

Selección eficiente de candidatos a cara mediante detección de ojos

Procesamiento de Imágenes Digitales (Topología Digital)

9 de Febrero de 2004

Pontes Balanza, Beatriz
(cuaqui@supercable.es)
Resinas Arias de Reyna, Manuel
(resman@ieee.org)

Índice de contenido

Introducción.....	1
Descripción del algoritmo.....	3
Conversión a escala de grises.....	4
Ocho vecinos.....	4
Eliminación de componentes conexas grandes.....	5
Eliminación de píxeles dispersos.....	5
Ampliación de regiones candidatas a ojos.....	6
Eliminación de falsos candidatos a ojos.....	6
Detección de posibles pares de ojos.....	7
Preparación de la imagen para el verificador de caras.....	7
Manual de usuario.....	9
Requisitos mínimos.....	9
Instalación y ejecución.....	9
Funcionalidad.....	9
Cargar una Imagen.....	9
Elección del Modo de Funcionamiento.....	10
Ejecución Continua: Run All.....	10
Ejecución Paso a Paso: Step By Step.....	11
Implementación.....	17
Arquitectura de la aplicación.....	17
Interfaz de usuario.....	17
Selector de candidatos a cara.....	18
Control del algoritmo: TransformSequence.....	18
Operador average: AverageOp.....	18
Cálculo de componentes conexas: ConnectedPixelsOp.....	19
Conversión a escala de grises: GrayScale.....	20
Ocho vecinos: EightNeighOp.....	20
Eliminación de componentes conexas grandes: RemoveLargeOp.....	20
Eliminación de píxeles dispersos: RemoveSparseOp.....	21
Ampliación de regiones candidatas a ojos: MarkAnalogOp.....	21
Eliminación de falsos candidatos a ojos: RemoveNotEyesOp.....	21
Detección de posibles pares de ojos: PairOfEyesOp.....	22
Preparación de la imagen para el verificador de caras: FinalMaskOp.....	22
El verificador de caras.....	22
Mejoras desarrolladas.....	23
Resultados y conclusiones.....	24
Limitaciones.....	25
Planificación del trabajo.....	26
Bibliografía.....	28

Introducción

Supongamos que te sientas delante de tu ordenador y, sin necesidad de hacer nada, él es capaz de identificarte y abrir tu sesión para que te pongas a trabajar inmediatamente. Imaginemos que se le entrega al ordenador una fotografía y automáticamente te devuelve los nombres de las personas que aparecen o que se implanta un sistema en un aeropuerto de forma que analiza las imágenes de las cámaras de seguridad y alerta si en alguna de ellas aparece un individuo buscado por la policía.

Las aplicaciones del reconocimiento automático de caras son muchas y muy variadas. Por eso, es un campo de investigación que ha atraído una atención importante durante las últimas décadas en ámbitos como el análisis de imágenes, la visión por computador y el reconocimiento de patrones.

Para el ser humano resulta muy fácil reconocer y distinguir una cara dentro de una imagen, da igual las condiciones de luz, tipo de fondo, etc. Sin embargo, para una máquina no es algo sencillo en absoluto debido a la gran variedad de expresiones, edad, posiciones (de perfil, de frente), inclinaciones, etc. que puede presentar la cara de una persona. Además, todo esto se complica si la cara está parcialmente oculta, lleva una máscara o está más inclinada de lo habitual.

En líneas generales los algoritmos de reconocimiento de caras pueden dividirse en dos categorías, aunque, obviamente, existen algunas excepciones. Estas dos categorías son el reconocimiento por características de rasgos faciales como pueden ser ojos, boca, cejas o nariz y el reconocimiento basado en la comparación de la imagen con plantillas de caras.

Se puede observar que las dos vertientes presentan un elemento común: la necesidad de, dada una imagen, detectar caras y ojos. La razón de esto es sencilla: para el primer tipo de algoritmos, los ojos son una de las características fundamentales para lograr el reconocimiento de una cara pues presentan particularidades únicas para cada individuo. Para el segundo tipo de algoritmos, la mayoría de sistemas de reconocimiento basado en comparación con plantillas requieren que la imagen de entrada haya sido alineada previamente y, la forma más habitual de hacerlo es mediante los ojos. Por tanto, la correcta detección de caras y ojos va a ser fundamental para lograr un posterior reconocimiento facial efectivo y será la parte en la que nos centremos en este trabajo.

Si profundizamos un poco en los distintos algoritmos para lograr esta detección de caras nos podemos dar cuenta de que, en la mayoría de ellos, se sigue un proceso en dos fases: la primera fase consiste en detectar regiones que sean posibles candidatos a caras y la segunda, en verificar cuál de los candidatos a caras encontrados en la primera etapa son realmente caras.

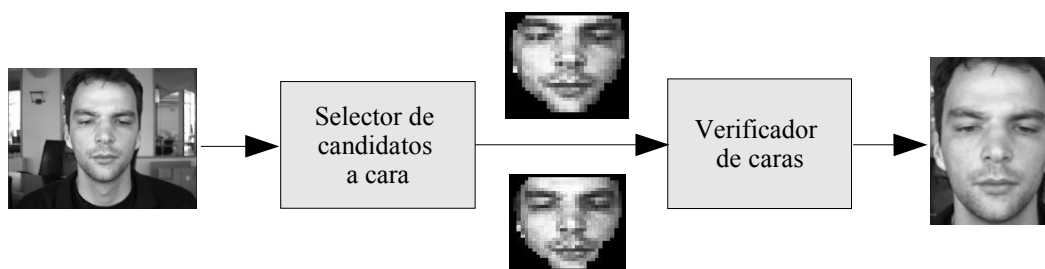


Figura 1. Etapas en la detección de caras.

Los distintos métodos para la detección de caras ponen más énfasis en una fase o en otra. Así, si tenemos un selector de candidatos a cara muy bueno, el verificador de caras podría ser extremadamente simple o, ni tan siquiera existir. En el caso opuesto tendríamos un sistema en

el que el verificador de caras fuese prácticamente perfecto. En esas circunstancias, no sería necesario utilizar ningún selector de candidatos, se podría ir aplicando el verificador a toda la imagen trozo a trozo para encontrar todas las posibles caras allí existentes.

Nuestro trabajo consistía en la implementación de un algoritmo que sirva para realizar de una forma eficiente la primera de las etapas de la detección de caras. Este documento está organizado de la siguiente manera. Tras esta introducción general a los sistemas de reconocimiento de caras, necesaria para poder establecer el ámbito que cubre la implementación realizada, pasaremos a realizar una descripción del algoritmo que hemos seguido. A continuación, incluiremos un manual de usuario de la aplicación que se ha desarrollado explicando la forma de utilizarse y los distintos pasos necesarios para su instalación. Tras esto, analizaremos distintos detalles de implementación y algunos añadidos incorporados al algoritmo. Finalmente, comentaremos los resultados obtenidos con la aplicación y algunas conclusiones a las que hemos llegado. Para terminar, incluiremos una hoja de tiempos y la bibliografía empleada para la realización del trabajo.

Descripción del algoritmo

Como hemos comentado, en este trabajo hemos realizado la implementación de un selector de candidatos a caras que forma parte de las fases preliminares para la realización de un reconocedor completo de caras. Además, con el fin de obtener resultados más completos con la aplicación, hemos desarrollado también un verificador de caras muy simple que elige una cara entre todos los candidatos encontrados por el selector de candidatos a caras.

El algoritmo que hemos utilizado está descrito en el artículo *Efficient face candidates selector for face detection*, de Jianxin Wu y Zhi-Hua Zhou, publicado en el volumen 36 de *Pattern Recognition* (Elsevier).

El algoritmo se basa en el hecho de que la zona de los ojos es más oscura que otras zonas de la cara debido a que está ligeramente hundida y tiene pestañas e iris que son, en general, de tonos más oscuros que las zonas de alrededor. Utilizando esta idea, se irá recorriendo la imagen en busca de regiones del tamaño de ojos que sean más oscuras que regiones de alrededor. Esos serán nuestros candidatos a ojos. Una vez encontrados los ojos, se aplicará una plantilla para obtener el candidato a cara.

Antes de comenzar a comentar el algoritmo con mayor profundidad es necesario hacer unas consideraciones iniciales. En primer lugar, el algoritmo sólo funciona bien con caras frontales, es decir, no es capaz de reconocer caras que estén de perfil o que presenten la cabeza girada en un ángulo demasiado grande. Otra restricción que presenta el algoritmo es que la inclinación de la cabeza con respecto a la horizontal tampoco debe ser demasiado elevada, esto es, tampoco sería capaz de reconocer a personas tumbadas en una cama, por ejemplo. El tercer aspecto que hay que considerar es que el algoritmo trabaja con un tamaño prefijado de ojo (que nosotros consideraremos que es $h_e \times w_e$). Para detectar ojos de otros tamaños, simplemente es necesario reescalar la imagen o bien, considerar un h_e y w_e diferentes (que es la opción elegida en nuestra implementación).

Tal y como se ha mencionado, el algoritmo se basa en la detección de zonas más oscuras que otras en la imagen de un tamaño similar al ojo humano. Para la detección de estas zonas oscuras se utiliza el operador *average*¹: sea $P(x,y)$ un punto de una imagen de tamaño $N_1 \times N_2$, donde $x \in [1, N_1]$, $y \in [1, N_2]$, $P(x,y) \in [0,1]$ siendo x la fila e y la columna, se define el operador $avg(P, x, y, h, w)$ como la intensidad media del trozo de imagen cuya esquina superior derecha es (x,y) y su tamaño es $h \times w$. Esto es:

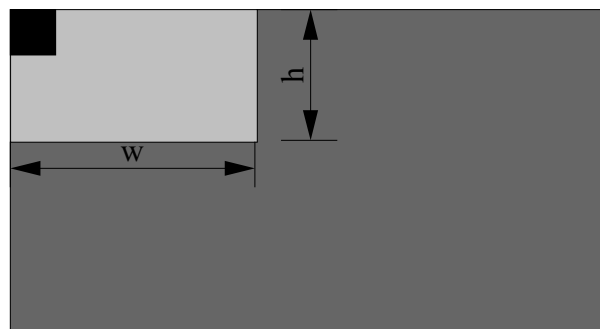


Figura 2. Operador average

De esta forma, se puede conocer si un píxel es más oscuro que sus vecinos: si su intensidad es superior al resultado del operador *average* para un determinado tamaño, se puede decir que ese píxel es más oscuro que todos los que la mayoría de los que hay a su alrededor.

¹ Hemos decidido mantener el nombre inglés para evitar confusiones con la media aritmética y que sea fácilmente reconocible al leer literatura inglesa.

El algoritmo está compuesto por dos etapas principales: en la primera se realiza una búsqueda de regiones análogas a ojos. Durante esta búsqueda, se realizará en primer lugar un marcado de píxeles análogos a ojos y, posteriormente, se realiza un procesamiento de estos píxeles para eliminar aquellos que están muy dispersos, conectar píxeles que están cercanos y eliminar regiones que no cumplan las características geométricas (relaciones altura y anchura) de los ojos humanos.

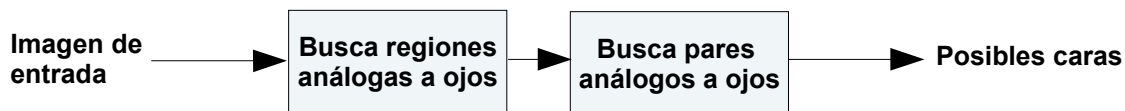


Figura 3. Diagrama de bloques para el selector de candidatos.

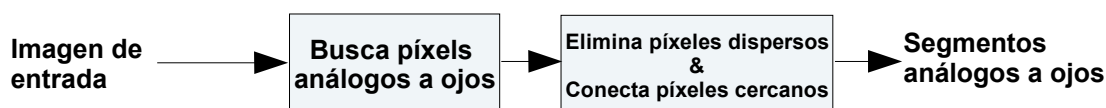


Figura 4. Diagrama de bloques para la búsqueda de regiones candidatas a ojos.

La segunda etapa consiste en la búsqueda de pares análogos a ojos, es decir, de pares de segmentos obtenidos en la etapa anterior que cumpla unas determinadas restricciones de distancia, inclinación, etc. y de preparar la cara para pasarla posteriormente al verificador de caras. Este preparado consiste en un reescalado, rotación y la aplicación de una máscara para eliminar partes que no interesan de la imagen.

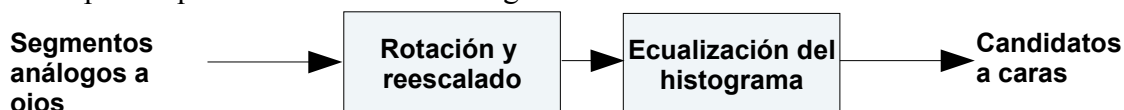


Figura 5. Diagrama de bloques para la detección de candidatos a caras a partir de segmentos análogos a ojos.

El algoritmo completo se puede descomponer en ocho pasos, los seis primeros pertenecientes a la primera etapa y los dos últimos a la segunda. A continuación vamos a ir comentando cada uno de estos ocho pasos detalladamente.

Conversión a escala de grises

El algoritmo requiere que la imagen de entrada se encuentre en escala de grises, por tanto, es necesario realizar este paso previo antes de comenzar con el tratamiento de la imagen propiamente dicho. La conversión a escala de grises no supone ninguna limitación pues convertir una imagen en colores a escala de grises es trivial. Por tanto, el algoritmo sirve también para imágenes en color, el único inconveniente es que en éstas hay información adicional que se pierde al convertir a escala de grises y un algoritmo para tratar imágenes en color podría aprovechar para procesarlas más eficientemente.

Ocho vecinos

Como hemos comentado, el algoritmo se basa en que los ojos son regiones más oscuras que el resto de la cara, por tanto, el primer paso consiste en buscar esas regiones que sean análogas a ojos. Para esto, consideraremos que un píxel pertenece a una de estas regiones (diremos que es un píxel análogo a ojo) si es más oscuro que la mayoría de sus vecinos. Concretamente, si se cumplen seis o más de estas restricciones:

Como se puede observar, utilizamos el operador *average* que definimos anteriormente para establecer si un píxel es más oscuro que sus vecinos. Recordamos también que h_e y w_e son el tamaño de ojos prefijado.

$$\begin{aligned}
 P(x, y) &< 0.9 * \text{avg}(P, x - \lfloor h_e/2 \rfloor, y - \lfloor w_e/2 \rfloor, \lfloor h_e/2 \rfloor, \lfloor w_e/2 \rfloor) \\
 P(x, y) &< 0.9 * \text{avg}(P, x - \lfloor h_e/2 \rfloor, y, \lfloor h_e/2 \rfloor, 1) \\
 P(x, y) &< 0.9 * \text{avg}(P, x - \lfloor h_e/2 \rfloor, y + 1, \lfloor h_e/2 \rfloor, \lfloor w_e/2 \rfloor) \\
 P(x, y) &< 0.9 * \text{avg}(P, x, y - \lfloor w_e/2 \rfloor, 1, \lfloor w_e/2 \rfloor) \\
 P(x, y) &< 0.9 * \text{avg}(P, x, y + 1, 1, \lfloor w_e/2 \rfloor) \\
 P(x, y) &< 0.9 * \text{avg}(P, x + 1, y - \lfloor w_e/2 \rfloor, \lfloor h_e/2 \rfloor, \lfloor w_e/2 \rfloor) \\
 P(x, y) &< 0.9 * \text{avg}(P, x + 1, y, \lfloor h_e/2 \rfloor, 1) \\
 P(x, y) &< 0.9 * \text{avg}(P, x + 1, y + 1, \lfloor h_e/2 \rfloor, \lfloor w_e/2 \rfloor)
 \end{aligned}$$

Gráficamente, resultaría algo así:

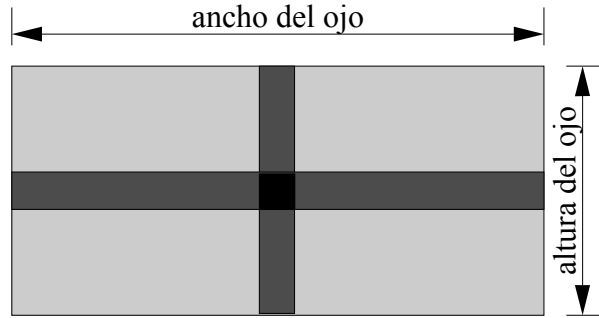


Figura 6. Un píxel y las regiones de los ocho vecinos con las que compara.

La imagen resultante es una imagen binaria en la que los píxeles candidatos a ojos están en blanco y el resto de la imagen es negro.

Eliminación de componentes conexas grandes

Una vez hemos obtenido en el paso anterior el conjunto de píxeles análogos a ojos, el resto de los pasos del 2 al 6 del algoritmo consiste en ir refinando este conjunto de píxeles. En primer lugar, detectaremos agrupaciones de píxeles con un tamaño demasiado grande para ser una región candidata a ojo y las eliminaremos. Para eso, buscaremos las componentes conexas de la imagen obtenida en el paso anterior y eliminaremos aquellas cuyo rectángulo exterior sea más alto que $2h_e$ o más ancho que $2w_e$.

Eliminación de píxeles dispersos

Casi todos los píxeles entre las pestañas son análogos a ojos, esto quiere decir que en una imagen los píxeles análogos a ojos estarán cerca de otros píxeles análogos a ojos, de manera que aquellos que estén demasiado aislados no serán de posibles ojos y se eliminarán.

La forma de conseguir esto es acudiendo de nuevo al operador *average*. Trabajamos ahora sobre la imagen obtenida en el paso anterior y borraremos un píxel previamente marcado como análogo a ojo si se cumple:

$$\text{avg}(P', x - \lfloor h_e/2 \rfloor, y - \lfloor w_e/2 \rfloor, h_e, w_e) \leq 0.2$$

Recordemos que los píxeles candidatos a ojo han sido marcados de color blanco, mientras que el resto de la imagen es negra, por tanto, si la mayoría de los píxeles que rodean a uno dado en un área del tamaño del ojo son oscuros en su mayoría quiere decir que ese píxel está aislado y se desmarcará como candidato a ojo.

Ampliación de regiones candidatas a ojos

Las pestañas, pupila y lagrimal son zonas más oscuras que el resto del ojo, por tanto, es

probable que hayan quedado zonas en el ojo en la que los píxeles no hayan sido marcados como candidatos a ojos. Para solucionarlo, marcaremos como píxeles análogos a ojos todos aquellos que tengan un número suficiente de píxeles análogos a ojos a su alrededor.

Para esto, recurrimos al operador *average*, de forma que un píxel será marcado como píxel análogo a ojo si se cumple:

$$\text{avg}(P', x - \lfloor h_e/4 \rfloor, y - \lfloor w_e/4 \rfloor, \lfloor h_e/2 \rfloor, \lfloor w_e/2 \rfloor) > 0.35$$

Eliminación de falsos candidatos a ojos

El último paso de la primera etapa consiste en eliminar aquellos grupos de píxeles que son demasiado grandes o demasiado pequeños para ser ojos. Para esto, volvemos a obtener las componentes conexas de la imagen y eliminaremos aquellas cuyo rectángulo exterior no cumpla alguna de estas restricciones:

$$\begin{aligned} h/h_e &< 0.5 \text{ ó } h/h_e > 2.0 \\ w/w_e &< 0.5 \text{ ó } w/w_e > 2.0 \\ h/w &> 0.8 \end{aligned}$$

donde h y w son la altura y anchura del rectángulo exterior de la componente conexa y h_e y w_e son la altura y la anchura del ojo que estamos buscando.

Con las dos primeras restricciones descartamos ojos que son demasiado pequeños o demasiado grandes. Se le da un amplio margen para evitar perder posibles caras ya que la selección final se realizará en el verificador de caras. Por otro lado, la tercera ecuación se asegura de que el posible ojo tiene unas proporciones apropiadas.

Al finalizar este paso lo que obtenemos es una lista de regiones candidatas a ojos, ahora sólo falta ver cuáles de ellas son posibles ojos de caras.

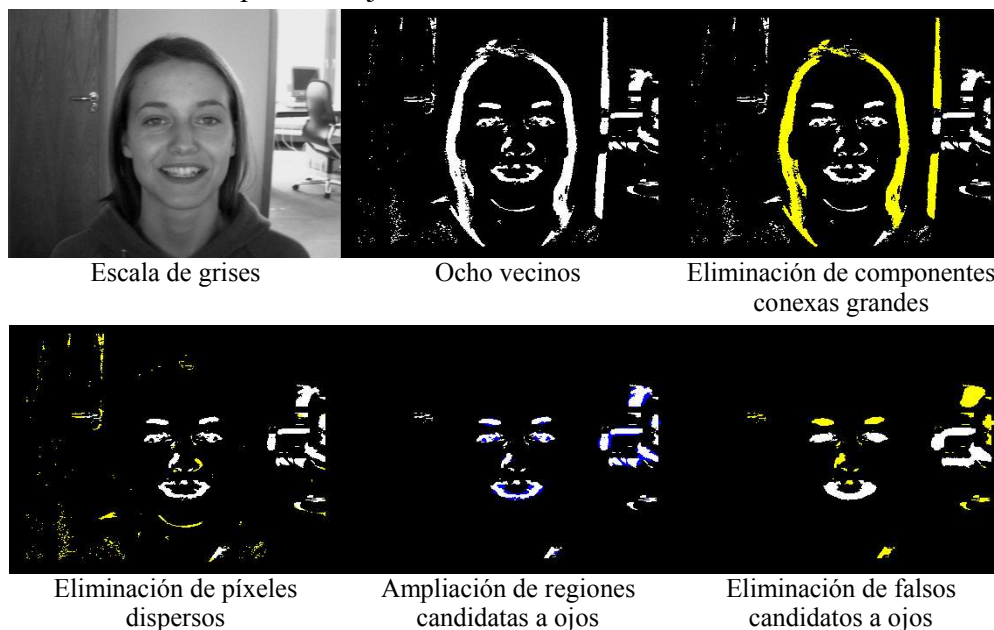


Figura 7. Pasos de la primera etapa de búsqueda de candidatos.

Detección de posibles pares de ojos

Tras finalizar el paso anterior, ya hemos encontrado todas las regiones candidatas a ojos. El paso que tenemos que hacer ahora es buscar pares de esas regiones con características similares a las de dos ojos humanos. Para esto, nos basaremos en las distancias existentes

entre los centros de las dos regiones análogas a ojos.

De este modo, si (x_i, y_i) y (x_j, y_j) son los centros de las dos regiones análogas a ojos que estamos considerando, existirá un candidato a cara si se verifican las siguientes restricciones:

$$\begin{aligned}d_{ij} &< 2.5 \cdot w_e \\d_{ij} &> 1.5 \cdot w_e \\|x_i - x_j| &< h_e\end{aligned}$$

en donde d_{ij} es la distancia entre los dos centros:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Los valores de las ecuaciones se han elegido teniendo en cuenta que, por media, $d_{ij} = 2w_e$

Al finalizar este paso, obtenemos los ojos de todos los candidatos a caras que el algoritmo pasará al verificador.

Preparación de la imagen para el verificador de caras

El verificador requiere que las imágenes cumplan una serie de propiedades en cuanto a tamaño, luminosidad y orientación. Por lo tanto, es necesario realizar un procesamiento previo de los distintos candidatos a caras encontrados en el paso anterior antes de pasárselas al verificador. Este procesamiento consiste en lo siguiente:

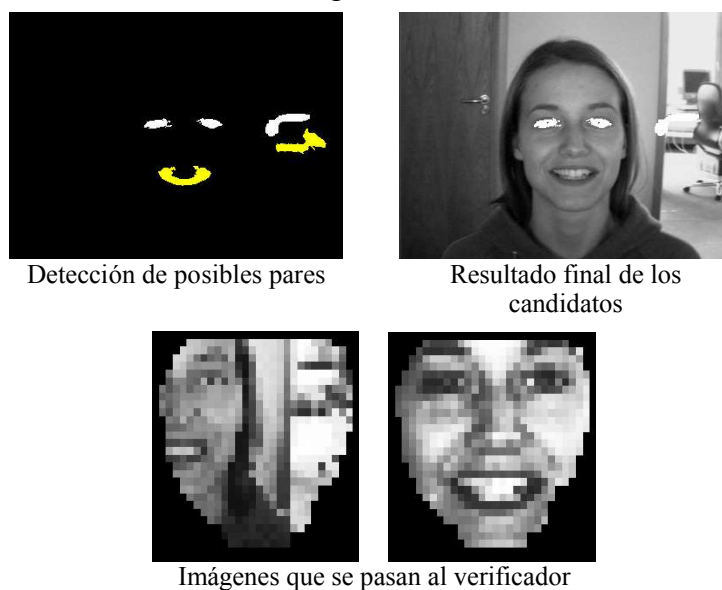


Figura 8. Pasos de la segunda etapa de búsqueda de candidatos.

1. **Rotamos** la imagen para corregir ligeras inclinaciones de la posible cara. Para esto, utilizamos los ojos como guía: se supone que los dos centros han de estar alineados, por lo tanto, el ángulo de rotación que hay que corregir viene dado por:

$$\theta = \arctan \frac{x_i - x_j}{y_i - y_j}$$

2. **Recortamos y reescalamos** la posible cara a un tamaño normalizado. Un patrón de imagen normalizado es de tamaño 30 x 30 y los centros de los segmentos análogos a ojos deben tomar las posiciones (6,8) y (6,22) respectivamente.
3. **Aplicamos una máscara** para eliminar algunos píxeles cercanos a los bordes.

4. Ecualizamos el histograma para suavizar cambios de iluminación de la imagen original.

Una vez realizados este procesado ya podemos pasar las distintas posibles caras al verificador para que compruebe si lo son realmente o no.

Para detectar caras de distintos tamaños, la imagen de entrada se puede escalar repetidamente por un factor de 1.2, 7 veces. En cada escalado, todos los candidatos son marcados y verificados por el verificador de caras.

Otra alternativa posible es ir buscando ojos de tamaño más reducido en lugar de reducir la imagen. En nuestra implementación, hemos usado esta última opción y, además, hemos establecido que sólo continuará buscando si no se ha encontrado ningún candidato a cara en la iteración anterior, de esta forma conseguimos evitar el problema de detectar dos veces la misma cara.

Manual de usuario

Esta sección es un manual de usuario de la aplicación. Empezaremos viendo los requisitos mínimos del sistema para poder ejecutar la aplicación, luego veremos la forma en que se ejecuta y, finalmente, una guía de utilización.

Requisitos mínimos

La aplicación ha sido desarrollada en Java lo que permite que se pueda ejecutar en cualquier sistema operativo para el que exista una implementación de la máquina virtual de Java. Por tanto, para la ejecución del programa es necesario:

- **Máquina virtual de Java.** Al menos la **versión 1.4**
- **Java Advanced Imaging API.** API para el manejo de imágenes en Java. Es necesario sólo para poder cargar determinado tipo de archivos. En el CD se incluyen versiones de esta API para sistemas operativos Windows y Linux.

Instalación y ejecución

La aplicación no requiere instalación (salvo la de la máquina virtual de Java en caso de que no se tenga la versión apropiada).

Para ejecutarla, tan sólo se requiere hacer doble click en el icono de la aplicación si se encuentra en Windows o ejecutar el archivo `eyesdetector` en el caso de encontrarse en Linux.



Figura 9. Ejecutable en Windows e instalación de la versión 1.4 de Java.

```
$ ./eyesdetector
```

Figura 10. Ejecución en Linux.

Funcionalidad

La ventana inicial de la aplicación es la que se muestra a continuación:

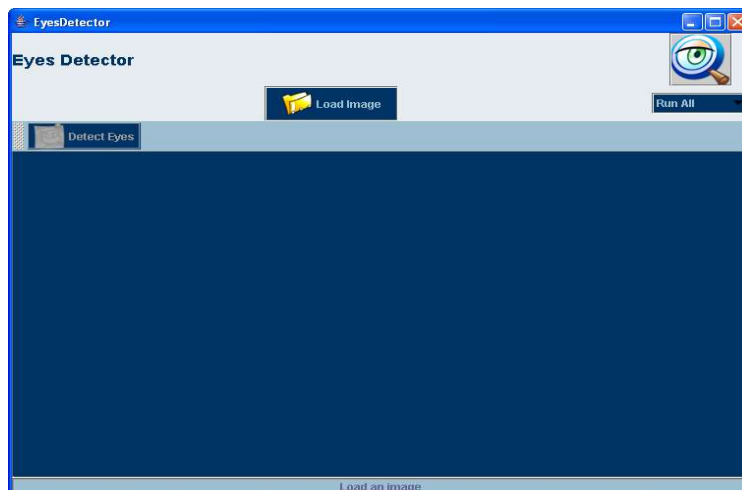


Figura 11. Situación inicial.

Cargar una Imagen

Para abrir una imagen basta con hacer clic en el botón “Load Image” de la parte superior de la aplicación. Se abrirá una ventana del explorador para poder seleccionar la imagen que se quiera procesar:

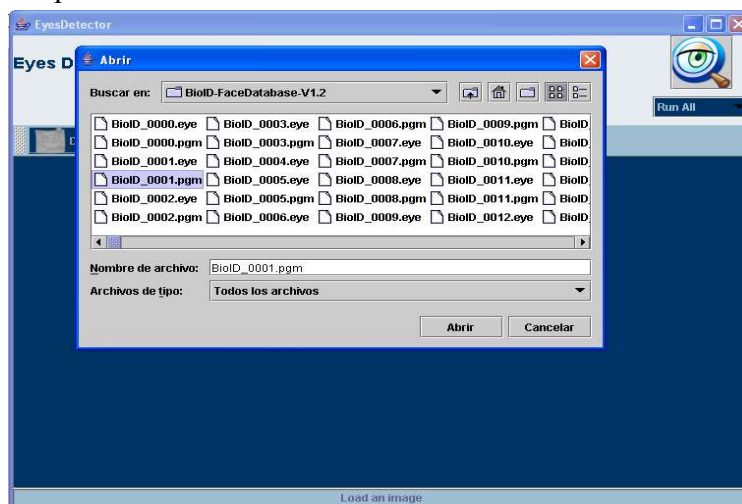


Figura 12. Elección del archivo.

Obtenemos finalmente la imagen cargada:

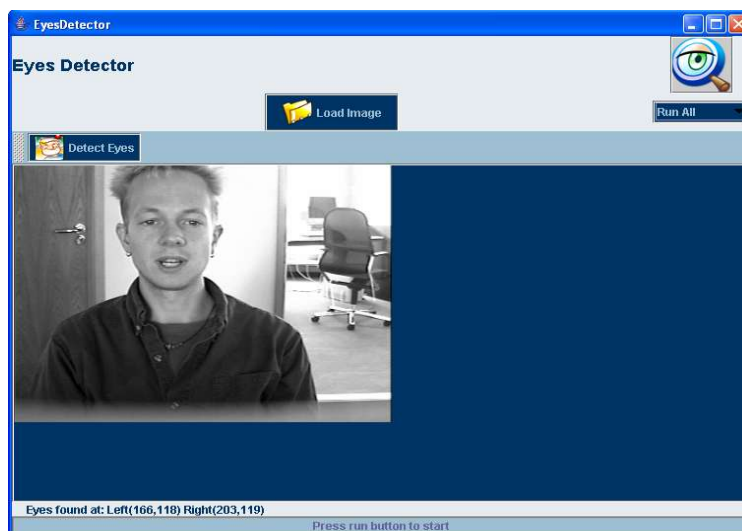


Figura 13. Imagen cargada.

Elección del Modo de Funcionamiento

Existen dos modos de funcionamiento del programa:

- Ejecución continua “Run All”.
- Ejecución paso a paso “Step By Step”.

Se puede seleccionar cualquiera de ellas mediante el menú desplegable que aparece en la parte superior derecha. Se detallan a continuación ambos modos:

Ejecución Continua: *Run All*

En ese primer modo de ejecución solamente podremos contemplar las imágenes inicial y final: esto es, la imagen original que cargamos y la imagen final que obtenemos tras aplicar el algoritmo. Mediante este método siempre encontraremos una solución, sea correcta o no, por lo tanto, el resultado vendrá dado por dos círculos que indican la posición de ambos ojos.

Todo el proceso será transparente al usuario, esto es, la única manera de poder ver la evolución será mediante la **barra de progreso inferior**, que nos mostrará la fase de cada iteración en la que se encuentra, pudiéndose realizar hasta un máximo de 8 iteraciones por ejecución. (Ver Figura 14)

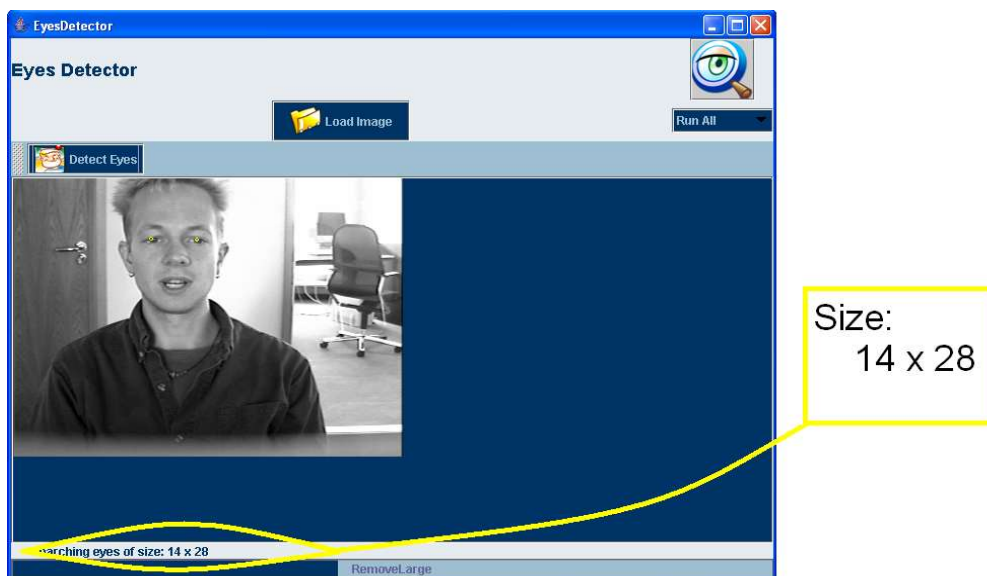


Figura 14. Progreso de ejecución

El **tamaño de ojos** con el que se está trabajando en cada iteración se muestra por encima de la barra de progreso siempre que se esté ejecutando el algoritmo. En el momento en que éste termine podremos conocer en ese mismo espacio la **posición de los ojos** localizados. (Ver Figura 15)

Para comenzar la ejecución basta con hacer clic en el botón “Detect Eyes”. Una vez comenzada no podremos detenerla hasta que ésta finalice:

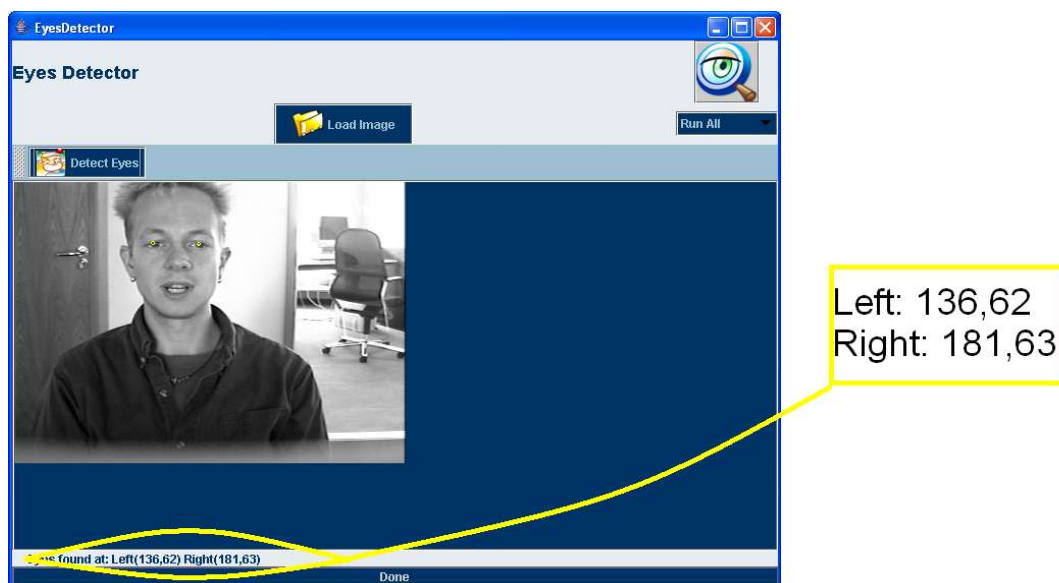


Figura 15. Ejecución finalizada.

Ejecución Paso a Paso: Step By Step

En la siguiente figura se muestra la ventana que aparece al seleccionar “Step By Step” del menú desplegable: (Figura 16).

En ella podemos ver como la imagen ha sido situada en el primero de los nueve posibles lugares. El resto serán utilizados para representar las sucesivas imágenes.

Next Step: Botón situado arriba a la derecha. Pulsando sobre él obtenemos la **siguiente imagen** generada por el algoritmo, dependiendo de la etapa en la que nos encontremos.

- Si no se dispone de una imagen cargada, este botón estará deshabilitado a la espera de una imagen.
- Si disponemos de una imagen, este botón inicialmente indicará el siguiente paso que será aplicado. Inicialmente será *GrayScale*.

Eyes Height/Width: Permiten seleccionar el tamaño de los ojos que van a ser buscados durante el algoritmo. Sus valores por defecto son de 20 y 40 pixels respectivamente. Una vez comenzado el algoritmo no podrán ser modificados hasta que se complete una iteración. Al finalizar podremos modificarlos y repetir el proceso tantas veces como se desee.

En el **panel de texto** situado a la izquierda de las imágenes podremos leer una breve **descripción** de lo que representa la **última imagen** generada en cada fase. Asimismo, la **barra de texto** inferior nos dice cuál ha sido la última imagen generada.

Show Candidates: Este botón solamente estará activo en la fase final de cada iteración que se realice, en el caso de que se haya encontrado como mínimo un candidato.

Haciendo clic sobre cualquiera de las imágenes obtendremos una **ampliación** de la misma, donde se podrán apreciar mejor los detalles.

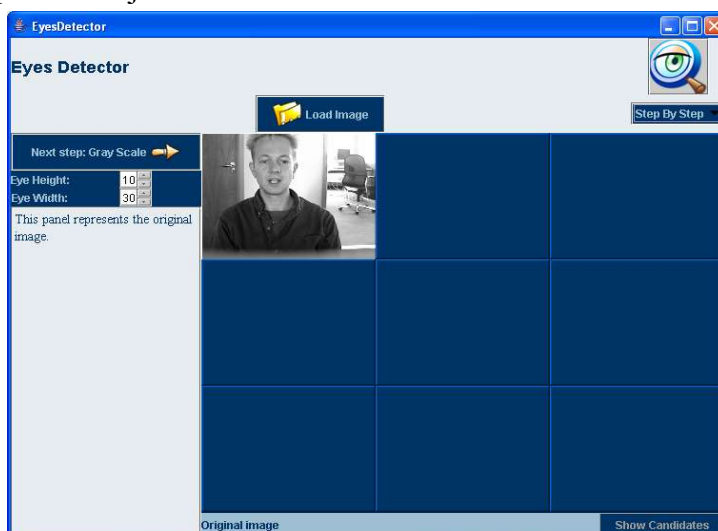


Figura 16. Situación inicial Step By Step.

Como norma general, cada vez que se realice un paso, el resultado aparecerá en el recuadro siguiente, pero esto no es siempre así. Las excepciones son:

- **GrayScale:** La imagen en escala de grises se muestra en el mismo panel que la anterior ya que en realidad esto no constituye un paso del algoritmo. Siempre que la imagen de entrada ya venga dada en escala de grises, no se apreciará cambio alguno en este paso.
- **PairOfEyes:** En este paso se generan dos imágenes con los conjuntos de pares de ojos encontrados. En la primera de ella muestra los candidatos como manchas blancas y en la segunda las superpone a la imagen original.

Se muestra a continuación una secuencia de imágenes para una iteración normal que finaliza de manera exitosa y habiendo encontrado dos candidatos: (Figura 17 a Figura 23).



Figura 17. Ocho adyacencia.

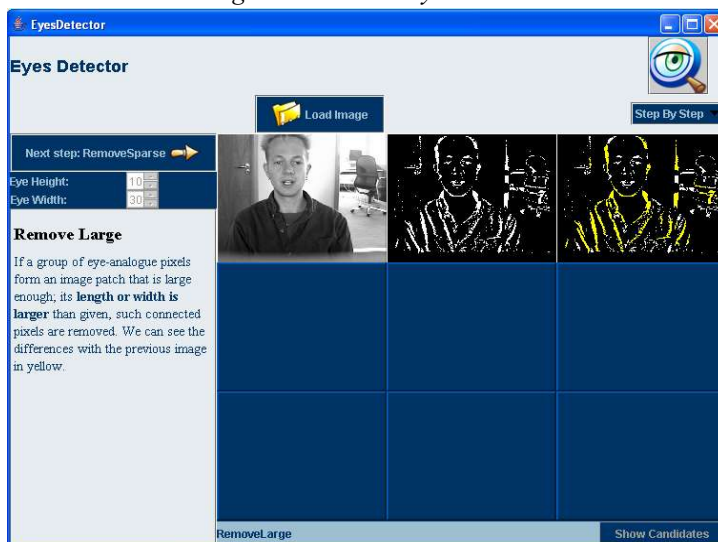


Figura 18. Borra componentes conexas grandes.

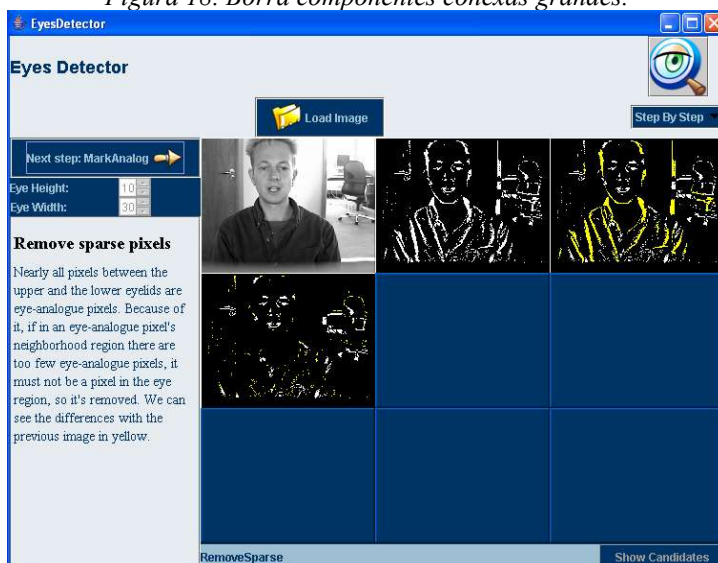


Figura 19. Elimina píxeles dispersos.

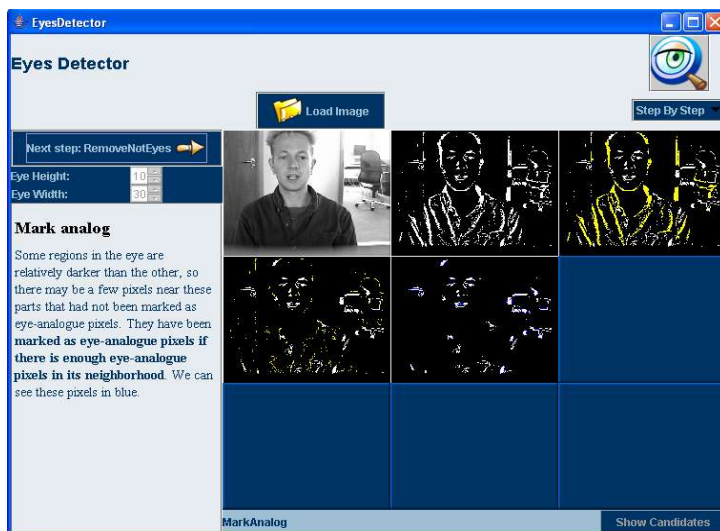


Figura 20. Agranda mejores candidatos a ojos.

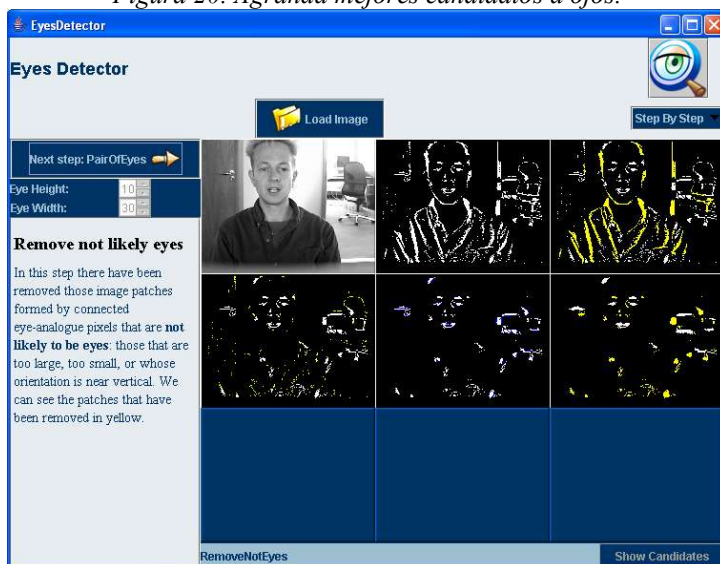


Figura 21. Elimina falsos candidatos a ojos.

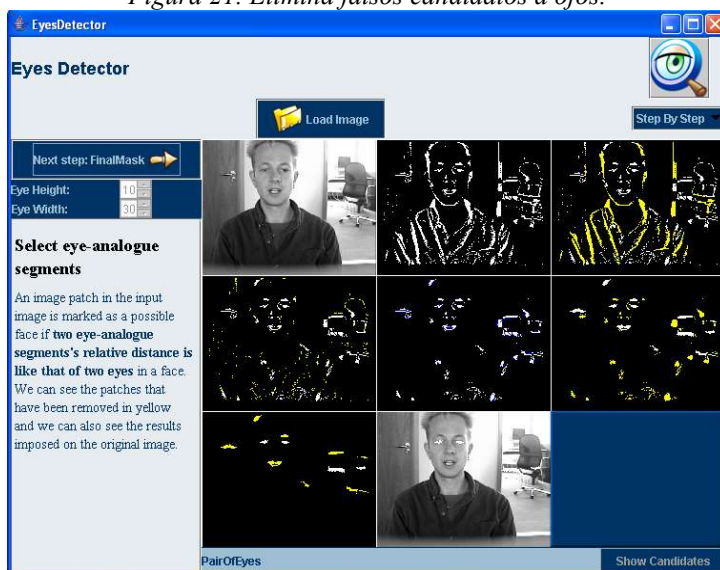


Figura 22. Detecta posibles pares de ojos.

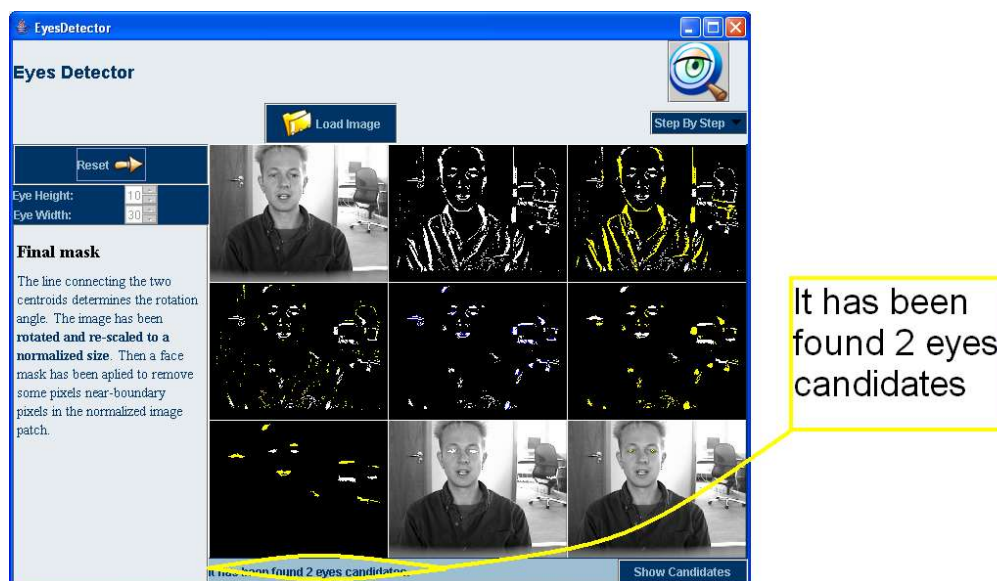


Figura 23. Muestra el resultado final.

Tras una ejecución como la anterior en la que se han encontrado candidatos, el botón *Show Candidates* se activa y haciendo clic sobre él nos aparece una ventana en la que se muestra lo siguiente: (Figura 24 y Figura 25).

Los candidatos que se consideran falsos, y por tanto, no aceptados, se rodean de un marco de color rojo. Por el contrario, cuando un candidato se acepta, según su valor de correlación, éste se recuadra con un marco de color verde.

Mediante los botones *Previous Candidate* y *Next Candidate* podemos recorrer todos los candidatos que hayan sido encontrados para esa iteración.



Figura 24. Un falso candidato.

Para cerrar la aplicación hacer clic sobre el botón para cerrar cualquier ventana.

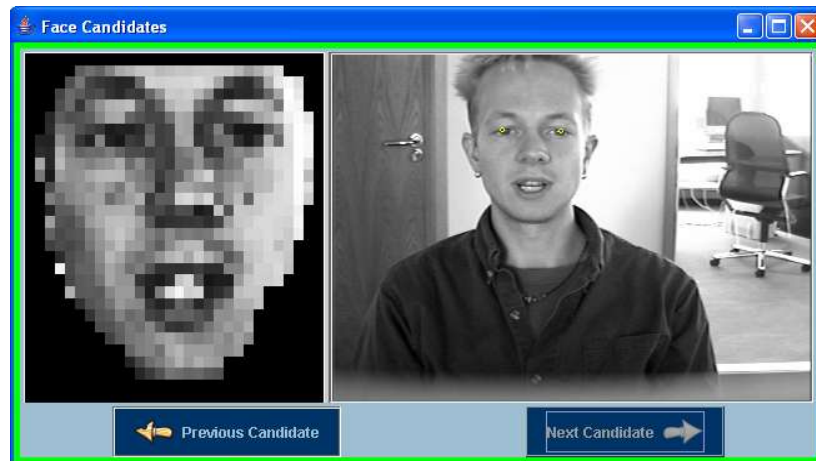


Figura 25. Un candidato aceptado.

Implementación

La aplicación ha sido implementada en Java. Las razones por las que se ha elegido esta plataforma son, fundamentalmente, porque es independiente del sistema operativo sobre el que se ejecuta, de forma que la aplicación puede usarse tanto en Windows como en Linux. La segunda razón es que Java proporciona un conjunto de APIs para poder tratar imágenes de una forma sencilla.

En un principio, para la ejecución del programa sólo sería necesaria la versión 1.4 de la máquina virtual de Java. Sin embargo, esto limitaba los ficheros que podía abrir a sólo tres formatos gráficos distintos. Para superar esta limitación decidimos utilizar una API adicional de *Sun Microsystems* denominada *Java Advanced Imaging*. Gracias a la utilización de esta librería el conjunto de formatos gráficos reconocidos por el sistema aumenta a unos diez.

Además del algoritmo que hemos comentado en una sección anterior, también se ha implementado un sencillo verificador de caras con el fin de demostrar la eficacia del selector de candidatos a caras, pues se consiguen unos resultados bastante aceptables usando un verificador de caras extremadamente simple. Sobre el verificador de caras hablaremos con un poco más en profundidad algo más adelante.

Arquitectura de la aplicación

En la aplicación se pueden distinguir tres partes bien diferenciadas. Por un lado tenemos la parte encargada de la interfaz gráfica de usuario: los botones, mostrar imágenes, etc. Por otra parte, está el código encargado de la implementación del algoritmo de obtención de candidatos a caras. Por último, tenemos la parte del código encargado de la implementación del verificador de caras.

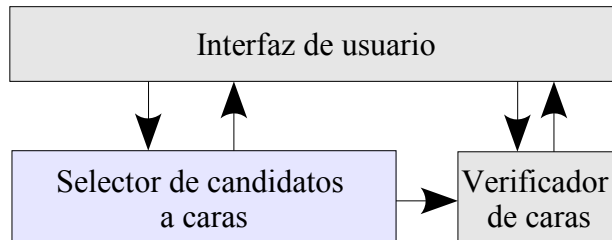


Figura 26. Arquitectura de la aplicación.

Las ventajas de la utilización de esta arquitectura son varias:

- En primer lugar, el selector de candidatos a caras se puede utilizar de forma independiente al interfaz de usuario y al verificador de caras. Esto permite utilizarlo como librería para otra aplicación en que se quiera hacer uso de él.
- En segundo lugar, el verificador de caras es totalmente independiente del resto del sistema por lo que se podría sustituir fácilmente por otro verificador mucho más potente sin tener que modificar nada del resto del código.

A continuación, vamos a ver detalles de cada uno de los tres bloques de los que se compone la aplicación centrándonos sobre todo en la implementación del selector de candidatos a caras.

Interfaz de usuario

Esta parte de la aplicación esta formada por un conjunto de clases encargadas de mostrar las distintas ventanas de la aplicación. En general son iguales que cualquier otra interfaz de usuario realizada en Java. Contamos con clases encargadas de:

- Interfaz de usuario de la ejecución paso a paso (`StepByStep`)
- Interfaz de usuario de la ejecución continua (`OneExecution`)
- Interfaz de usuario para mostrar los distintos candidatos encontrados (`CandidatesPanel`).

La parte que merece una atención más importante es la clase `CanvasPanel` que es la encargada de mostrar las imágenes en pantalla. Esta clase realiza las siguientes tareas:

- Dibujar una imagen en pantalla
- Marcar dentro de la imagen los ojos obtenidos en la fase de selección de candidatos a cara.
- Realizar la carga de imágenes de archivos.

Selector de candidatos a cara

Esta es la parte principal de la aplicación y el objetivo fundamental de la práctica. Se ha intentado que su implementación sea lo más parecida a la establecida en Java para el tratamiento de imágenes de forma que sus distintas partes puedan usarse por separado sin mucha complicación en caso de que sea necesario.

La forma en que hemos implementado el selector de candidatos a cara es la siguiente:

- Por un lado, tenemos la clase `TransformSequence` que se encarga del control del algoritmo e ir llamando a las distintas funciones cuando sea necesario.
- Por otra parte, tenemos otro grupo de clases encargadas de las operaciones básicas que utilizarán los distintos pasos del algoritmo como pueda ser el operador *average* o el cálculo de componentes conexas.
- En tercer lugar, tenemos el conjunto de clases encargadas de la realización de cada uno de los pasos del algoritmo. Cada paso está encapsulado en una clase distinta lo que facilita la modularidad de la aplicación.
- Por último, hay un par de clases encargadas de encapsular la información sobre los candidatos a ojos y caras encontrados durante el proceso con el fin de poder pasárselo al verificador de caras.

Control del algoritmo: `TransformSequence`

Como ya hemos comentado, esta clase se encarga de controlar el algoritmo, mantener la información común a los distintos pasos e ir invocando a la clase correspondiente en función de la situación en que nos encontremos en el algoritmo.

También permite consultar el paso en que se encuentra el algoritmo, algo que resulta muy útil durante la ejecución paso a paso.

Operador *average*: `AverageOp`

Esta clase se encarga de, dada una imagen P y unos valores w y h , devolver una matriz P' en la que se cumple que:

$$P'(x, y) = \text{avg}(P, x, y, h, w)$$

Este operador se utilizará en varios pasos del algoritmo. Concretamente en el segundo (*ocho vecinos*), cuarto (*eliminación de píxeles dispersos*) y quinto (*ampliación de regiones candidatas a ojos*).

La aplicación del operador *average* a toda la imagen puede llevar mucho tiempo. Sin embargo, es posible aplicarle una optimización de forma que se consiga hacer en tiempo lineal. Esta optimización está basada en las siguientes ecuaciones que se obtienen fácilmente de la definición del *average*:

$$avg(P, x, y, h, w) = \frac{\sum_{i=x}^{x+h-1} (\sum_{j=y}^{y+w-1} P(i, j) / w)}{h}$$

$$avg(P, x, y+1, 1, w) = avg(P, x, y, 1, w) + \frac{P(x, y+w) - P(x, y)}{w}$$

$$avg(P, x+1, y, h, 1) = avg(P, x, y, h, 1) + \frac{P(x+h, y) - P(x, y)}{h}$$

De esta forma, utilizando las dos últimas ecuaciones, podemos calcular el valor del siguiente elemento de la matriz en función del anterior. Por tanto, el algoritmo utilizado es:

1. Calcula el operador *average* de los píxeles de la primera fila utilizando la definición original del mismo.
2. Calcula el resto de filas haciendo uso de la ecuación mostrada más arriba.
3. Sobre los resultados obtenidos, vuelve a aplicar el operador *average* de la definición original pero ahora sobre los píxeles de la primera columna.
4. Calcula el resto de columnas con la ecuación indicada antes.

Con eso logramos el equivalente de aplicar el operador *average* a todos los píxeles de la imagen para un w y h dados pero con muchos menos cálculos.

Cálculo de componentes conexas: `ConnectedPixelsOp`

Esta clase se encarga de:

- Dada una imagen binaria, obtener todas las componentes conexas existentes en ella devolviendo una lista de todos los rectángulos externos de cada una de ellas.
- Dada una imagen binaria de la que ya han sido calculadas las componentes conexas y una lista de componentes conexas de dicha imagen, eliminar los píxeles correspondientes a ella.

Esta operación se utiliza en dos pasos del algoritmo: el tercero (*eliminación de componentes conexas grandes*) y el sexto (*eliminación de falsos candidatos a ojos*).

El algoritmo utilizado para el cálculo de componentes conexas es el indicado en los apuntes de la asignatura:

- 1 Recorremos la imagen de izquierda a derecha y de arriba a abajo.
 - 1.1 Para cada punto que tenga valor 1, miramos los vecinos de arriba y la izquierda de P. Si existen ya han sido visitados por el rastreo, de forma que si eran 1s, ya han sido etiquetados.
 - 1.1.1 Si todos los vecinos son ceros, damos a P una nueva etiqueta.
 - 1.1.2 Si todos los que son 1 tienen la misma etiqueta, se la damos a P.
 - 1.1.3 Si tienen distintas etiquetas, le damos a P una cualquiera de ellas y anotamos la equivalencia entre todas esas etiquetas (pertenecen a la misma componente).

- 2 A continuación, ordenamos las etiquetas en clases de equivalencia y a cada clase de equivalencia le asignamos una etiqueta nueva.
- 3 Finalmente, recorremos otra vez la imagen sustituyendo la etiqueta puesta en la primera iteración por la de la clase de equivalencia que le corresponde. Mientras vamos haciendo esto, vamos calculando los rectángulos externos de cada una de las componentes.
- 4 Devolvemos esos rectángulos externos de las componentes.

Para el borrado de píxeles pertenecientes a una componente simplemente hay que indicar cuál es la etiqueta que se dio a los píxeles de esa componente.

Conversión a escala de grises: `GrayScale`

Esta clase se encarga de convertir la imagen cargada de un archivo en una imagen en escala de grises para que pueda ser procesada por el algoritmo.

La forma en que realiza la conversión es haciendo uso de una funcionalidad que incluye el API de Java para tal efecto.

Ocho vecinos: `EightNeighOp`

Esta clase se ocupa de marcar los píxeles análogos a ojos utilizando el procedimiento descrito anteriormente (basado en el cálculo del operador *average* sobre las ocho regiones vecinas al píxel).

Para llevar a cabo su labor, esta clase sigue el siguiente procedimiento:

1. Hace uso de `AverageOp` para calcular el operador *average* de cada uno de los píxeles de la imagen para tres valores distintos de (h, w) .
2. Tras esto, crea una nueva imagen del tamaño de la original totalmente negra.
3. Marca en blanco todos aquellos píxeles en los que se cumplan seis o más de las restricciones establecidas en el algoritmo.

El resultado de aplicar esta operación es una imagen binaria en la que están marcados los píxeles análogos a ojos.

Eliminación de componentes conexas grandes: `RemoveLargeOp`

La misión de esta clase es eliminar aquellas componentes conexas demasiado grandes para ser un ojo, tal y como se indica en el algoritmo descrito anteriormente.

La forma en que realiza esto es:

1. Haciendo uso de `ConnectedPixelsOp`, obtiene todas las componentes conexas.
2. A continuación mira cuáles de ellas tienen rectángulos exteriores que no cumplen con las restricciones establecidas (mayores que dos veces el ancho o el alto del tamaño del ojo que estamos buscando) y las va guardando en una lista.
3. Obtiene una nueva imagen de la que se han eliminado todas aquellas componentes conexas que no cumplían las restricciones.

El resultado de este paso es una nueva imagen binaria a la que se le han eliminado aquellas componentes conexas demasiado grandes para poder ser un ojo.

Eliminación de píxeles dispersos: RemoveSparseOp

Esta clase se ocupa de eliminar aquellos píxeles análogos a ojos que están demasiado solos para poder pertenecer realmente a una región análoga a ojos.

Este proceso lo realiza haciendo uso del operador *average* de la siguiente forma:

- 1 Utiliza la clase `AverageOp` para calcular el operador *average* de cada uno de los píxeles de la imagen.
- 2 Recorre píxel a píxel imagen y, si el píxel es blanco (es un píxel análogo a ojos), comprueba si verifica la desigualdad indicada en el algoritmo haciendo uso de la matriz obtenida en el paso anterior.
 - 2.1 Si la verifica, se desmarca como píxel análogo a ojo (se pone en negro).
 - 2.2 Si no, se deja igual.

El resultado obtenido tras esta función es una imagen binaria a la que se la han eliminado aquellos píxeles análogos a ojo que estaban demasiado dispersos por la imagen.

Ampliación de regiones candidatas a ojos: MarkAnalogOp

Esta clase se encarga de añadir como píxeles análogos a ojos a aquellos que están muy rodeados por otros píxeles análogos a ojos.

Para lograrlo, volvemos a utilizar el operador *average* con un procedimiento muy similar al que hemos comentado en la clase anterior:

- 1 Utiliza la clase `AverageOp` para calcular el operador *average* de cada uno de los píxeles de la imagen.
- 2 Recorre píxel a píxel imagen y, si el píxel es negro (no es un píxel análogo a ojos), comprueba si se verifica la desigualdad indicada en el algoritmo haciendo uso de la matriz obtenida en el paso anterior.
 - 2.1 Si se verifica, se marca el píxel como análogo a ojo (se pone en blanco).
 - 2.2 Si no, se deja igual.

El resultado obtenido tras esta función es una imagen binaria a la que se la han añadido como píxeles análogos a ojo aquellos que estaban muy rodeados de otros también análogos a ojos.

Eliminación de falsos candidatos a ojos: RemoveNotEyesOp

Por medio de esta clase se eliminan aquellas regiones análogas a ojos que no cumplen las dimensiones adecuadas para ser ojos. Para hacerlo, hacemos uso de la clase encargada del cálculo de componentes conexas siguiendo el siguiente algoritmo:

1. Haciendo uso de `ConnectedPixelsOp` obtiene todas las componentes conexas.
2. A continuación, mira cuáles de ellas tienen rectángulos exteriores que no cumplen con las restricciones establecidas y las va guardando en una lista.
3. Obtiene una nueva imagen de la que se han eliminado todas aquellas componentes conexas que no cumplían las restricciones.

El resultado de la operación es una nueva imagen binaria a la que se le han eliminado aquellas componentes conexas que no cumplían las restricciones establecidas de tamaño para ser posibles ojos.

Detección de posibles pares de ojos: `PairOfEyesOp`

Esta clase tiene una doble función: por un lado, crea una nueva imagen basada en la que se le pasa pero a la que se le ha eliminado todas aquellas regiones análogas a ojos que no tenían otra región análoga a ojos cercana para poder formar un par de ojos. La otra función es devolver una lista de pares de ojos que se utilizará en el siguiente paso.

Para llevar a cabo estas dos funciones se sigue el siguiente procedimiento:

1. Cálculo de las componentes conexas de la imagen haciendo uso de `ConnectedPixelsOp`.
2. Recorre todas las componentes conexas buscando pares de ellas que verifiquen las restricciones especificadas en el algoritmo y las va almacenando en una lista.
3. Todas aquellas componentes conexas que no hayan sido almacenadas en la lista son las regiones que hay que eliminar. Para hacer esto, se utiliza la función implementada a tal efecto en `ConnectedPixelsOp`.

Preparación de la imagen para el verificador de caras: `FinalMaskOp`

El objetivo de esta clase es preparar los distintos candidatos para el verificador de caras. Para ello, recibe como entrada la lista de pares de ojos obtenida en el paso anterior y la imagen original y realiza lo siguiente:

- 1 Para cada par de ojos de la lista:
 - 1.1 Rota, reescala y recorta la imagen para obtener una imagen normalizada y pasársela al verificador de caras. Para hacer estas tres operaciones, utiliza un conjunto de funciones de la API de Java destinadas a tal efecto.
 - 1.2 Ecualiza el histograma de la imagen utilizando funciones de la API de *Java Advanced Imaging*.
 - 1.3 Aplica una máscara especificada en el artículo con el fin de eliminar aquellos píxeles del borde de la imagen que no forman parte de la cara.

El resultado de esta operación es una lista de imágenes que será lo que se vaya pasando al verificador de caras.

El verificador de caras

Junto al selector de candidatos a caras, hemos implementado un verificador muy sencillo pero que nos sirve para determinar la cara correcta en un número elevado de casos.

Este verificador se basa en la comparación de los candidatos a cara con una plantilla. Aquel candidato que más se aproxime a la plantilla será el seleccionado como cara. Por tanto, no es capaz de discernir si algo es o no una cara, simplemente indica qué es lo que se parece más a una cara. Esto es, si se le pasan tres imágenes que no son caras, no será capaz de decir “ninguna es una cara”, simplemente seleccionará alguna y dirá “esta es la que más se parece a una cara”.

La forma en que hemos implementado el verificador de caras permite que pueda ser sustituido por otro más potente sin necesidad de modificar nada del resto del código.

Para analizar cuál es la imagen que más se parece a una cara se basa en el valor de correlación entre el candidato a cara I_T y la plantilla T :

$$C_N(I_T) = \frac{\langle I_T T \rangle - \langle I_T \rangle \langle T \rangle}{\sigma(I_T) \sigma(T)}$$

donde $\langle \rangle$ es la media, $I_T T$ representa el producto píxel a píxel y σ es la desviación estándar.

La forma en que funciona el verificador es sencillo: se le pasa una lista con todos los candidatos a caras. Les calcula el valor de correlación a cada uno de ellos y aquél con mayor valor de correlación es el seleccionado.

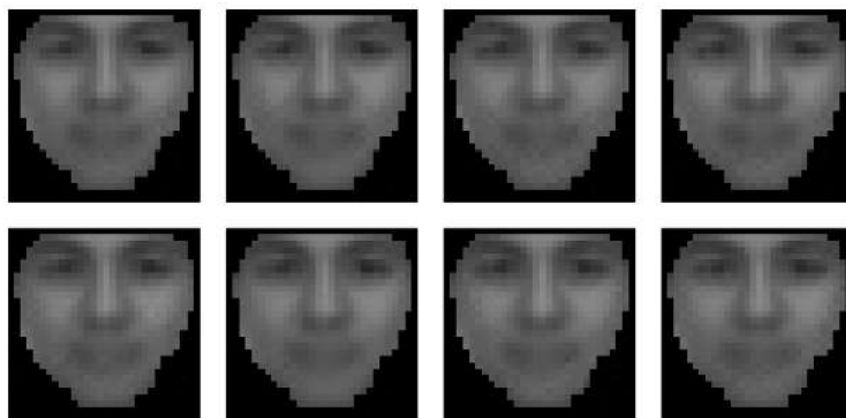


Figura 27. Plantillas de caras para el verificador.

Mejoras desarrolladas

Como ya se ha comentado a lo largo de todo este apartado las mejoras fundamentales desarrolladas en la aplicación son:

- Utilización de la mejora comentada anteriormente para incrementar la eficiencia en el cálculo del operador *average*.
- Implementación de un verificador de caras simple.

Resultados y conclusiones

Para evaluar la eficacia del algoritmo, utilizamos la base de datos de caras BioID (<http://www.bioid.com>). Esta base de datos contiene un conjunto de caras utilizadas para la prueba de programas de reconocimiento facial. Las imágenes son frontales y de cabeza y hombros. Las imágenes presentan una gran variedad de iluminaciones y tamaños de cara distintos. Además los fondos son muy complejos. Concretamente, la base de datos consiste en 1521 imágenes frontales en escala de grises de 23 personas distintas con una resolución de 384 x 286 píxeles.

Al aplicar el algoritmo a algunas de estas imágenes observamos que en un gran número de ellas se consigue detectar la posición de los ojos con un error muy bajo. En concreto, tenía éxito en un número de imágenes por encima del 70%. Además, de las que fallaba, un alto número de ellas habrían podido detectarse con éxito si el verificador de caras hubiese sido algo más potente.

Por tanto, los resultados obtenidos son bastante satisfactorios en ese aspecto. Además, el algoritmo es muy eficiente ya que sólo requiere varias pasadas para lograr el reconocimiento de los ojos.

El algoritmo también es bastante robusto ya que sólo depende de que:

- los ojos son zonas más oscuras que sus vecinos
- las regiones análogas a ojos deben tener el tamaño de ojos a una determinada escala.



Figura 28. Algunas detecciones correctas realizadas.

Limitaciones

Pese a todo, el algoritmo presenta una serie de limitaciones a la hora de obtener los candidatos a ojos. En primer lugar, requiere que las imágenes sean frontales, no es capaz de trabajar con caras de perfil, por ejemplo.

Centrándonos ya en imágenes frontales, los principales problemas que le pueden afectar son: brillos en la zona de los ojos debido a una imagen muy clara o bien a una persona que lleve gafas. También le afectan las rotaciones de la cara demasiado pronunciadas. Finalmente, hay que destacar que la mayoría de los problemas los presenta al intentar diferenciar los ojos de las cejas debido a que las cejas tienen unas características muy similares a los ojos y también son regiones más oscuras que las de su alrededor.

Ejemplos de lo anterior se pueden comprobar en la Figura 29 en la que podemos ver como, en la primera de las imágenes no se encuentran los ojos debido al brillo de las gafas. En la segunda, confunde los ojos de la mujer con sus cejas (en muchos de estos casos, un verificador de caras más potentes conseguiría mejorar bastante los resultados). Por último, en la tercera imagen se equivoca completamente en la posición de los ojos (de nuevo un mejor verificador de caras podría solucionarlo) debido a que tiene los ojos cerrados y a la gran luminosidad que le da en la cara.



Figura 29. Algunas detecciones incorrectas.

Planificación del trabajo

La planificación que hemos llevado a cabo para la realización del trabajo ha sido la siguiente:

En primer lugar, los dos miembros del grupo nos reunimos para analizar el artículo que describe el algoritmo que hemos implementado y decidir qué lenguaje utilizaríamos para ello. A continuación, nos planteamos la arquitectura que seguiría nuestra aplicación. A partir de ese momento dividimos el trabajo por la mitad y cada uno realizó una parte. Algunas veces juntos (la mayoría) y otras por separado.

A medida que íbamos desarrollando la aplicación tuvimos varias reuniones con el fin de unir las distintas partes que habíamos ido desarrollando cada uno.

El tiempo total dedicado por cada miembro del grupo ha sido de, aproximadamente 54 horas cada uno. De las cuales, 32 han sido de tiempo conjunto y 22 de tiempo por separado.

A continuación incluimos un cuaderno de tiempos de las distintas sesiones:

Fecha	Inicio	Fin	Tiempo Interrupción	Δ tiempo	Miembros del grupo	Actividad	Com.
10/Dic/03	18.30	20.30	15 min.	1h45min	Todos	Conocimiento del algoritmo y elección de herramientas	
15/Dic/03	15.30	17.30	20 min.	1h40min	Todos	Creación del entorno	
18/Dic/03	19.45	21.00	15 min.	1h	Todos	Resolución de dudas en clase	
22/Dic/03	9.10	13.40	30 min.	4h	Todos	Programa principal y búsqueda de librerías auxiliares.	
23/Dic/03	9.15	14.15	30 min.	4h30min	Todos	Aprendizaje y Adaptación de Visual Editor	
26/Dic/03	9.15	14.30	45min	4h30min	Todos	Implementación operador average e introducción del mismo al algoritmo.	
27/Dic/03	16.00	21.00	1h	4h	Manuel	Continuación del algoritmo.	
29/Dic/03	15.30	20.00	1h	4h	Beatriz	Continuación del algoritmo.	
30/Dic/03	9.30	13.30	30 min.	3h30min	Todos	Correcciones y toma de decisiones de diseño de fases posteriores	
2/Ene/04	9.30	14.00	45min.	3h45min	Todos	Continuación del algoritmo y repartición de trabajo.	
3/Ene/04	16.30	20.45	30min.	3h45min	Beatriz	Finalización etapas reconocedor de ojos.	
4/Ene/04	15.00	19.30	45min.	3h45min	Manuel	Comienzo del reconocedor de caras	
8/Ene/04	9.10	11.50	15min.	2h25min	Todos	Integración, corrección de errores y continuación.	
9/Ene/04	11.00	14.00	15min.	2h45min	Todos	Finalización del algoritmo y repartición de trabajo.	

Selección eficiente de candidatos a cara mediante detección de ojos

Fecha	Inicio	Fin	Tiempo Interrupción	Δ tiempo	Miembros del grupo	Actividad	Com.
10/Ene/04	12.00	15.30	15min.	3h15min	Manuel	Mejora de la aplicación.	
10/Ene/04	16.30	20.00	45min.	3h15min	Beatriz	Interfaz de usuario de la aplicación.	
11/Ene/04	10.00	13.30	30min.	3h	Beatriz	Interfaz, comienzo de presentación	
11/Ene/04	17.00	21.00	1h	3h	Manuel	Finalización interfaz y presentación.	
12/Ene/04	15.30	17.30	15min.	1h45min	Todos	Pruebas y resolución de errores	
13/Ene/04	16.00	19.00	1h	2h	Manuel	Preparación de la presentación	
13/Ene/04	18.00	20.30	30min.	2h	Beatriz	Pruebas y elección de ejemplos	
14/Ene/04	9.10	10.30	0	1h20min	Todos	Resolución de problemas técnicos	
6/Feb/04	18.00	21.15	15min.	3h	Manuel	Documentación	
7/Feb/04	8.30	14.45	15min.	6h	Beatriz	Guía de usuario y tabla de hojas de tiempos.	
7/Feb/04	12.00	15.30	30min.	3h	Manuel	Documentación	

Bibliografía

Bibliografía técnica:

- Jianxin Wu, Zhi-Hua Zhou. *Efficient face candidates selector for face detection*. Pattern Recognition 36 (2003) 1175-1186.
- R. Brunelli, T. Poggio, *Face recognition: features versus templates*, IEEE Trans. Pattern Anal. Mach. Intell. 15 (10) (1993) 1042-1052.
- Rocío González, Pedro Real. *Apuntes procesamiento de imágenes digitales*. (<http://www.us.es/gtocom/pid/>).

Para la implementación

- Documentación API de Java (<http://java.sun.com>)
- Documentación *Java Advanced Imaging* API (<http://java.sun.com/products/java-media/jai/>).