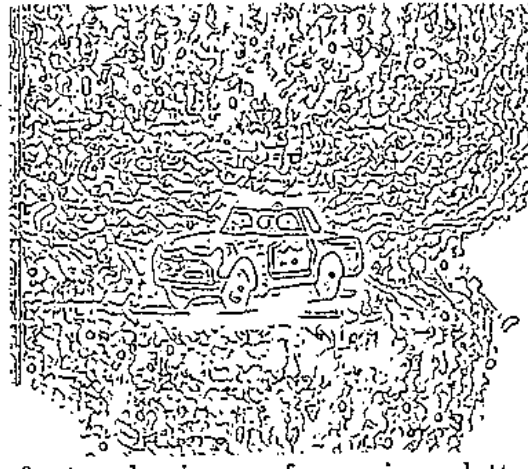


Figura 1.1: Una macchina su uno sfondo frastagliato.

funzionalità delle Classi progettate, l' analisi ed il funzionamento dell' applicazione ottenuta, riportando infine i risultati dei test effettuati ed il comportamento dell' applicazione a fronte di diversi input.

## Capitolo 2

# Salienza Strutturale

### 2.1 Salienza Locale e Globale

Il fenomeno di percezione umana di strutture salienti è dovuto fondamentalmente a due fattori: la *salienza locale* e la *salienza strutturale* (o *globale*). Il primo entra in gioco quando un elemento si distingue da quelli che circondano per caratteristiche quali il colore, il contrasto, l'orientamento, etc. Per fare un esempio basti pensare ad un punto blu su uno sfondo completamente giallo. Il secondo fattore è percepito invece dal nostro cervello non per una sua caratteristica specifica, ma come parte di una struttura globale: elementi che localmente non hanno caratteristiche salienti possono dunque risultare salienti nel loro insieme. Il problema della misura della salienza strutturale richiede quindi un'analisi più complessa dell'immagine nella sua interezza. A tale scopo ci proponiamo quindi di costruire una *Mappa di Salienza*, ovvero una rappresentazione dell'immagine che enfatizzi le sue parti salienti. Per fare ciò dobbiamo definire una *misura* di salienza, che otteniamo tramite una funzione  $\Phi$ . Tale funzione, applicata ad un punto  $P$  del piano, tiene conto di:

- lunghezza della curva passante per  $P$
- smoothness della linea (misura della variazione di curvatura di una curva)

La misura di  $\Phi$  cerca di enfatizzare curve lunghe e con bassa curvatura, premiandole con valori elevati di Salienza.

## 2.2 Elementi di Orientamento

Per analizzare un'immagine la rappresentiamo tramite una griglia di  $n \times m$  punti (pixel). Per ogni punto  $P$  della griglia definiamo  $k$  **elementi di orientamento** uscenti da  $P$  ed altrettanti entranti dai punti vicini (fig. 2.1).

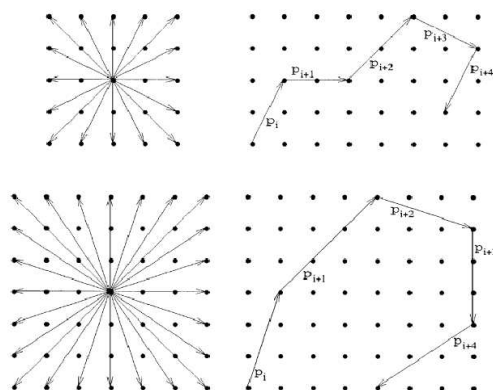


Figura 2.1: Elementi di orientamento.

Ogni elemento  $p_i$  segnala la presenza della corrispondente porzione di linea nell'immagine (in tal caso l'elemento è detto *active* o *real*). Elementi che non segnalano segmenti (anche detti *virtual*) sono associati a gap (fig. 2.2).

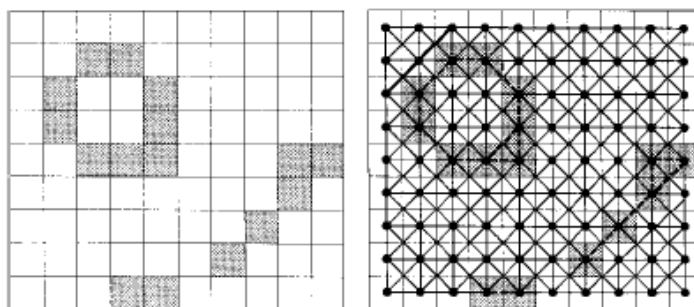


Figura 2.2: Elementi active e virtual.

Nelle implementazioni  $k$  assume solitamente valori di 8, 16 o 24: maggiore è il numero di elementi di orientamento per pixel e maggiore è la precisione

dell' algoritmo. In quest' ottica una curva di lunghezza  $N$  è rappresentata dalla sequenza di elementi  $p_i \dots p_{i+N}$  (ognuno dei quali è una porzione di linea o un gap).

## 2.3 Calcolo della Misura di Salienza

Per il calcolo della Mappa di Salienza vengono assegnati ad ogni elemento  $p_i$  una serie di attributi:

- lo Stato  $E_i$
- la salienza locale  $\sigma_i$
- l' orientamento  $\Theta_i$ , espresso in radianti
- il fattore di attenuazione  $\rho_i$ , che assume il valore 1 se l' elemento è *active*, un valore minore di 1 se l'elemento è *virtual*

Ogni elemento aggiorna il suo Stato iterativamente, eseguendo un processo decisionale *step by step*, proprio della Programmazione Dinamica. In un approccio discreto la computazione si basa sulle seguenti formule:

$$E_i^{(0)} = \sigma_i$$

$$E_i^{(n+1)} = \sigma_i + \rho_i \max_{p_j \in \delta(p_i)} E_j^{(n)} f_{i,j}$$

La prima iterazione dunque inizializza lo Stato con il valore di salienza locale dell' elemento corrente. Nelle successive iterazioni ogni elemento esplora gli elementi del pixel adiacente ( $\delta(p_i)$  rappresenta l' insieme degli elementi di orientamento di  $p_i$ ), alla ricerca di quello che fornisce il contributo più elevato, tenendo conto dell' influenza del fattore di attenuazione. Tale contributo si ottiene moltiplicando lo Stato (al passo  $n-1$ ) dell' elemento vicino per la funzione  $f$ , così definita:

$$f_{k,k+1} = e^{-\frac{2\alpha_k \tan \frac{\alpha}{2}}{\Delta s}}$$

Al termine dell' iterazione  $n$ ,  $E_i$  conterrà la misura della salienza, che sarà massima tra tutte le possibili curve di lunghezza  $n$  che partono da  $p_i$  (continue o con gap).

## Capitolo 3

# Estrazione di linee salienti

L'estrazione delle linee salienti, fine ultimo dell' algoritmo, è ottenuta tramite due step fondamentali, come mostrato in fig. 3.1.



Figura 3.1: Procedimento di estrazione

Questo procedimento (fig. 4.2, a destra) consente l'isolamento delle linee con salienza più elevata e la ricostruzione di parti di esse che presentano discontinuità (gap filling).

A partire dalla Mappa di Salienza l'estrazione e la ricostruzione di una linea avviene nel seguente modo:

- si ricava una mappa binaria a partire dalle informazioni della Mappa attuale (corrispondente all' ultima iterazione dell' algoritmo)
- si analizzano in sequenza tutti i punti rilevati nella mappa binaria (pixel con alto valore di salienza): ogni punto rappresenta uno *start point*, ovvero un punto di partenza di una linea saliente
- il punto viene marcato come “visitato” ed aggiunto ad un vettore di punti rappresentante la linea saliente. Tale vettore verrà utilizzato successivamente per l' output a video

- ogni punto sceglie una direzione da seguire (quindi il prossimo pixel da analizzare) in base da una *local preference*, stabilita dal proprio elemento di orientamento con valore più alto di salienza
- si continua ad “inseguire” i pixel preferiti fin quando non si arriva ad un punto con salienza “0” per ogni elemento di orientamento, oppure fin quando non si incontra un ciclo (ovvero un pixel già facente parte della linea attuale). In tal caso il punto decreta la terminazione della linea saliente
- si rappresenta su una nuova immagine di output la linea così ottenuta e si prosegue nell’elaborazione della mappa binaria, analizzando il successivo *start point*

L’implementazione qui descritta presenta però qualche difficoltà nella ricostruzione delle linee più salienti (ad esempio allo scopo di gap filling), impedendo il corretto “inseguimento” delle linee elaborate, in due casi rilevanti: *ambiguità strutturali* e *cicli*.

### 3.1 Ambiguità

Le *ambiguità* nascono nel momento in cui un pixel si trova a dover far delle scelte in presenza di elementi di orientamento con uguale valore di salienza: in questo caso la *local preference* del pixel non garantisce l’ottimalità della scelta. Vi sono infatti più direzioni possibili da seguire, tutte localmente apparentemente favorevoli nell’individuazione di linee salienti.

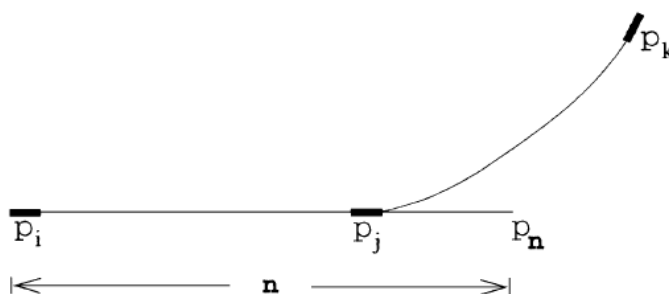


Figura 3.2: Local preference

Prendiamo come esempio la computazione della linea di fig. 3.2 che ci mostra la scelta di un pixel: dopo un numero finito di iterazioni la curva più saliente passante per  $p_i$  può non coincidere con la curva più saliente passante per  $p_j$ . In particolare all'  $n$ -esima iterazione  $p_i$  è influenzato da punti *al più* a distanza  $n$  (si può formalizzare il concetto dicendo che  $p_i$  “vede” a distanza  $n$ ). Per questo motivo la curva più saliente passante per esso sarà la linea retta di lunghezza  $n$ . Al contrario partendo da  $p_j$  la scelta della curva più saliente cadrà sulla porzione di curva da  $p_j$  a  $p_k$ .

Nel caso appena visto, durante l' estrazione delle linee più salienti il punto  $p_j$  sceglierà come direzione da seguire quella verso  $p_k$  (supponendo che lo *start point* sia  $p_i$ ), ottenendo una curva ottimale. Un risultato differente si sarebbe ottenuto prendendo  $p_n$  come *start point*:  $p_j$  effettuerà sempre la propria scelta, localmente ottimale, convogliando la linea saliente verso  $p_k$ .

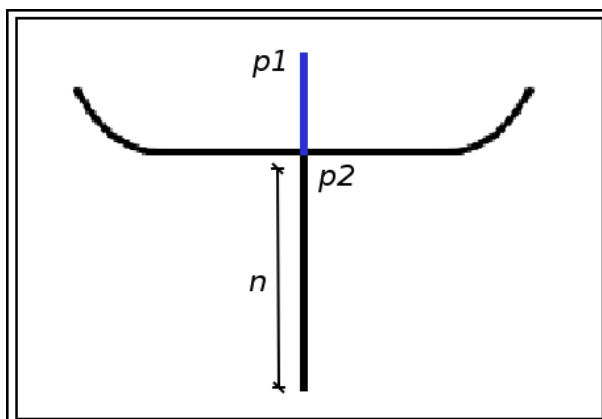


Figura 3.3: Ambiguità

Nella fig. 3.3 ci troviamo invece in presenza di una situazione ambigua. Il punto  $p2$  si trova sull' intersezione di due linee (ad esempio una linea retta che attraversa un cerchio), e deve operare una scelta in una situazione di simmetria. In tal caso gli elementi di orientamento in direzione dei pixel della linea intersecata avranno pari valore di salienza. Supponendo che il segmento blu proveniente da  $p1$  sia la porzione di linea già computata, la scelta effettuata da  $p2$  non garantisce in questo caso l' ottimalità della linea

ottenuta.

Con un numero di iterazioni minore di  $n$  (non noto a priori), infatti, il pixel  $p2$  potrebbe avere più elementi di orientamento con lo stesso valore di stato, ottenendo dunque una local preference non univoca. Nella fig. 3.4 sono mostrate in rosso le possibili scelte effettuate dal pixel. In tal caso non è possibile controllare a priori la scelta operata: il risultato potrebbe non coincidere con l' ottimo.

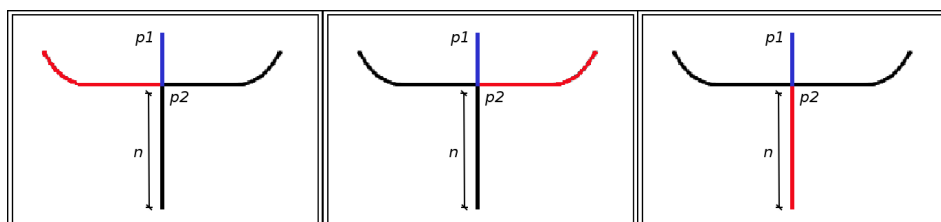


Figura 3.4: Ambiguità: possibili scelte

## 3.2 Cicli

I cicli si presentano spesso in situazioni in cui si elaborano “agglomerati” di punti: in tal caso gruppi di pixel si scelgono reciprocamente nel calcolo della *local preference*, andando a vanificare l' inseguimento di una linea saliente. Tale fenomeno non è raro in presenza di linee con spessori marcati o diramazioni con piccole differenze di angolo.

L' immagine in fig. 3.5 mostra una porzione di immagine con un possibile caso di computazione interrotta a causa di un ciclo. I pixel colorati rappresentano l' intersezione tra due linee. Le frecce bianche mostrano le scelte effettuate dai pixel in base alle proprie local preference (supponiamo di essere all'  $n$ -esima iterazione). Prendendo come *start point* il pixel  $s$  e ricordando che la local preference di un punto non può mai coincidere con la direzione di provenienza della linea (un pixel non può scegliere quindi il suo predecessore, altrimenti instaurerebbe immediatamente un ciclo), notiamo come un gruppo di pixel possa operare delle scelte localmente corrette ma

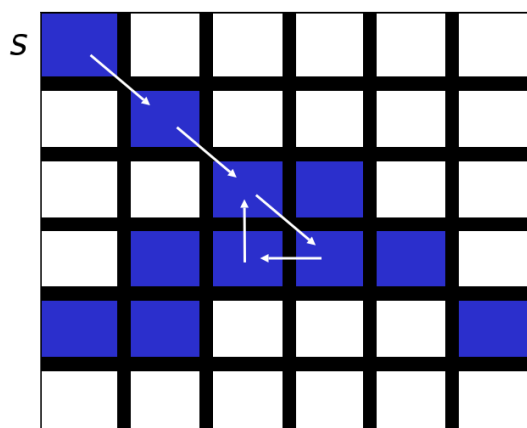


Figura 3.5: Un possibile ciclo

nell'insieme non ottimali. Ogni pixel, nel rispetto dei vincoli locali di direzione, sceglie come successore il vicino che contribuisce maggiormente alla crescita della salienza totale della curva, senza tenere conto dell'andamento della linea nel suo insieme.

In presenza di un ciclo l'elaborazione della linea corrente viene interrotta ed il vettore di punti individuati fino a quel momento viene mandato in output.



## Capitolo 4

# Implementazione

### 4.1 Approccio

Viene qui descritta l' implementazione dell' algoritmo precedentemente introdotto. Come linguaggio di riferimento è stato utilizzato il Java, appoggiandosi alle librerie native per la manipolazione delle immagini. Di seguito viene riportata una descrizione di alto livello delle classi implementate, documentando gli elementi di maggiore interesse ai fini della comprensione dell' algoritmo e tralasciando i dettagli implementativi. Vengono poi esaminati degli esempi di esecuzione dell' applicazione ed i relativi output. Infine viene descritto nel dettaglio il procedimento di individuazione ed estrazione delle linee salienti, che vede la Mappa di Salienza come struttura intermedia di supporto.

### 4.2 Classi sviluppate

Nella figura 4.1 è viene mostrato il diagramma UML delle classi.

#### **SaliencyComputer**

E' la classe che contiene tutti i metodi per la computazione della Mappa di Salienza e per l' estrazione delle linee salienti.

**Attributi principali:**

- *mappa[][]* : la mappa dei Pixel, un array bidimensionale di oggetti, delle stesse dimensioni dell' immagine da processare

**Metodi principali:**

- *computeLocalSaliency()* : imposta le localSaliency di tutti gli Element di ogni Pixel e lo Stato iniziale (sono uguali al primo passaggio)
- *computeSaliency()* : metodo alla base di ogni iterazione. Ricalcola il valore di Salienza di ogni elemento
- *creaMappaSalienzaImg()* : crea l' immagine corrispondente alla Mappa di Salienza per mandarla in output
- *estraiLineeSalienti()* : estrae linee salienti dalla Mappa di Salienza per stamparle a video

**Pixel**

E' la classe che rappresenta il singolo Pixel. Ad ogni oggetto che istanzia questa classe sono delegate le responsabilità di gestione dei propri elementi di orientamento.

**Attributi principali:**

- *elements[]* : array contenente oggetti di tipo Element, uno per ogni orientamento prefissato

**Metodi principali:**

- *findMaxState()* : dato un elemento di orientamento di un pixel adiacente restituisce lo stato dell' elemento del pixel corrente con valore massimo di [Stato \* couplingConstant]. Svolge il ruolo principale nella *computeSaliency()* del SaliencyComputer
- *couplingConstant()* : calcola la Coupling Constant

## Element

L' elemento base introdotto dall' algoritmo.

### Attributi principali:

- *stato* : stato attuale dell' elemento
- *localSaliency* : salienza locale
- *orientamento*
- *attenuazione* : fattore di attenuazione dell' elemento (0.7 in questa implementazione)

## SwingImage

Classe che si occupa dell' interfaccia grafica. Crea e mostra a video tre box, nei quali sono messe a confronto l' immagine di input, la relativa Mappa di Salienza e le linee salienti estratte.

### Metodi principali:

- *createAndShowGUI()* : crea la finestra con le immagini di input ed output

## Salienza

E' la main class. Si occupa inizialmente del parsing dei parametri di input. Da essa viene quindi istanziato opportunamente un oggetto SaliencyComputer, del quale vengono richiamati i metodi. Istanza infine un oggetto SwingImage per la creazione dell' interfaccia grafica.

## 4.3 Esempi di esecuzione

Per eseguire l'applicazione lanciare:

```
java Salienza fileInput numero_iterazioni soglia_salienza [-gui]
```

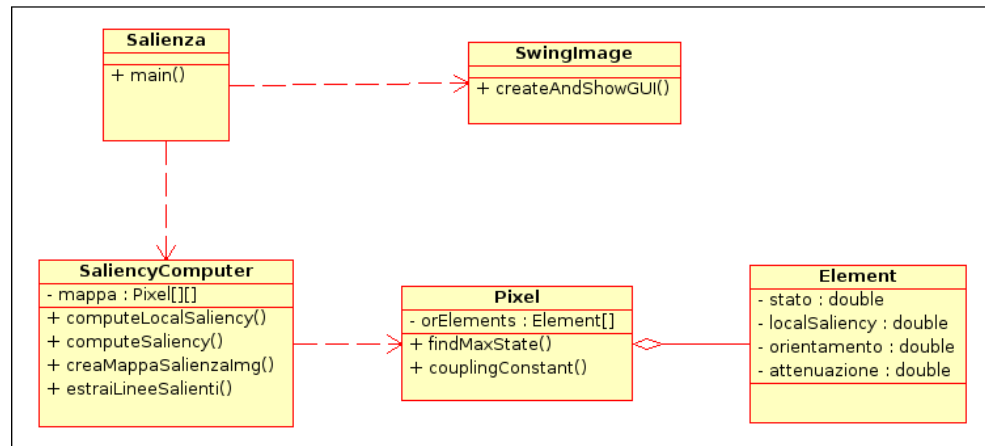


Figura 4.1: Diagramma delle classi

I parametri da tenere in considerazione sono:

- *fileInput*: l'immagine da processare. L'applicazione processa correttamente file .png, .gif e .bmp con immagini in bianco e nero, in modalità RGB. Per ottenere immagini di questo tipo a partire da immagini reali si possono utilizzare algoritmi di *edge detection*
- *numero\_iterazioni*: un buon risultato si ottiene con valori compresi tra 10 e 20
- *soglia\_salienza*: stabilisce il valore di soglia affinché un elemento venga rappresentato nella visualizzazione della Mappa di Salienza. Deve avere un valore compreso tra 0.0 (=0% del max) e 1.0 (=100% del max). 0.6 è un buon valore
- *-gui* è un parametro facoltativo: se presente viene avviata una interfaccia grafica che visualizza l'immagine di input e quelle di output (Mappa di Salienza e linee salienti estratte) per un confronto visivo

L' **output** della computazione della Mappa di Salienza viene salvato nella stessa directory in cui viene lanciato il programma, sotto il nome di *output.png*. Nella medesima posizione viene collocato anche l'output del procedimento di estrazione delle linee più salienti, con il nome di *outputSalienti.png*.

Per immagini di grandi dimensioni (oltre 400x400 pixel da quanto si evince dai test) utilizzare i parametri `-Xms` (dimensione allocazione memoria iniziale) e `-Xmx` (dimensione allocazione memoria massima) per poter allocare più memoria nella JVM. Esempio:

```
java -Xms10m -Xmx80m Salienza img/cerchi400.png 8 0.6 -gui
```

Un esempio di esecuzione è:

```
java Salienza img/cerchi300.png 10 0.6 -gui
```

il cui output è rappresentato in Fig. 4.2.

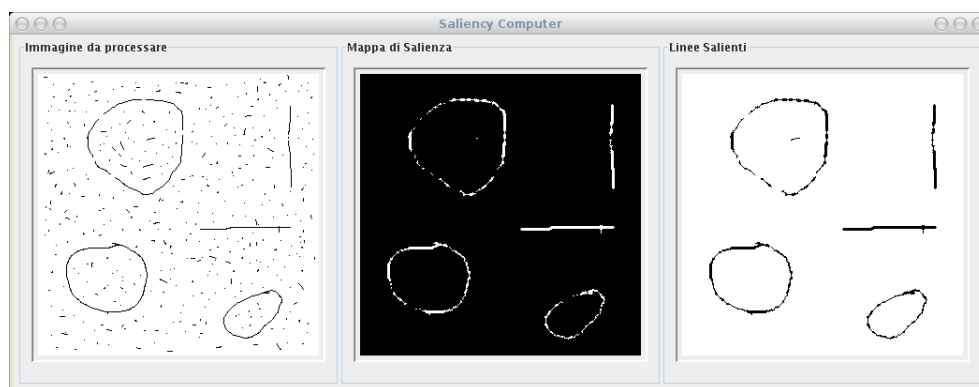


Figura 4.2: Esecuzione con interfaccia grafica

Come si può notare il rumore presente sullo sfondo dell'immagine da processare (a sinistra) scompare quasi totalmente nella Mappa di Salienza (al centro), nella quale vengono riportati a schermo soltanto gli elementi con valore di salienza superiore ad una certa soglia (*soglia\_salienza*). Le linee rette (orizzontali e verticali) ottengono valori di Salienza molto elevati, ed appaiono dunque particolarmente marcate in output.

Variando il valore della soglia ed il numero di iterazioni si ottengono risultati diversi, con output più o meno affetti da rumore ed elementi salienti più o meno marcati. In particolare il numero di iterazioni accentua le differenze nella mappa di salienza tra elementi molto salienti ed elementi poco salienti, mettendo i primi maggiormente in rilievo rispetto ai secondi. Di conseguenza variando il valore di soglia è possibile ottenere output più o meno "filtrati".

## 4.4 Applicazione ad immagini reali

Di seguito vengono mostrate delle esecuzioni applicate ad immagini reali, pre-elaborate tramite un algoritmo di *edge-detection*.

```
java Salienza img/Salienza3.png 10 0.8 -gui &
```

Nel caso specifico della fig. 4.3 le linee rette verticali sulla sinistra ottengono valori di salienza molto elevati (l' algoritmo di Sha'ashua-Ullman premia linee rette e lunghe, soprattutto se parallele agli assi cartesiani). Nella mappa di salienza viene comunque eliminato parte del rumore di fondo dell' immagine di partenza.

Il soggetto al centro dell' immagine (un uomo con la chitarra) viene poi in parte ricostruito nella fase di estrazione delle linee più salienti tramite le informazioni estrapolate dalla Mappa di Salienza.

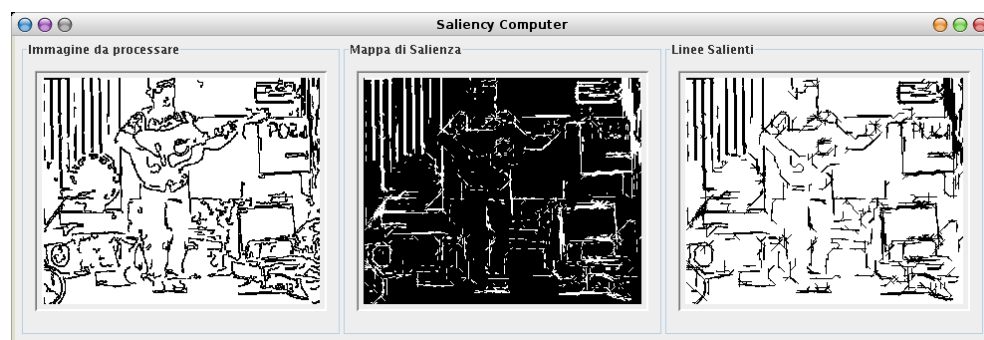


Figura 4.3: Esecuzione su immagine reale

Nella Fig. 4.4 viene invece elaborata la stessa immagine in presenza di minor informazione visiva (e minor rumore). In questo caso la Mappa di Salienza appare più chiara, e la ricostruzione è addirittura in diversi punti migliore. Questo è dovuto alla minore influenza delle linee verticali sulla sinistra dell' immagine, che, nella prima elaborazione, essendo più marcate, influenzavano maggiormente la Mappa di Salienza, accentuando le differenze sui valori di salienza con gli altri elementi della figura e distogliendo l' attenzione dal soggetto principale. Il soggetto principale presenta linee maggiormente continue e precise, tranne nei punti con minor informazione (assente ad esempio nella zona delle gambe).

```
java Salienza img/Salienza1.png 10 0.8 -gui &
```

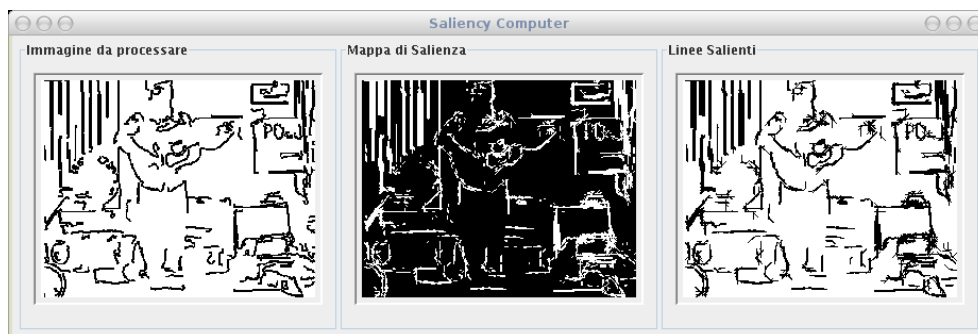


Figura 4.4: Esecuzione su immagine reale

Infine in Fig. 4.5 la stessa immagine dell' esempio precedente viene elaborata attraverso un numero maggiore di iterazioni:

```
java Salienza img/Salienza1.png 15 0.6 -gui &
```

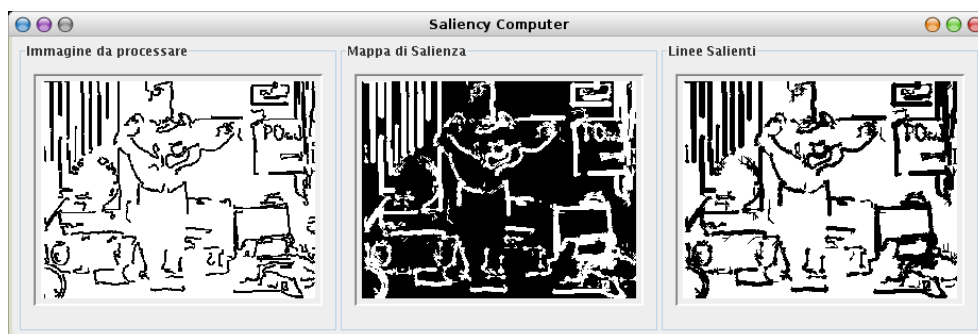


Figura 4.5: Esecuzione su immagine reale

Il maggiore numero di iterazioni ha messo maggiormente in evidenza quelle che secondo l' algoritmo sono le linee più salienti.

## 4.5 Tempi di esecuzione

Vengono qui presentati alcuni tempi di esecuzione in funzione del numero di iterazioni effettuate. Il valore di soglia non influisce sulla durata dell' esecuzione, ma viene riportato per completezza, come valore ottimale per

l'elaborazione corrente. I test sono stati effettuati su una macchina con processore Intel Centrino 1600 Mhz.

In riferimento alle immagini di figura 4.4 e 4.5 (258 x 204 pixel) i tempi misurati sono:

<b>Iterazioni</b>	<b>Soglia</b>	<b>Tempo Medio</b>
10	0.8	18.3 sec.
15	0.6	24.9 sec.
20	0.5	34.9 sec.
30	0.5	52.7 sec.

Tabella 4.1: Tempi di esecuzione per una immagine di 258 x 204 pixel

Nella tabella 4.2 vengono invece riportati i tempi di esecuzione per una immagine di test di dimensioni maggiori: 400 x 400 pixel.

<b>Iterazioni</b>	<b>Soglia</b>	<b>Tempo Medio</b>
10	0.8	58.5 sec.
15	0.6	1 min. 26.4 sec.
20	0.5	1 min. 52.4 sec.

Tabella 4.2: Tempi di esecuzione per una immagine di 400 x 400 pixel

Come si può notare il tempo di esecuzione cresce sensibilmente all'aumentare della dimensione dell' input (numero di pixel dell' immagine).

## 4.6 Considerazioni

L' algoritmo di Sha'ashua - Ullman consente di ottenere ottimi risultati qualitativi per il riconoscimento di linee salienti nelle immagini. Da quanto sperimentato però non è altrettanto performante per quanto riguarda la velocità di esecuzione: ciò è dovuto principalmente alla complessità dell' algoritmo. Se, infatti, è vero che l' algoritmo in questione presenta una soluzione semplificata al problema di ottimizzazione del valore di Salienza lungo ogni curva appartenente all' immagine, riducendo la complessità inizialmente esponenziale (approccio con enumerazione esaustiva di tutte le combinazioni di elementi), è pur vero che, al crescere delle dimensioni dell' input (numero di pixel dell' immagine), l' onere computazionale cresce di conseguenza. Supponiamo di avere un numero  $p$  di pixel nell' immagine, e  $b$  elementi di orientamento per ogni pixel. Il numero totale di elementi sarà dunque  $pb$ . Ad ogni iterazione ogni elemento deve valutare la salienza di tutti gli elementi del pixel ad esso connesso: la complessità di una singola iterazione sarà dunque  $pb^2$ .

Un altro fattore che sicuramente influenza le prestazioni, nel caso specifico, è l' utilizzo del linguaggio Java, che, se da un lato ha semplificato l' implementazione grazie all' approccio ad oggetti, d' altra parte ha penalizzato la velocità di esecuzione su calcoli reiterati di grossa mole.



# Bibliografia

- [SU88] Sha'ashua, Ullman “**Structural Saliency: The Detection of Globally Salient Structures Using a Locally Connected Network**”
- [AB98] Alter, Basri “**Extracting Salient Curves from Images: An Analysis of the Saliency Network**”