Trabajo de Fin de Grado

Herramienta para la corrección automática de autómatas finitos

Tool for the automatic correction of finite automata

Iván García Campos

San Cristóbal de La Laguna, 2 de julio de 2017

D. **Gara Miranda Valladares**, con N.I.F. 78.563.584-T profesora Ayudante de Doctor adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

C E R T I F I C A (N)

Que la presente memoria titulada:

"*Herramienta para la corrección automática de autómatas finitos*"

ha sido realizada bajo su dirección por D. **Iván García Campos**, con N.I.F. 43833191K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de julio de 2017

Agradecimientos

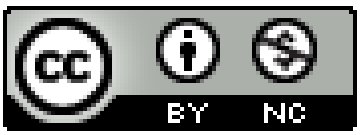
Querría dar las gracias a todas aquellas personas que han contribuido a la realización de este Trabajo Fin de Grado:

En primer lugar, quisiera agradecer a Gara Miranda Valladares, tutora de este proyecto, por la confianza que ha depositado en mi. Su apoyo, paciencia y compromiso ha sido parte esencial en el desarrollo de este trabajo.

Quisiera también, reconocer a todas aquellas personas que me han apoyado y dado ánimos para seguir hacia delante. Agradecer a todos aquellos que han conseguido que en mi etapa universitaria haya crecido como persona. Ha sido un largo y duro camino pero ha merecido la pena.

Por último me gustaría dar las gracias a mis padres y hermana por estar siempre ahí y confiar siempre en mi, por saber comprenderme, relativizar cualquier problema y enseñarme lo que realmente es importante.

Licencia



Resumen

El objetivo de este trabajo ha sido diseñar una herramienta que, partiendo de la imagen o captura de un diagrama de estados especificado en papel que representa, detecte los estados y transiciones del mismo y, automáticamente, genere la especificación digital del autómata para, a continuación, compararlo con otro autómata y comprobar si son equivalentes. Se considerarán la detección de DFA (Deterministic Finite Automaton) y NFA (Nondeterministic Finite Automaton). Para ello, el presente trabajo describe brevemente el tipo de herramientas que existen para la simulación de autómatas y/o para la docencia de autómatas y lenguajes formales. Se describe también el tipo de herramientas disponibles en la actualidad para el procesamiento de imágenes a partir de una imagen y detección de formas básicas. Seguidamente, se profundiza en el diseño y desarrollo de una aplicación que cumpla con el objetivo deseado.
DCAFI (Detector y Corrector de Autómatas Finitos en Imágenes) es una aplicación de escritorio capaz de comprobar la equivalencia entre autómatas y, a partir de una imagen, detectar y extraer la codificación de este tipo de modelo computacional. La herramienta ha sido diseñada en C++ y puede dividirse en tres fases: la fase de carga, la fase de detección y la fase de corrección.
Para el tratamiento de las imágenes se utilizaran librerías OpenCV sobre el lenguaje de programación C++.

Palabras clave: Autómatas finitos, detección de fórmas básicas, herramienta de corrección, DCAFI, OpenCV.

Abstract

The purpose of this work has been to design a tool that, based on an image or capture of a state diagram specified in paper, detects its states and transitions and automatically generates the digital specification for, after that,comparing to another automaton and verifying if they are equal. Detecting DFA (Deterministic Finite Automaton), NFA (Nondeterministic Finite automaton) will be considered. In order to do it, this paper briefly describes the available types of tools for simulating automata and/or for teaching formal languages and automata. It is also described the type of tools currently available for image processing based on an image and detection of basic shapes. After, it focuses into the design and development of an application that meets the expected goal. DCAFI (Detector y Corrector de Autómatas Finitos en Imagenes) is a desktop application capable of checking the equivalence between automatas, and from an image, detecting and extracting the coding of this type of computational model. The tool has been designed in C++ and can be divided into three phases: the loading phase, the detection phase and the correction phase. OpenCV on C++ programming language libraries will be used for image processing.

Keywords: *DFA, NFA, Turing Machine, OpenCV.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El presente Trabajo Fin de Grado (TFG) se enmarca en el ámbito de la asignatura "Computabilidad y Algoritmia". Esta asignatura se imparte en la titulación Grado en Ingeniería Informática y pertenece al bloque de formación básica. El módulo principal de esta asignatura se dedica al estudio de la teoría de autómatas y de los lenguajes formales. La teoría de autómatas y lenguajes formales se puede ubicar en el campo científico de la Informática Teórica, un campo clásico y multidisciplinar dentro de los estudios universitarios de Informática. Es un campo clásico debido no sólo a su antigüedad (anterior a la construcción de los primeros ordenadores) sino, sobre todo, a que sus contenidos principales no dependen de los rápidos avances tecnológicos que han hecho que otras ramas de la Informática deban adaptarse a los nuevos tiempos a un ritmo vertiginoso. Es multidisciplinar porque en sus cimientos encontramos campos tan variados como la lingüística, las matemáticas o la electrónica.

El hecho de que esta materia no haya sufrido grandes cambios en las últimas décadas no le resta interés dentro de un plan de estudios de una Ingeniería Informática puesto que el estudio de las máquinas secuenciales que abarca la teoría de autómatas, por una parte, sienta las bases de la algoritmia y permite modelar y diseñar soluciones para un gran número de problemas. Por otra parte, permite abordar cuestiones de gran interés en el campo de la informática como qué tipo de problemas pueden ser resueltos por un computador o incluso, en caso de existir una solución computable para un problema, cómo podemos medir la calidad (en términos de eficacia) de dicha solución. Es decir, el estudio de la teoría de autómatas que se lleva a cabo en el marco de la asignatura "Computabilidad y Algoritmia" es la puerta que nos permite la entrada hacia campos tan interesantes como la computabilidad y la complejidad algorítmica. Además, una de las principales aportaciones del estudio de los lenguajes formales, sobre todo desde un punto de vista práctico, es su contribución al diseño de lenguajes de programación y a la construcción de sus correspondientes traductores, cuestión de especial importancia para estudiantes de Ingeniería Informática.

1.1. Objetivos

Durante la impartición de la asignatura Computabilidad y Algoritmia se realizan diferentes actividades cuya finalidad principal es la de diseñar, trabajar y visualizar el comportamiento de expresiones regulares, autómatas finitos deterministas, autómatas finitos no deterministas, gramáticas, y máquinas de Turing. Además de programar dichos comportamientos, al alumno se le facilitan herramientas como JFLAP [44] *(Java Formal Language and Automata Package)* para la simulación de autómatas y lenguajes formales.

Sin embargo, en algunas tareas evaluativas en el marco de la asignatura (como, por ejemplo, el examen final) se solicita al alumnado que diseñe, en papel, un autómata o máquina de Turing que reconozca un determinado lenguaje. Para la evaluación de estas tareas en un entorno como el JFLAP sería necesario copiar, manualmente el diseño del autómata a la interfaz gráfica de la herramienta para luego comprobar su buen funcionamiento con un conjunto determinado de cadenas. Si la evaluación de estos autómatas se realizara a partir de los programas que diseñan los alumnos a lo largo de la asignatura, sería necesario definir manualmente el fichero de entrada, sus especificaciones y el conjunto de cadenas de prueba. Además, actualmente los alumnos no tienen posibilidad de comparar directamente un autómata que hayan diseñado a papel dado otro escrito en la pizarra o realizado por otro compañero.

En este sentido, **el objetivo principal de este TFG es diseñar una herramienta que, partiendo de la imagen o captura de un diagrama de estados especificado en papel, detecte los estados y transiciones del mismo y, automáticamente, genere la especificación digital para, a continuación, comprobar si su comportamiento es equivalente al de un autómata proporcionado como entrada verificando así su correcto funcionamiento.** Para poder realizar dicha herramienta, se analizará brevemente el tipo de herramientas que existen para la simulación de autómatas y/o para la docencia de autómatas y lenguajes formales. Además, se analizará también el tipo de herramientas disponibles actualmente para el procesamiento de imágenes a partir de una imagen y detección de formas básicas. Hay que tener en cuenta que la detección será más complicada ya que el autómata estará dibujado a mano.

1.2. Planificación

Este trabajo se ha dividido en 5 actividades:

- **ACT1: Reconocimiento de formas simples.** Partiendo de una imagen o fotografía de un examen, apuntes o pizarra, identificar el diagrama de transiciones y detectar en él los estados (círculos) y las transiciones (líneas o curvas), así como la diferenciación de los estados inicial/finales y los

símbolos que etiquetan las transiciones.

- **ACT2: Definición del autómata a partir del conjunto de formas reconocidas.**
  Partiendo de las formas básicas reconocidas en la imagen, generar una especificación o modelo digital de un autómata.

- **ACT3: Simulación ante cadena de entrada.**
  Partiendo de la especificación del autómata simular su comportamiento ante una cadena de entrada especificada por el usuario, indicando si la misma es aceptada o no.

- **ACT4: Validación final.**
  Comprobar la equivalencia entre autómatas pudiendo introducir estos a partir de una imagen o fichero codificado.

- **ACT5: Documentación**
  Elaboración del material de soporte para el uso de la herramienta así como la memoria del TFG, conclusiones y trabajo futuro.

1.3. Conceptos previos.

Durante el desarrollo de la aplicación se han usado diferentes algoritmos para el análisis de autómatas finitos y lenguajes formales. Para un correcto estudio de la aplicación, es necesario conocer algunos conceptos teóricos. [40] [41] [30]

Lenguaje formal.
En matemáticas, lógica y ciencias de la computación, un lenguaje formal está formado por un conjunto de palabras de longitud finita en los casos más simples o expresiones validas formadas a partir de un alfabeto (conjunto de caracteres) finito. Un posible alfabeto podría ser el que contiene los símbolos *'a'* y *'b'*, teniendo como posible cadena válida *'abbaaab'*. Un lenguaje sobre este alfabeto, que incluya esta cadena podría ser el conjunto de todas las cadenas que contienen 'a' pares y terminan con al menos una 'b'.

Autómata finito.
Un autómata es un modelo matemático para una máquina de estado finita (*Finite State Machines, FSM*) [9]. Una FSM es una máquina que, dada una entrada de símbolos, circula por una serie de estados de acuerdo a una función de transición. En este documento se ha trabajado con dos tipos de autómatas finitos:

- **Autómata finito determinista**
  Cada estado de un autómata de este tipo tiene una transición por cada

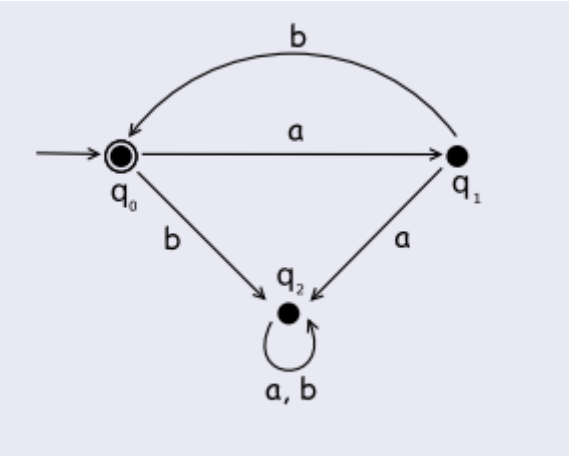


Figura 1.1: Ejemplo DFA

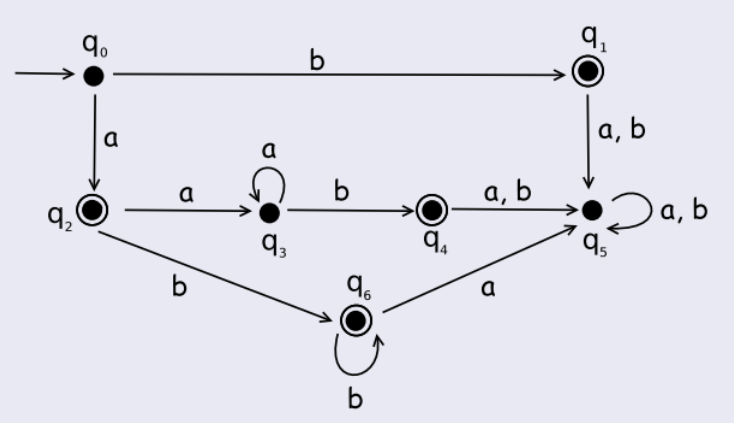


Figura 1.2: Ejemplo NFA

símbolo del alfabeto desde cada estado. Así por ejemplo, para la Figura1.1, si se parte del alfabeto $\sigma = \{a, b\}$, el autómata que se presenta a continuación reconocerá las cadenas de 'a' seguidas de 'b' de la forma `(ab)*`.

- **Autómata finito no determinista**
  Además de ser capaz de alcanzar más de un estado leyendo un símbolo es capaz de alcanzar algunos sin leer ningún símbolo. Este tipo de transiciones se representan etiquetándolas con ε. El autómata que se presenta en la Figura 1.2 parte del alfabeto de entrada $\sigma = \{a, b\}$ y reconoce las cadenas que contiene una única *'a'* al inicio de la cadena seguida de un número indeterminado de *b's* o un número indeterminado de *'aes'* seguido de una única *'b'* al final de la cadena.

Máquina de Turing

En ciencias de la computación, una maquina de Turing es un dispositivo que

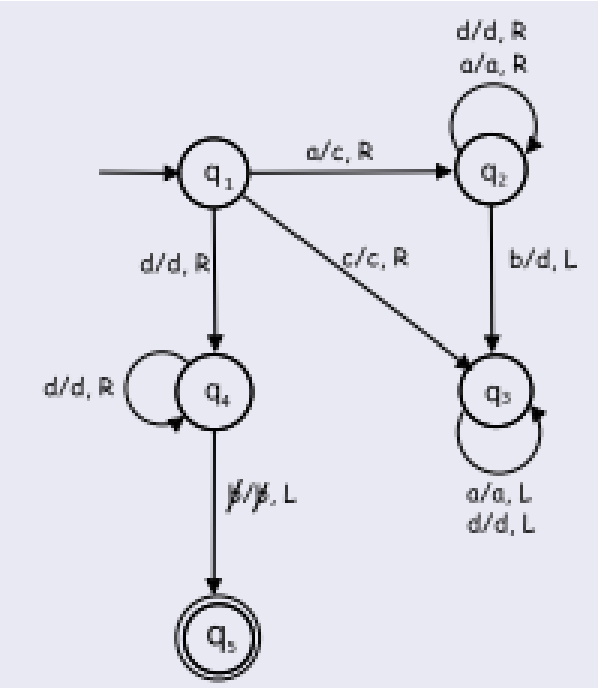


Figura 1.3: Ejemplo MT

manipula símbolos sobre una tira de cinta infinita de acuerdo a una tabla de reglas. Una máquina de Turing puede ser adaptada para simular la lógica de cualquier algoritmo de computador. Un ejemplo de máquina de Turing podría ser la de diseñar un sistema que acepte cadenas de *a's* seguidas de *b's* teniendo el mismo número de ambos símbolos (véase la Figura 1.3.

Equivalencia entre autómatas

Dados dos autómatas finitos, M_1 y M_2, se dice que M_1 y M_2 son equivalentes si reconocen el mismo lenguaje:

$$L(M_1) = L(M_2)$$

Para determinar si dos autómatas finitos reconocen el mismo lenguaje habría que asegurar que todas las cadenas que son aceptadas por $L(M_1)$ también son aceptadas por M_2. El hecho de encontrar una única cadena para la cual este comportamiento no se cumple, sería una prueba de que $L(M_1)$ y $L(M_2)$ son distintos.

Una alternativa que si es viable, consiste en obtener para cada uno de los autómatas finitos definidos, su correspondiente DFA mínimo y comparar la equivalencia entre ambos. Para convertir un NFA en un DFA se aplicará el *algoritmo de construcción de subconjuntos* [2] y para minimizar el DFA se aplicará el *algoritmo de minimización de estados* [3]

Capítulo 2

Herramientas para la simulación de autómatas

Actualmente, existen algunas herramientas didácticas que se han desarrollado para dar solución a la necesidad de simular autómatas en un marco educativo como SELFA [35], TALFi [28], JFLAP [44], dk.brics [29] [7] o SEPa! Project [47] La mayoría de este tipo de herramientas ofrece al estudiante la posibilidad de afrontar y superar diferentes problemas prácticos [34] mejorando el aprendizaje y la enseñanza por parte del personal docente. Otros estudios intentan analizar la perspectiva con el que se enfrenta el alumno a este tipo de problemas y buscar una relación entre la velocidad y el acierto que se produce en el alumnado [39] para evaluar tanto a los alumnos como la dificultad de los problemas planteados.

Lo que queda patente en la mayoría de estudios es que los alumnos encuentran dificultades cuando se enfrentan a los autómatas finitos debido al nivel de complejidad y abstracción de los mismos, aunado a la escasa participación en clase, los malos hábitos de estudio o la falta de dedicación. Además, para adquirir los conocimientos en esta materia es necesaria la realización de un gran número de ejercicios. Por ello, otros estudios basan sus investigaciones en el diseño y desarrollo de aplicaciones que ayuden al aprendizaje a través de ejercicios prácticos, explicaciones y evaluación del usuario [37]. La necesidad de diseñar y proporcionar ejercicios con sus respectivas soluciones permite a los estudiantes resolverlos por su cuenta, comprobar la solución obtenida y ver paso a paso la estrategia a seguir.

A continuación se analiza el estado del arte de las principales herramientas utilizadas para la simulación, análisis y validación de autómatas finitos y lenguajes regulares existentes en la actualidad.

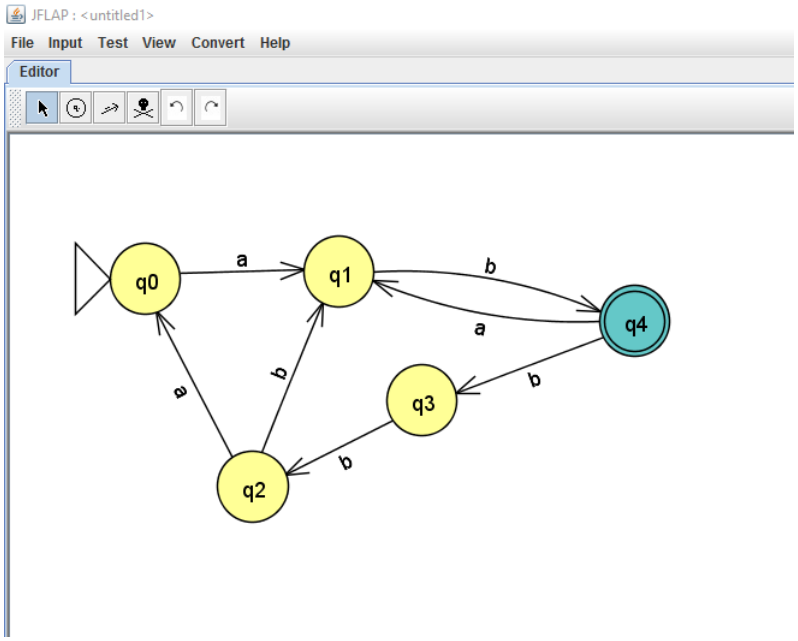


Figura 2.1: Ejemplo JFlap

2.1. Herramienta gráfica JFLAP

La herramienta gráfica JFLAP es una aplicación Java muy usada en el ámbito docente para el estudio y simulación de autómatas y lenguajes formales. Es capaz de realizar las operaciones más comunes sobre los lenguajes regulares (autómatas finitos, expresiones regulares y gramáticas regulares), lenguajes independientes del contexto (autómatas con pila y gramáticas independientes del contexto) y lenguajes recursivamente enumerables (máquinas de Turing). En la Figura 2.1 puede verse la representación de un DFA con JFLAP.

La herramienta presenta una interfaz gráfica en la que se puede diseñar el diagrama de estados de un autómata finito para, a continuación, indicar una relación de cadenas de entrada con las cuales verificar el correcto funcionamiento del mismo. Este diseño se realiza de forma manual desde la propia interfaz de la herramienta y, en el caso concreto de un autómata finito, supone la especificación y/o creación de los elementos siguientes:

- Estados del autómata
- Estado inicial o de arranque
- Estados finales
- Transiciones que parten de un determinado estado origen y alcanzan un determinado estado destino. Estas transiciones son etiquetadas con alguno de los símbolos del alfabeto o con la cadena vacía.

Figura 2.2: Ejemplo SELFA

Este tipo de herramientas de simulación son primordiales para la realización de prácticas en el ámbito de la **Teoría de la Computación**, así como para la fase de estudio de los conceptos fundamentales, pues permite al alumnado verificar si los elementos diseñados tienen el comportamiento deseado.

2.2. Herramienta SELFA

SELFA(SoftwarE for Learning Formal languages and Automata theory) es otra herramienta presentada en Febrero de 2009 [35] para la simulación y aprendizaje de lenguajes formales y autómatas finitos fue diseñada para mejorar la calidad de enseñanza en los cursos de Teoría de Computadores.
La herramienta permite la propuesta y resolución de problemas en todos los contenidos de la materia a excepción de las máquinas de Turing. Su diseño está basado en un procesador de lenguajes y la interacción con la misma se realiza a través de un lenguaje formal. En la Figura 2.2 puede observarse como se codifica un autómata con la herramienta SELFA.

SELFA facilita el aprendizaje de los principales conceptos de la Teoría de la Computación siendo una interfaz intuitiva pero compleja de utilizar ya que requiere que el usuario posea conocimiento avanzado además de no permitir al estudiante practicar el proceso de creación de DFA ni comprobar si cierta cadena es aceptada o no por el autómata correspondiente.

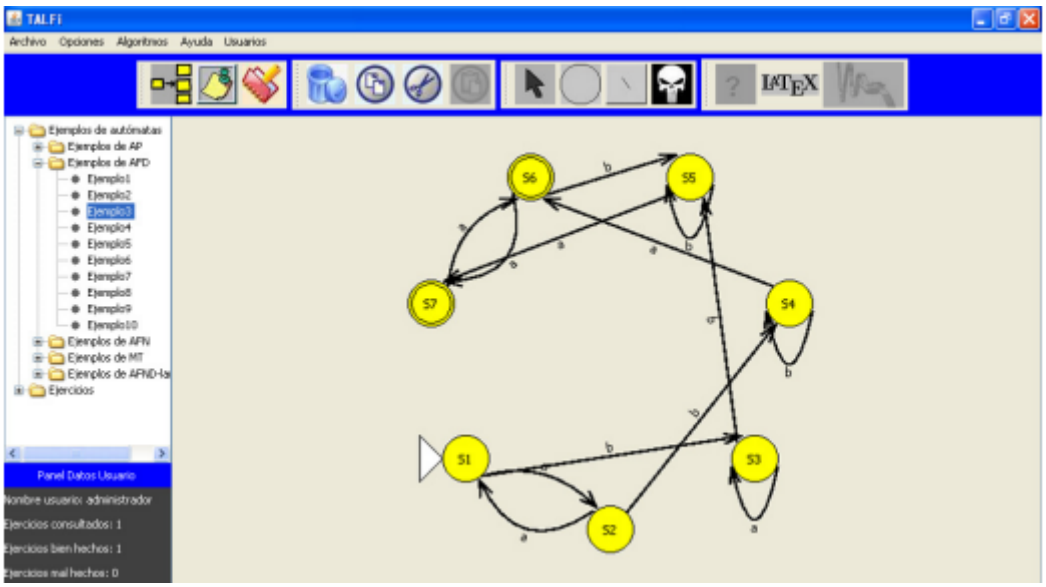


Figura 2.3: Ejemplo TALFi

2.3. Herramienta TALFi

TALFi [28] es una herramienta para el aprendizaje y el uso de diversos algoritmos aplicados al tratamiento de autómatas. TALFi dispone de una base de datos con diversos ejemplos pudiendo ampliarse por el usuario. La aplicación está pensada para ayudar a los estudiantes que quieran aprender lenguajes formales y autómatas. En la Figura 2.3 se observa un ejemplo de autómata proporcionado por la herramienta TALFi.

El proyecto fue desarrollado por dos grupos de trabajo distintos del departamento de Sistemas Informáticos de la Universidad Complutense de Madrid. Posteriormente fue modificado como proyecto de innovación educativa por la misma universidad.

2.4. Herramientas Dk.Brics

2.4.1. Dk.Brics.Automaton

Dk.Brics.Automaton [29] es una biblioteca Java para el tratamiento de autómatas con estados finitos que soportan operaciones con expresiones y gramáticas regulares como la concatenacion, union, cierre de Kleene e intersección. En la Figura 2.4 puede verse parte del código de esta biblioteca.

```
import dk.brics.automaton.RegExp;
import dk.brics.automaton.RunAutomaton;

class RegexRefPredicate extends OperatorPredicate<ChangeData> {
  private final Provider<ReviewDb> dbProvider;
  private final RunAutomaton pattern;

  RegexRefPredicate(Provider<ReviewDb> dbProvider, String re) {
    super(ChangeQueryBuilder.FIELD_REF, re);

    if (re.startsWith("^")) {
      re = re.substring(1);
    }

    if (re.endsWith("$") && !re.endsWith("\\$")) {
      re = re.substring(0, re.length() - 1);
    }

    this.dbProvider = dbProvider;
    this.pattern = new RunAutomaton(new RegExp(re).toAutomaton());
  }

  @Override
  public boolean match(final ChangeData object) throws OrmException {
    Change change = object.change(dbProvider);
    if (change == null) {
      return false;
    }
    return pattern.run(change.getDest().get());
  }

  @Override
  public int getCost() {
    return 1;
  }
}
```

Figura 2.4: Ejemplo Uso de dk.brics.automaton

2.4.2. Dk.Brics.Grammar

Dk.Brics.Grammar [7] es un analizador para la correcta escritura y búsqueda de ambigüedades en gramáticas libres de contexto. La aplicación escrita en lenguaje Java permite introducir gramáticas desde fichero o URL. Ambas herramientas han sido desarrolladas por Anders Møller en la Universidad Aarhus de Dinamarca y forma parte de un conjunto de herramientas para el análisis de lenguajes. En la Figura 2.5 puede verse un ejemplo de búsqueda de ambigüedades con Dk.Brics.Grammar.

2.5. Proyecto SEPa!

El **Proyecto SEPa!** [47] busca la creación de una serie de herramientas para la simulación de autómatas y traductores de lenguajes formales con el objetivo de facilitar al estudiante el aprendizaje autónomo de conceptos y procedimientos. La herramienta *Chalchalero* (véase la Figura 2.6) incluida en dicho proyecto permite simular autómatas finitos, gramáticas regulares, analizadores lexicográficos y autómatas de pila.

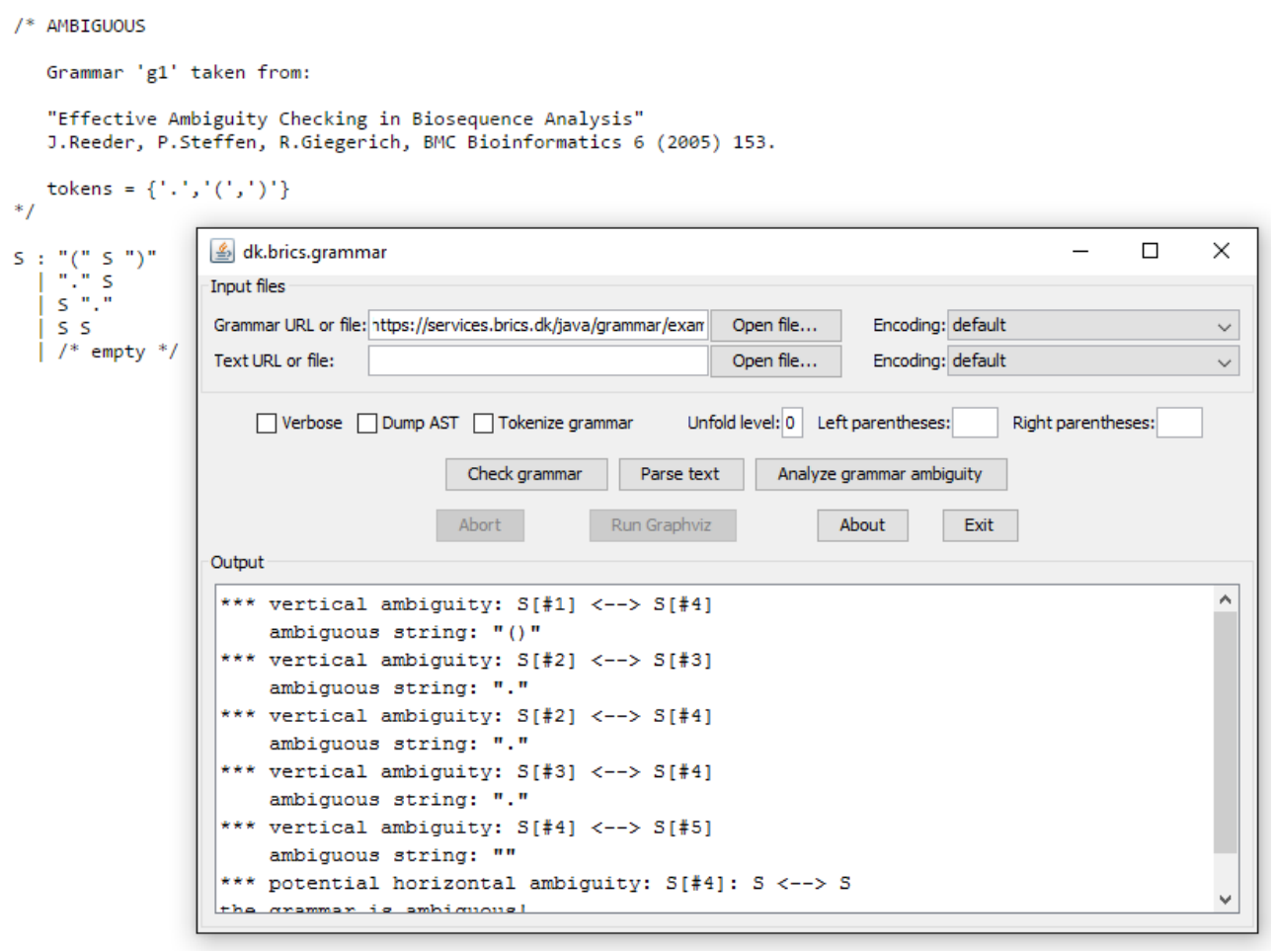


Figura 2.5: Ejemplo dk.brics.grammar

Chalchalero utiliza la herramienta *Graphviz* [15] para la representación de grafos y diagramas de estado. Entre las funcionalidades más importantes de *Chalchalero* se encuentra la explicación paso a paso de algoritmos sobre autómatas finitos y lenguajes formales, conversiones, comparativa entre lenguajes regulares, lema de bombeo y detección de ambigüedades. Además, es una herramienta bastante intuitiva que cuenta con diferentes ejemplos de prueba para cada una de las opciones.

2.6. Comparativa de las herramientas

Las herramientas actualmente utilizadas en el marco de la *Teoría de la Computación* para el aprendizaje autónomo y guiado de autómatas y lenguajes formales han mejorado enormemente en las últimas dos décadas, logrando interfaces mucho más intuitivas, dinámicas y extensas en sus funcionalidades [1]. Sin embargo, no hemos encontrado ninguna herramienta que automatice la evaluación de autómatas finitos a partir de una imagen real. Actualmente la

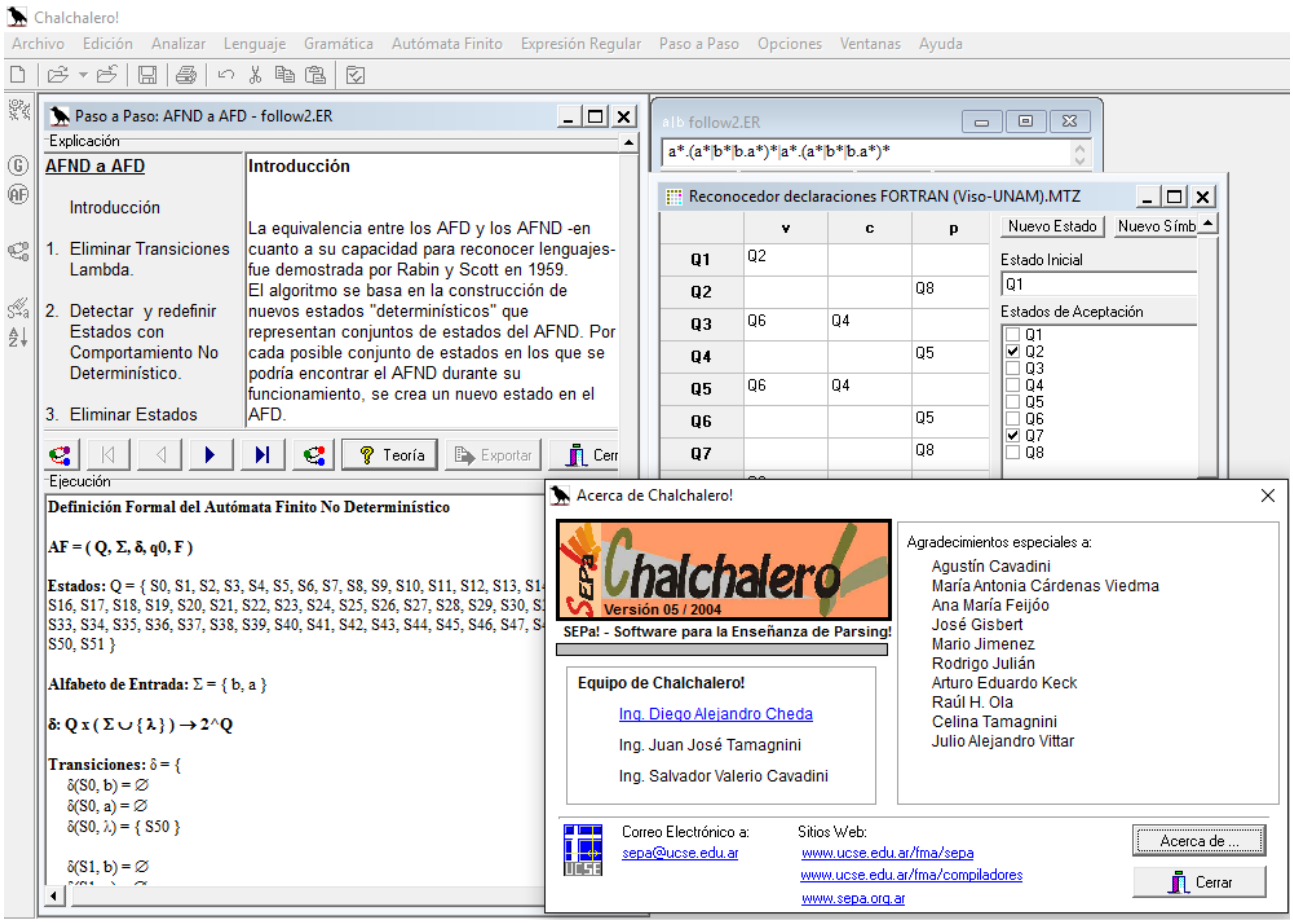


Figura 2.6: Ejemplo Proyecto SEPa! Chalchalero

mayor parte de las aplicaciones existentes sirven para la simulación y aprendizaje autónomo. Para la evaluación de un autómata partiendo de una imagen, si se desea utilizar entornos como el JFLAP o SELFA sería necesario copiar, manualmente, el diseño del autómata a la interfaz gráfica de la herramienta.

En la actualidad, en la asignatura *Computabilidad y Algoritmia*, el sistema de corrección de este tipo de ejercicios es manual. Si el profesorado pide un ejercicio al alumnado, debe corregir uno por uno, cada autómata comprobando que se cumplen las especificaciones. Por otro lado, si el alumno diseña un autómata y quiere probar su funcionamiento, debe introducirlo manualmente en alguna herramienta como JFLAP y analizarlo con un conjunto de cadenas.

Capítulo 3

Procesado de imágenes y detección de formas básicas

El procesamiento digital de imágenes es el conjunto de técnicas que se aplican a las imágenes digitales con el objetivo de mejorar la calidad o facilitar la búsqueda de información para su manejo e interpretación. En este caso se pretende la identificación de los elementos en diagramas de transición como el de la Figura 3.1.

Para ello, deberemos identificar formas básicas como son:

- Círculos para la identificación de los estados como son q_0, q_1 y q_2 en la Figura 3.1. Se deberá identificar cuáles de ellos son nodos finales.
- Líneas para la identificación de las transiciones teniendo en cuenta que dichas líneas deben tener un estado origen y uno final. Además, debe detectarse el sentido de las transiciones. Ej: de q_0 a q_1 con 1 en la Figura 3.1.
- Letras para identificar los símbolos del alfabeto.
- Línea que establece el estado de arranque del autómata. Apunta a un estado de la máquina y es único.
- Doble círculo que indica un estado de aceptación.

Se deberá tener en cuenta filtrar la imagen para reducir el ruido, ajustar el contraste y reducir la cantidad de elementos encontrados hasta que se ajusten a la realidad. Por otro lado, se deberá reconocer estructuras geométricas, tener en cuenta la convexidad, conectividad, tamaño y distancia entre los distintos elementos.

Por ello, antes de realizar cualquier tipo de implementación se realizará un estudio de las diferentes herramientas que se encuentran disponibles para el procesamiento de imágenes y detección de formas básicas. Las herramientas

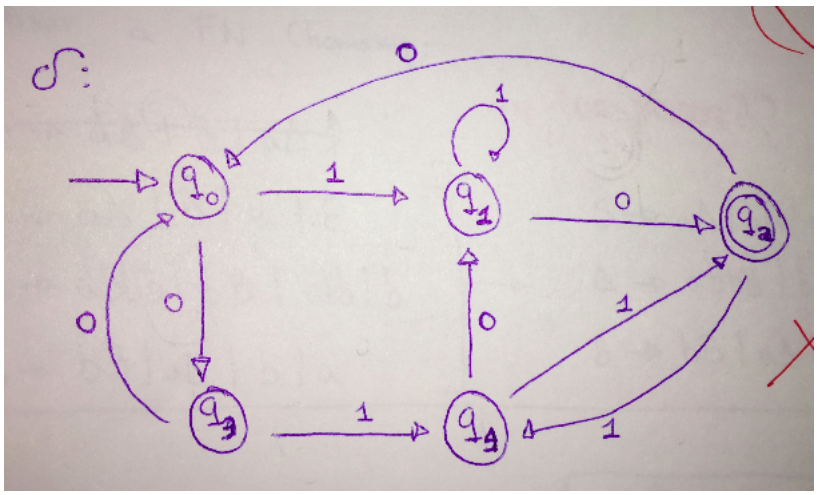


Figura 3.1: Ejemplo Real DFA

más utilizadas en el campo del procesamiento de imágenes son Matlab [42], OpenCV [24], SimpleCV [26] y Octave [23].

3.1. Matlab

Matlab [42] es un lenguaje de alto nivel y entorno interactivo para el cálculo numérico, visualización y programación incluyendo procesamiento de imágenes. Con él, es posible analizar datos, desarrollar algoritmos y crear modelos y aplicaciones. El programador no debe preocuparse en el uso de las bibliotecas, declaración de variables, gestión de memoria u otros problemas de programación a bajo nivel.
Algunas funcionalidades de Matlab para el tratamiento de imágenes son las siguientes:

```
% Lectura desde fichero
  rgb = imread('grafo.jpg');
  imshow(rgb);

% Escala de grises:
  imagenEscalaGrises = rgb2gray(imagen);

% Busqueda de circulos utilizando la transformacion de Hough
  centers = imfindcircles(A, radius)

% Deteccion de lineas utilizando la transformacion de Hough
  BW = edge(imagenEscalaGrises, 'canny');
  H, T, R = hough(BW);
  P = houghpeaks(H, 10, 'threshold', ceil(0.5 + max(H(:))));
  lunes = houghlines(BW, T, R, P, 'FillGap', 5, 'MinLength', 7);
```

Sin embargo, Matlab no es un software de licencia libre por lo que se evitará usarlo en el desarrollo de este proyecto. Por otro lado, en el caso de OSX

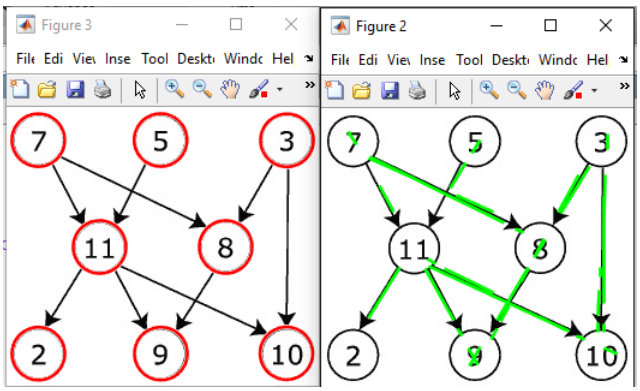


Figura 3.2: Ejemplo Reconocimiento de formas básicas en Matlab

de Apple, puede ser algo incómodo de usar debido a que no tiene soporte por parte de Cocoa [5]. Puede verse un ejemplo de detección de líneas y círculos con Matlab en la Figura 3.2.

3.2. Octave

Octave [23] es un paquete de licencia gratuita que ofrece funciones para el procesamiento de imágenes. El paquete proporciona herramientas para la extracción de características, estadísticas, transformaciones espaciales y geométricas, operaciones morfológicas, filtrado lineal y mucho más.
Octave es una buena alternativa a Matlab para aquellos desarrolladores que empiecen a adentrarse en este tipo de programación. Ofrece similitudes como el intérprete y la posibilidad de ejecutar órdenes en modo interactivo pero no así, el depurador que es primitivo en comparación con Matlab y una gran cantidad de herramientas de las que Octave no dispone.
Matlab tiene un entorno mucho más agradable y es mucho más rápido para muchos tipos de operaciones. Octave por su parte, no tiene una interfaz gráfica de usuario nativa. Se basa en una serie de paquetes que se pueden incorporar al desarrollo del programa. Por último, la compatibilidad desde Octave a Matlab es bastante buena para programas sencillos.

3.3. OpenCV

OpenCV [24] es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Desde que apareció su primera versión alfa en el mes de enero de 1999 se ha utilizado en infinidad de aplicaciones, desde sistemas de seguridad con detección de movimiento, hasta aplicativos de control de procesos donde se

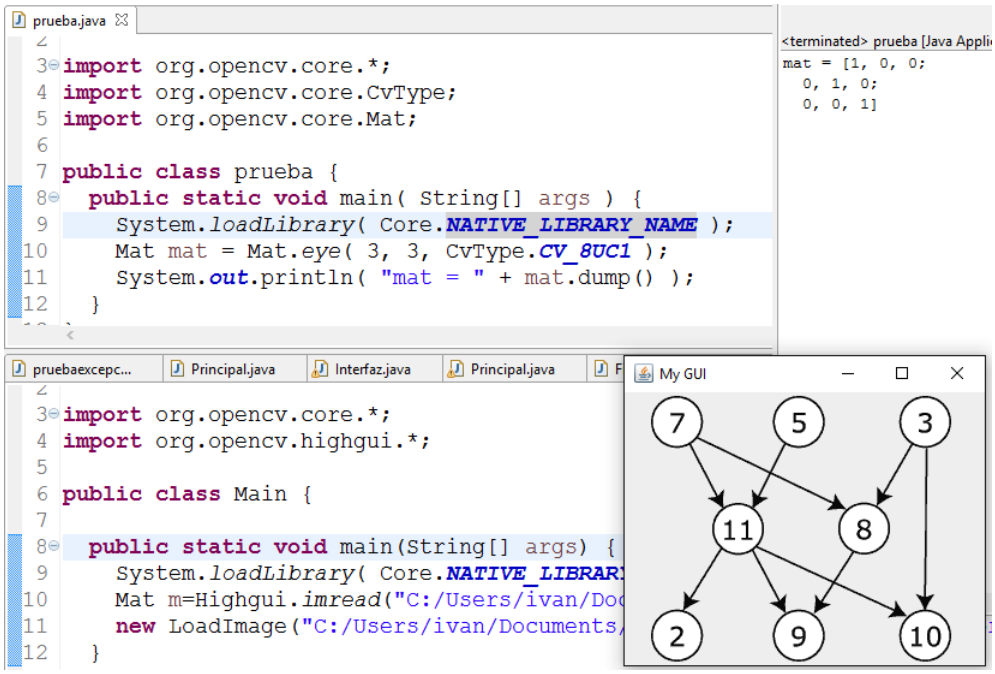


Figura 3.3: Ejemplo OpenCV

requiere reconocimiento de objetos. Está escrita en lenguaje C por lo que los programas desarrollados con esta herramienta se ejecutarán mucho más rápido que otros programas similares escritos en Matlab o SimpleCV [26]. Además, es de licencia gratuita y no existe un IDE particular para utilizarlo.
En este estudio, se ha probado el procesamiento de imágenes sobre el IDE Eclipse [18] bajo el lenguaje de programación Java y el IDE Qt Creator [25] con el lenguaje C++. Desde la versión 2.2.4, OpenCV soporta Java. Para poder utilizarlo se debe instalar la biblioteca como cualquier otro tipo de herramientas y configurar Eclipse agregando la ruta correspondiente. Sin embargo, la mayoría de documentación de OpenCV se encuentra en el lenguaje C++ y Python, por lo que utilizar estos lenguajes es más recomendable si se empieza por primera vez a utilizar esta herramienta. Puede verse un ejemplo de apertura de imagen desde fichero utilizando OpenCV en la Figura 3.3.

3.4. SimpleCV

SimpleCV [26] es un framework de código abierto basado en Python y utilizado para la creación de aplicaciones de visión artificial capaz de acceder a varias bibliotecas de alto rendimiento para visión por computador como OpenCV sin necesidad de aprender en profundidad el formato, espacio de colores o gestión de memoria intermedia.
SimpleCV ofrece un entorno de desarrollo para empezar a realizar procesamiento de imágenes de manera mucho más intuitiva que el resto de herramientas comentadas anteriormente. Sin embargo, SimpleCV es muy dependiente del ta-

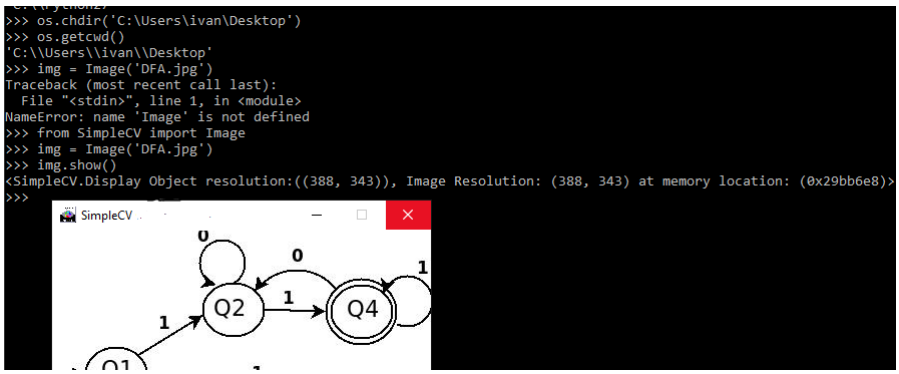


Figura 3.4: Ejemplo de uso de SimpleCV

maño de la imagen, así como de las habilidades del programador.
Si se desea aprender SimpleCV de una forma rápida y fácil, existe un tutorial [27] bastante interesante para empezar a desarrollar programas sencillos.
Además, SimpleCV permite su utilización bajo la interfaz de Shell. Bastará con escribir SimpleCV desde cualquier lugar en el *prompt* del sistema operativo y se comenzará automáticamente su entorno de desarrollo. Puede verse un ejemplo de apertura de imagen desde fichero utilizando SimpleCV en la Figura 3.4 y un ejemplo de detección de lineas en la Figura 3.5.

Provee funcionalidades como:

```
# Carga una imagen
img = Image('C:\Users\ivan\Desktop\grafo.png')

# Escala la imagen
img = img.scale(600, 600)

# Muestra la imagen
img.show()
sleep(2)

# Invierte los pixeles de la imagen
img = img.invert()
print("Invert")
img.show()
sleep(2)

# Deteccion de lineas
lines = img.findLines()
print lines
lines.draw(width = 3)
img.show()
sleep(20)

# Deteccion de circulos
blobs = img.findBlobs()
print blobs
if blobs:
circles = blobs.filter([b.isCircle(0.1) for b in blobs])
princt circles
if circles:
img.drawCircle((circles[-1].x, circles[-1].y),
```

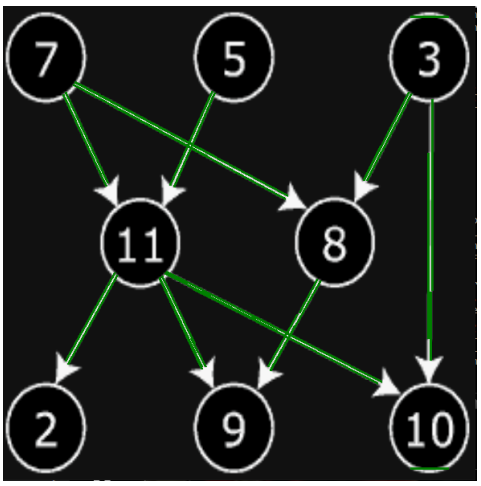


Figura 3.5: Detección de líneas SimpleCV.

```
32            circles[-1].radius(), Color.BLUE, 1)
33
34 img.show()
```

3.5. Comparativa de las herramientas

Analizadas las diferentes herramientas disponibles en la actualidad para el procesamiento de imágenes y detección de formas básicas, se ha podido observar cómo para la detección de formas básicas en una imagen, existen diferentes posibilidades para afrontar el mismo problema. Sin embargo, si se desea procesar algo más complejo como la identificación de una forma dado un patrón, el proceso puede ser bastante más complicado.
En este caso, se ha descartado el sistema Matlab para la detección de formas en la imagen porque, aunque es una herramienta bastante intuitiva y sencilla de utilizar, no es de licencia gratuita. En el caso de OpenCV y SimpleCV, el segundo es un framework sencillo que utiliza OpenCV para la detección de formas. Ambos son de código abierto y ambos, a diferencia de Octave, cuentan con mayor documentación y uso extendido.

Por todo esto, se ha decidido utilizar la biblioteca OpenCV, ampliamente utilizado en el campo de la visión por computador. Con respecto al lenguaje de programación, se planteó el desarrollo de una aplicación móvil para plataforma Android. Sin embargo, se han encontrado problemas con la configuración de OpenCV con el framework Android Studio [4] además de tener poco conocimiento del entorno de desarrollo. Por ello, se ha desarrollado una aplicación de escritorio utilizando el lenguaje de programación C++ y la herramienta Qt Creator [25]. Se ha elegido este lenguaje y no Java, por su amplia documentación con OpenCV y su mayor facilidad para la detección y clasificación de

contornos en imágenes. Aunque se pueda utilizar OpenCV en diferentes lenguajes de programación, algunas funcionalidades de la librería no se definen de la misma manera y la documentación en el lenguaje Java es mucho menor que en el lenguaje C++.

Capítulo 4

DCAFI

DCAFI (*Detector y Corrector de Autómatas Finitos en Imágenes*) es una aplicación de escritorio que permite la **detección de autómatas a partir de imágenes y diagramas dibujados a mano para su codificación y corrección**, comparándolo con un autómata de referencia o sobre un conjunto de cadenas verificando así, que ambos autómatas son equivalentes. La aplicación ha sido diseñada utilizando el entorno *Qt Creator* [25] y el lenguaje de programación C++ [46]. Además, se ha utilizado la biblioteca *OpenCV* [24] con las clases *Highgui* [16], *Core* [6], *ML* [22] y *Imgproc* [19].

DCAFI [31] es capaz de automatizar la corrección de autómatas partiendo de una imagen o fichero en formato de texto plano. A diferencia de herramientas como JFLAP o SELFA, DCAFI ofrece al alumno la posibilidad de comparar los autómatas que cree o diseñe a lo largo de la asignatura con los de sus compañeros o los ofrecidos por el profesorado, fomentando así el aprendizaje autónomo y la resolución de ejercicios prácticos. Por otro lado, la automatización de este proceso ayuda enormemente a la corrección de exámenes por parte del profesorado de la asignatura *Computabilidad y Algoritmia* que actualmente exige analizar y estudiar cada autómata, uno por uno. Además, DCAFI cuenta con un asistente de ayuda que ofrece al usuario la posibilidad de conocer todas las opciones disponibles según en que paso de la detección o corrección se encuentra.

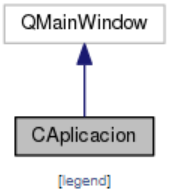
Por otro lado, se ha tratado de utilizar un diseño con una adecuada correlación de colores y tipografías en los diferentes paneles de la aplicación. Adicionalmente, se ha creado un logotipo adaptado al diseño dinámico de DCAFI. Puede verse en la Figura 4.1.

El proyecto ha sido documentado mediante DOXYGEN [8] y está alojado en la plataforma Github [14], tanto el código [31] en C++ de DCAFI como la documentación [32]. Para facilitar la utilización de OpenCV, se ha proporcionado un script de instalación para la versión 3.2. Este script se encuentra junto al código del proyecto [33].

Figura 4.1: Logo de la herramienta DCAFI

DCAFI ha sido desarrollado utilizando la programación orientada a objetos. La clase *CAplicacion* unifica toda la funcionalidad de DCAFI. En la Figura 4.2 puede verse su estructura.

Inheritance diagram for CAplicacion:



Collaboration diagram for CAplicacion:

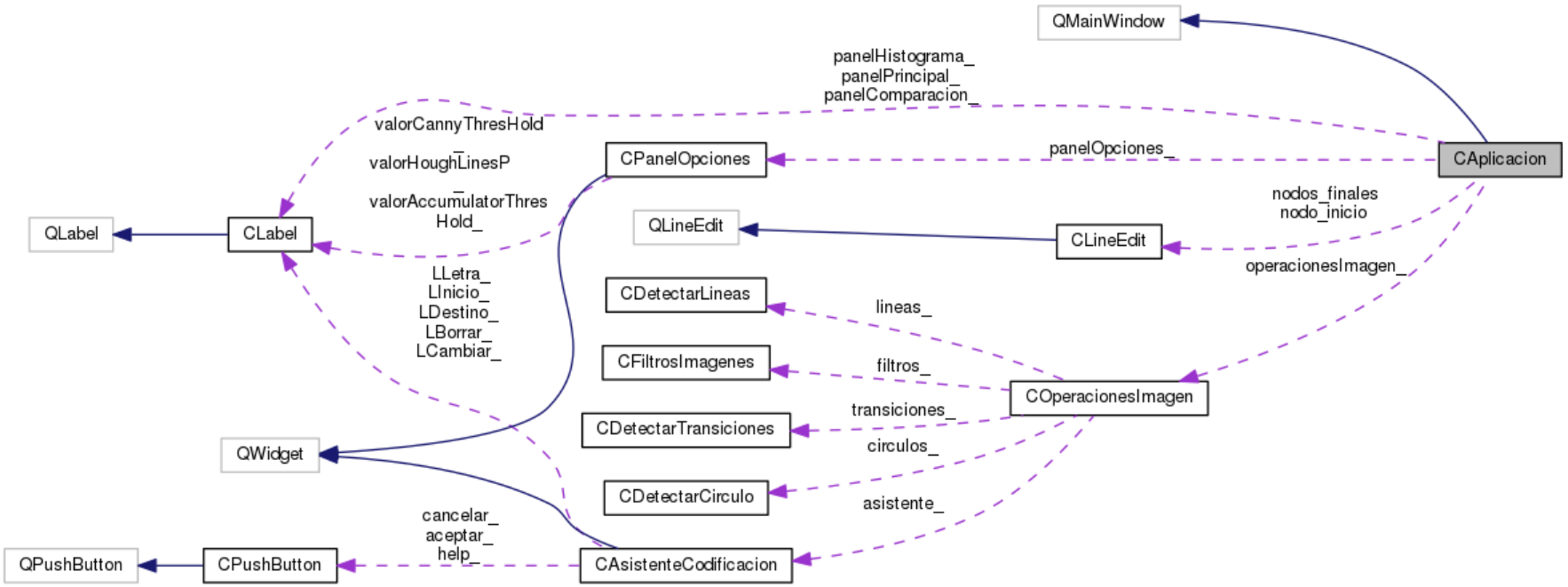


Figura 4.2: Estructura de código, interfaz de DCAFI

Figura 4.3: Iconos de la barra de herramientas

La herramienta cuenta con dos perspectivas de trabajo. Por un lado, la **perspectiva de corrección** permite comparar dos autómatas introducidos previamente determinando si son o no equivalentes. Además, esta perspectiva permite analizar un conjunto de cadenas sobre estos autómatas, transformar los NFAs a DFAs y minimizarlos en caso necesario. Por otro lado, la **perspectiva de detección** permite la extracción y codificación de un autómata a partir de una imagen reconociendo estados, transiciones, etiquetas o símbolos de dichas transiciones y sentido con la ayuda de OpenCV. Este proceso cuenta con diferentes asistentes para facilitar la correcta codificación de la imagen.

La aplicación se apoya en tres fases para comprobar finalmente si dos autómatas son equivalentes o no. La fase de carga es la primera en descubrirse al iniciar la aplicación y se encarga de solicitar por imagen o fichero los autómatas para su corrección. Si la información se transmite a través de una imagen, la herramienta inicia la fase de detección para la extracción y codificación de la imagen. Finalmente, cuando se hayan indicado los dos autómatas a corregir en la fase de carga, se iniciará la fase de corrección que permite comprobar si una cadena es aceptada o no por el autómata especificado, realizar transformaciones de NFA a DFA, minimización de estados y comprobación de equivalencias.

La aplicación cuenta con una barra de herramientas, como puede verse en la Figura 4.3, para facilitar la interacción con el usuario. Los dos primeros elementos corresponden con la carga de una imagen o fichero para su corrección. El resto de los elementos forman parte de la fase de detección de la aplicación.

A continuación, se explicará con mayor detenimiento cada una de las fases en las que se fundamenta DCAFI.

4.1. Fase de carga.

Al ejecutar la aplicación, se cargará la perspectiva corrección que se muestra en la Figura 4.4. En ella apreciamos dos paneles, izquierda y derecha. En el de la izquierda se introducirá el autómata a corregir y en el segundo, el autómata correcto o de referencia. Ambos se podrán introducir en formato de texto plano o mediante una imagen a partir del panel inferior de opciones. También podrá introducirse la información de un autómata para su corrección a partir de:

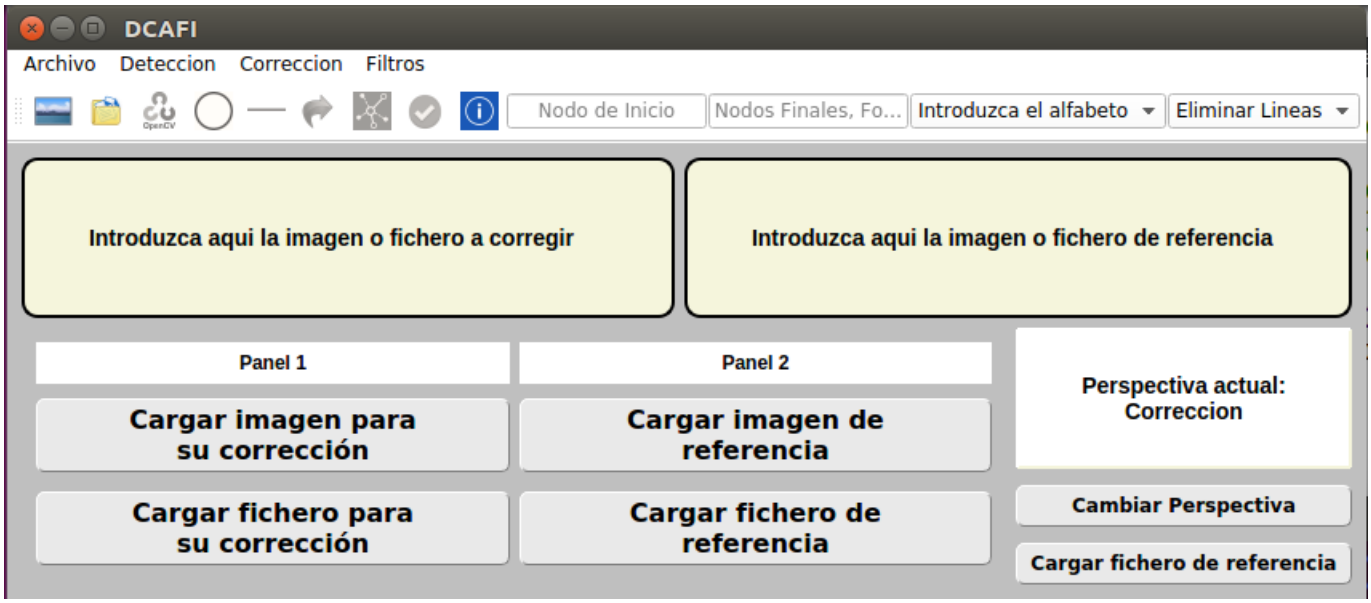


Figura 4.4: Perspectiva corrección, fase de carga

- *Menú - Archivo - Abrir Imagen.*
- *Menú - Archivo - Abrir Fichero.*
- Iconos 1 y 2 de la *barra de herramientas* 4.3.

En el caso de que se cargue una imagen para su codificación, se iniciará la fase de detección. tratará de identificar todos los elementos básicos del diagrama de transiciones. El objetivo final de la fase de carga será tener introducidos dos ficheros, cada uno en su respectivo panel, para comprobar su equivalencia posteriormente en la fase de corrección.

En la zona inferior derecha puede cambiarse la perspectiva de la herramienta eligiendo entre la fase de carga y la de detección. Las opciones del panel inferior serán distintas dependiendo de la perspectiva actual.

Por último, la aplicación contiene una opción que permite generar un nuevo autómata 4.5 especificando, directamente, en el formato texto, sus estados y transiciones:

- *Menú - Archivo - Crear nuevo fichero.*

Este asistente permite guardar la codificación en la ruta deseada o cargarla en el panel izquierdo de la fase de carga para su posterior corrección. Para facilitar la tarea al usuario, esta ventana cuenta con una opción de ayuda que especifica el formato correcto para la codificación de autómatas.

Figura 4.5: Nuevo fichero de codificación

4.2. Fase de detección

Si se desea cargar el autómata a partir de una imagen se iniciará la perspectiva de detección para iniciar su extracción y codificación, tal y como se muestra en la Figura 4.6. En esta perspectiva, se genera un panel principal que muestra la imagen que se desea procesar. Se podrá interactuar con este panel en el proceso de **reconocimiento paso a paso** para añadir, eliminar o modificar detecciones directamente sobre la imagen y así facilitar el proceso de extracción de información.

En la parte inferior de la ventana hay un panel de opciones que contiene tres variables, las dos primeras ()*CannyThresHold* y *AccumulatorThresHold*) para la detección de círculos y la última (*AccumulatorThresHold*) para la detección de líneas. Además, en la zona inferior derecha hay un panel de información que permite volver a la perspectiva de corrección y restablecer los valores por defecto de *CannyThresHold*, *AccumulatorThresHold* y *HoughLinesP*.

En esta perspectiva, se genera un panel principal que muestra la imagen que se desea procesar. Se podrá interactuar con este panel en el proceso de **reconocimiento paso a paso** para añadir, eliminar o modificar detecciones directamente sobre la imagen y así facilitar el proceso de extracción de información.

En la parte inferior de la ventana hay un panel de opciones que contiene tres variables, las dos primeras para la detección de círculos y la última para la detección de líneas. Además, en la zona inferior derecha hay un panel de información

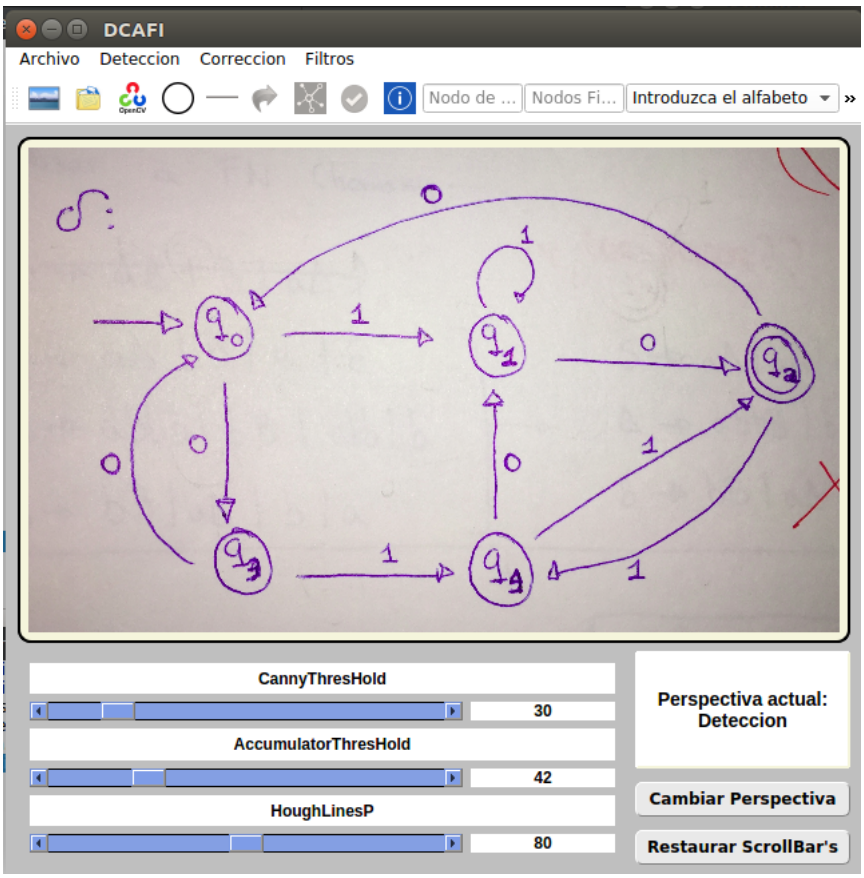


Figura 4.6: Perspectiva detección

que permite volver a la perspectiva de corrección y restablecer los valores por defecto de *CannyThresHold*, *AccumulatorThresHold* y *HoughLinesP*.

También se podrá acceder a la perspectiva de detección a través de:

- *Menú - Archivo - Abrir Imagen*
- *Icono 1 de la barra de herramientas - Abrir Imagen* (véase la Figura 4.3).
- *Cambiar Perspectiva*

Junto con la fase de detección, se abrirá una ventana de ayuda para informar al usuario del funcionamiento de la perspectiva. Para realizar el proceso de detección, debe especificarse si el alfabeto es numérico o de letras. Tras la detección de estados, debe introducirse el nodo de inicio y los nodos finales en los campos correspondientes 4.7.

En el caso del símbolo ε, que representa las epsilon-transiciones o transiciones vacías, tras ser detectado correctamente se representará en el fichero de codificación del autómata como un $\sim$. Por último, si se desea volver a cargar la imagen original, existe una opción en:

Figura 4.7: Datos necesarios para la detección

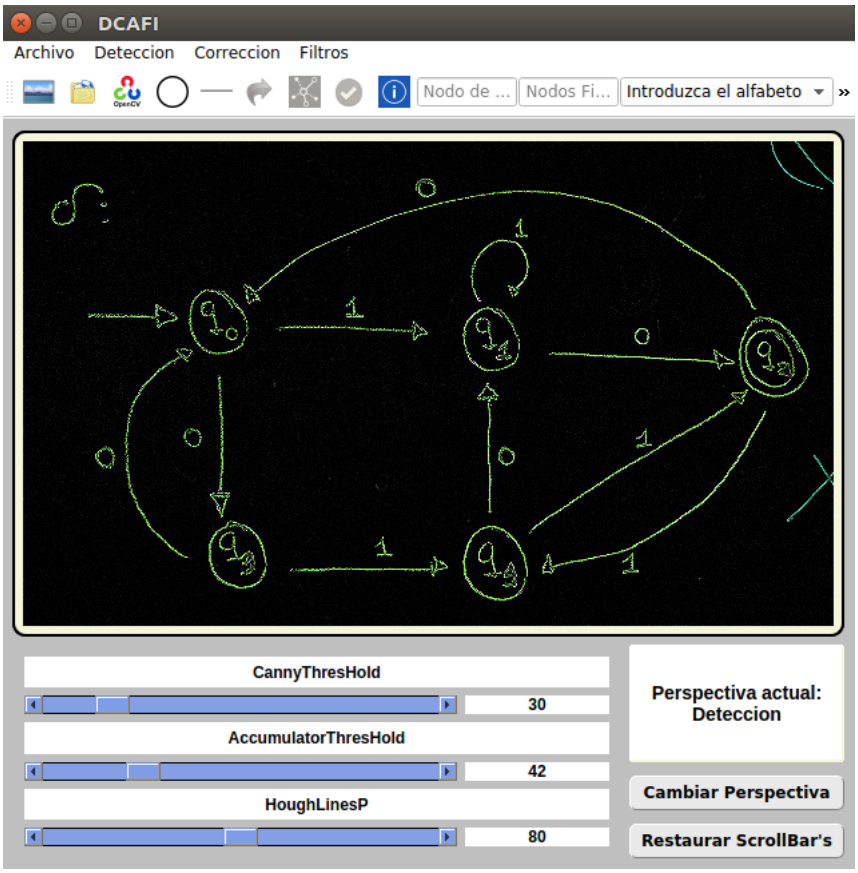


Figura 4.8: Filtro Gaussiano y Sobel sobre una Imagen

- *Menú - Detección - Cargar última imagen*

De esta manera, no será necesario volver a acceder a la ruta del archivo a través de la opción *Abrir Imagen*.

Adicionalmente, la herramienta DCAFI ofrece al usuario la posibilidad de experimentar con los diferentes filtros que se han utilizado en la fase de detección de formas simples con OpenCV. En la mayoría de procedimientos se ha transformado la imagen a escala de grises, aplicado un filtro Gaussiano para reducir el nivel de ruido o el operador Sobel para la detección de bordes en imágenes. Además, podrá calcularse el histograma de la imagen si así se desea. Para poder utilizar los filtros sobre imágenes, se tendrá que estar bajo la perspectiva de detección y haber previamente cargado una imagen. Un ejemplo de uso puede verse en la Figura 4.8.

A continuación, se explican las dos formas que existen para realizar la detec-

ción de una imagen y las distintas maneras de refinarla.

4.2.1. Detección paso a paso

Esta detección consta de 5 pasos.

1. **Detección de círculos:**
   Primeramente y tras haber cargado la imagen, se podrá empezar la detección de círculos a través de:

   - Opción del menú: *Detección - Detectar Círculos.*
   - Icono numero 4 de la barra de herramientas: *Detectar Círculos* 4.3.

   Para conseguir la detección se ha hecho uso de la función *HoughCircles* de la librería *Highgui* [16] de OpenCV, capaz de encontrar círculos en una imagen en escala de grises usando una modificación de la transformada de Hough, `CV_HOUGH_GRADIENT`.

   Sintaxis: [12]
   ```
   HoughCircles(InputArray image, OutputArray circles, int method,
   double dp, double minDist, double cannyThresHold = 100,
   double accumulatorThresHold = 100, int minRadius = 0, int
   maxRadius = 0 )
   ```

   La variable *cannyThresHold* representa el umbral superior utilizado en el detector interno de la función *Canny* [10] para la detección de círculos en imágenes. La variable *accumulatorThresHold* representa el umbral superior para la detección central de círculos. Ambas variables pueden ser modificadas por el usuario en el panel de opciones inferior si así lo cree conveniente para mejorar la detección de contornos en la imagen.
   Los elementos detectados se guardarán en un vector de tipo *Vec3f* de la forma *(x, y, radio)*. Los valores *x e y* corresponden con el punto central del círculo. Un ejemplo de detección de estados con la herramienta DCAFI puede verse en la Figura 4.9.

   Si la detección no fuera correcta, se podrá refinar la búsqueda de dos formas distintas:

   a) Cambiando los valores de los Scrollbar *CannyThresHold* y *AccumulatorThresHold*. Si estos valores son bajos, se detectarán falsos círculos en la imagen y radios de diversa amplitud. Si es muy alto, es posible que no se detecten todos los elementos.
   La modificación de estas variables afectará directamente sobre los

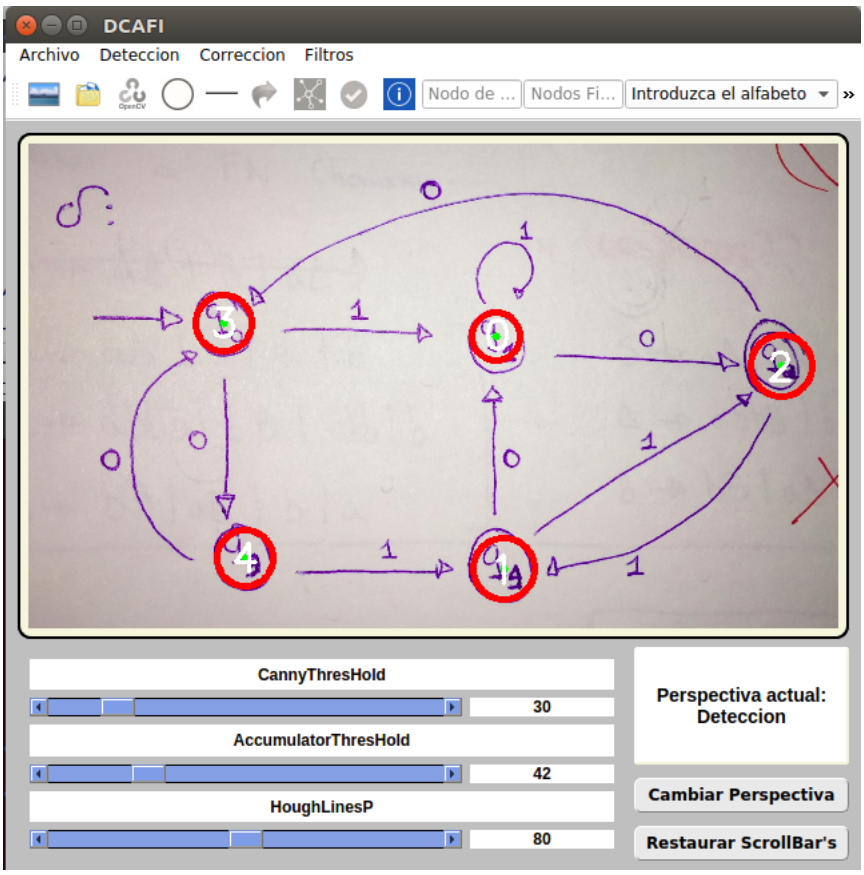


Figura 4.9: Estados detectados imagen

elementos detectados hasta el momento 4.10.

Podrán restaurarse los valores por defecto a partir del botón *Restaurar ScrollBar's* en la parte inferior derecha de la aplicación. Estos valores han probados sobre un conjunto de ejemplos para la detección de formas básicas.

b) Se pueden añadir o eliminar círculos interactuando directamente con la imagen. Sobre la detección de círculos hecha, si se pulsa sobre un círculo ya detectado, este se eliminará, si se pulsa sobre una zona de la imagen en la que no ha sido detectado ningún elemento, se añadirá uno nuevo en esa posición. Cabe destacar que **solo se podrá interactuar con los círculos cuando se acaben de detectar. Nunca en otra fase de la detección.**

2. **Detección de líneas:**
Hecha la detección de círculos, el programa habilita la posibilidad de detectar líneas. Esto se podrá hacer mediante:

   - Opción del menú: *Detección - Detectar Líneas.*
   - Icono número 5 de la barra de herramientas: *Detectar Líneas* 4.3.

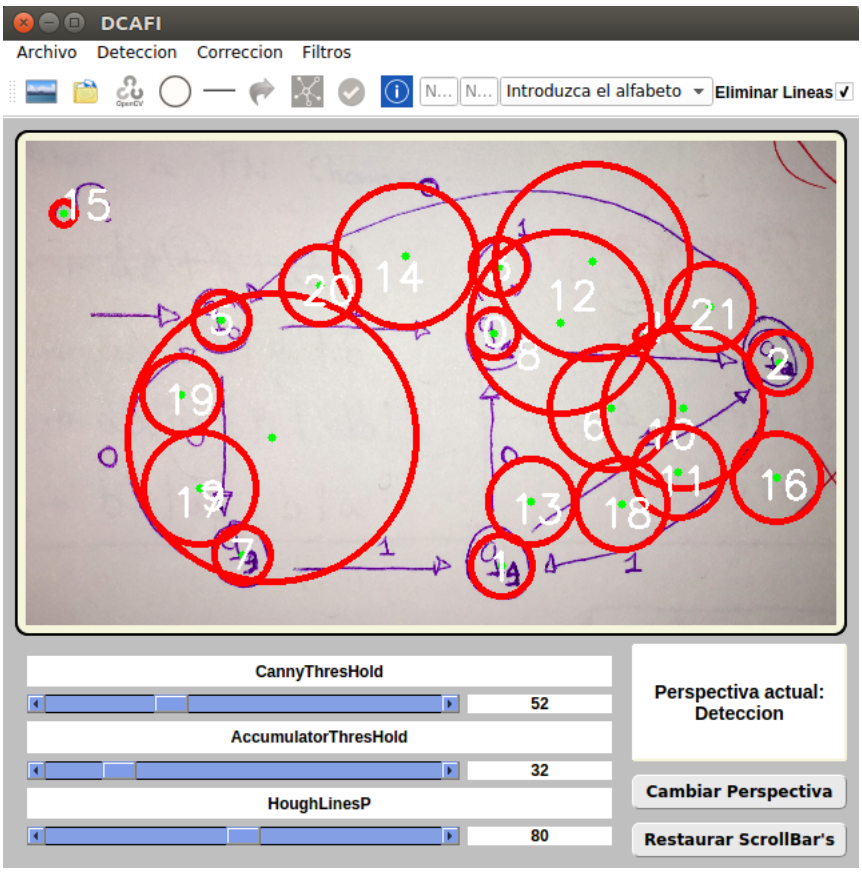


Figura 4.10: Ejemplo de mala detección de estados

Sin embargo, antes deberá introducirse el número del estado de inicio del autómata y los estados finales en los campos correspondientes, en la barra de herramientas. Para la detección de las líneas se utiliza la función *HoughLinesP* de la librería *Highgui* de OpenCV que busca segmentos de líneas en una imagen binaria usando la transformada de Hough probabilística.

Sintaxis: [13]

```
void HoughLinesP(InputArray image, OutputArray lines, double
rho, double theta, int threshold, double minLineLength=0, double
maxLineGap=0 )
```

La variable *HoughLinesP* del panel de opciones corresponde con la variable *threshold* de la función *HoughLinesP* y representa el número mínimo de intersecciones necesario para considerar una línea válida. Este valor puede ser modificado por el usuario en el panel de opciones inferior si así lo cree conveniente para mejorar la detección de contornos en la imagen. Se puede ver un ejemplo de detección de líneas con la herramienta DCAFI en la Figura 4.11.

Los elementos detectados se guardan en un vector de tipo *Vec4i* almacenando el punto x, y inicial y x, y final de la línea.

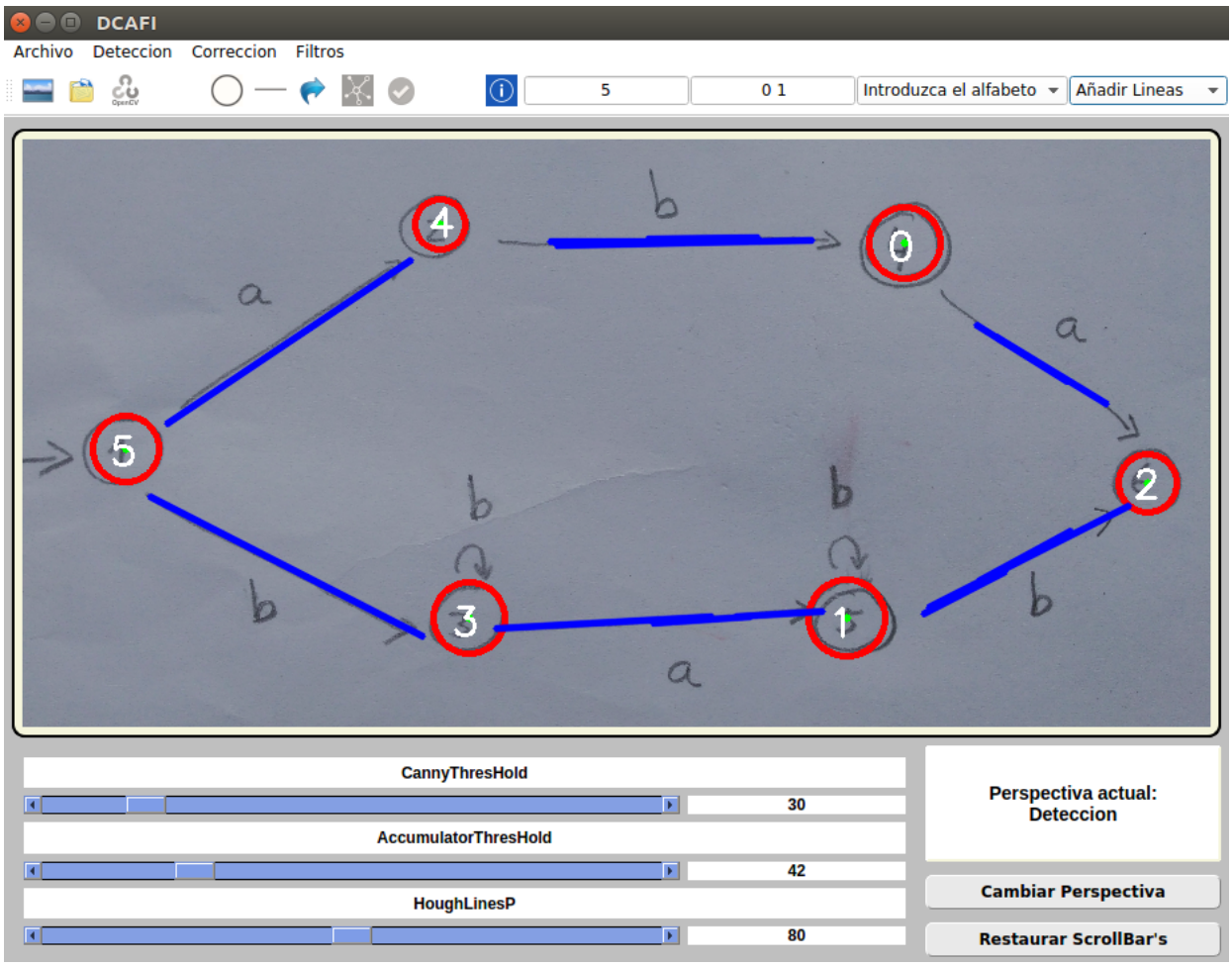


Figura 4.11: Líneas detectadas en la imagen

Si la detección no fuera correcta, se podrá refinar las líneas detectadas de dos formas distintas:

a) Cambiando el valor del *Scrollbar HoughLinesP* en la parte inferior del asistente. Si el valor de esta variable fuera bajo, se detectarán segmentos de líneas más pequeños, siendo algunas de ellas, falsos positivos. Si el valor fuera muy elevado, no se detectarán todos los contornos deseados. La modificación de esta variable afectará directamente sobre las líneas detectadas en el momento. Podrán restaurarse los valores por defecto a partir del botón *Restaurar ScrollBar's* en la parte inferior derecha de la aplicación.

b) Se pueden añadir o eliminar líneas interactuando directamente sobre la imagen. En la parte superior derecha del asistente, hay un *ComboBox* que permite decidir cuál de las dos opciones es la deseada 4.12.

Si se decide elegir la opción de eliminar líneas bastará con pulsar

Figura 4.12: Opción eliminar añadir lineas

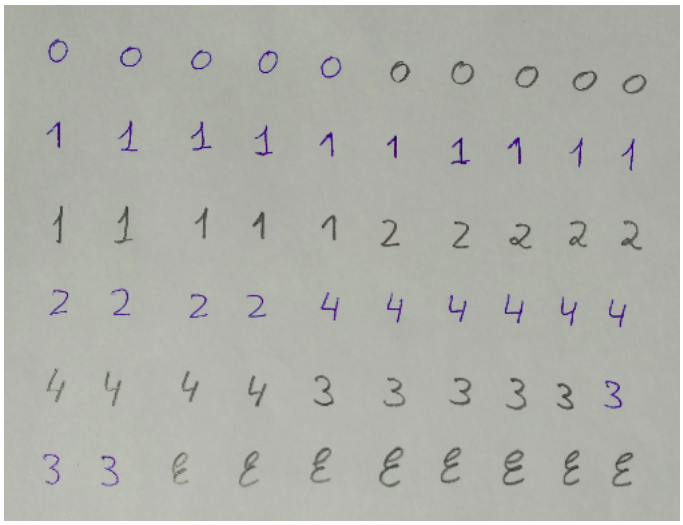


Figura 4.13: Alfabeto de Números.

sobre ellas para descartarlas. Si la opción elegida es añadir nuevas líneas, deberá introducirse el punto inicial y final de estas pulsando sobre la zona deseada de la imagen. Cabe destacar que **solo se podrá interactuar con las líneas cuando se acaben de detectar. Nunca en otra fase de la detección**

3. **Detección de transiciones.**
Una vez realizada la detección de líneas, el programa habilita la posibilidad de detectar transiciones. Para poder realizar esta clasificación de caracteres se ha hecho un entrenamiento de alfabetos mediante los métodos del fichero *generarClasificador.cpp*. En este proceso, se han leído dos imágenes con los caracteres deseados, alfabeto numérico y de letras (véanse las Figuras 4.13 y 4.14).

Posteriormente, se han detectado sus contornos con la función de OpenCV *findContours* [11] que busca contornos en una imagen binaria y se ha clasificado uno a uno por teclado. Puede verse el proceso en la Figura 4.15. Tanto la información de la clasificación resultante como las imágenes de cada contorno han sido guardadas en la carpeta *training* en formato *.xml*

Este entrenamiento debe hacerse correctamente ya que los ejemplos utilizados deben contener todas las formas posibles de caracteres escritos a

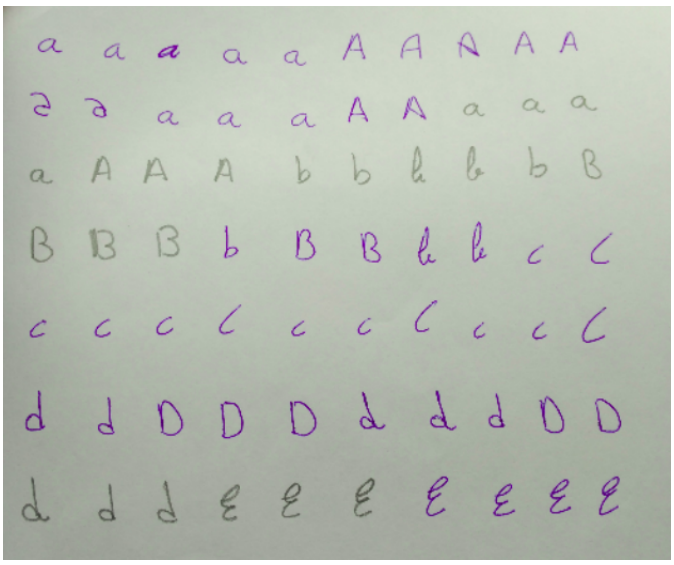


Figura 4.14: Alfabeto de Letras.

mano utilizados para dibujar autómatas.

Hecha la clasificación previa de caracteres y hecha la detección de líneas, se podrá realizar la detección de transiciones mediante:

- Opción del menú: *Detección - Detectar Transiciones.*
- Icono 6 de la barra de herramientas: *Detectar Transiciones*4.3.

Esta detección se hace utilizando de nuevo la función *findContours* de la librería *Highgui* y *KNearest* [21] de la librería *mll* capaz de identificar contornos en una imagen dado una clasificación previa [22].

Puede verse un ejemplo de detección de transiciones con la herramienta DCAFI en la Figura 4.16.

Si la detección no fuera correcta, se podrá refinar la búsqueda interactuando directamente con la imagen:

a) Para eliminar una transición se podrá pulsar sobre ella en la imagen. De esta manera, se descarta.

b) Si desea añadir una nueva transición, se podrá pulsar en una zona de la imagen donde no haya una definida. Se deberá introducir por teclado el carácter. Cabe destacar que **solo se podrá interactuar con las transiciones cuando se acaben de detectar. Nunca en otra fase de la detección**.

4. **Sentido de las transiciones.**

Figura 4.15: Generación de clasificador de letras y números

Terminada la detección de transiciones se podrá detectar el sentido de las transiciones. Esto se podrá hacer mediante:

- Opción del menú: *Detección - Detectar Sentido de las Transiciones.*
- Icono número 7 de la barra de herramientas: *Detectar Sentido de las Transiciones* (véase la Figura 4.3).

El usuario podrá cambiar en este punto, el sentido de algunas transiciones de forma interactiva, pulsando sobre el estado que deseamos que sea el inicial y a continuación el nodo destino. Puede verse un ejemplo de detección del sentido de las transiciones de un autómata en la Figura 4.17.

5. **Codificación de la imagen.**
Hecha la detección de sentido de las transiciones se habilita la opción de

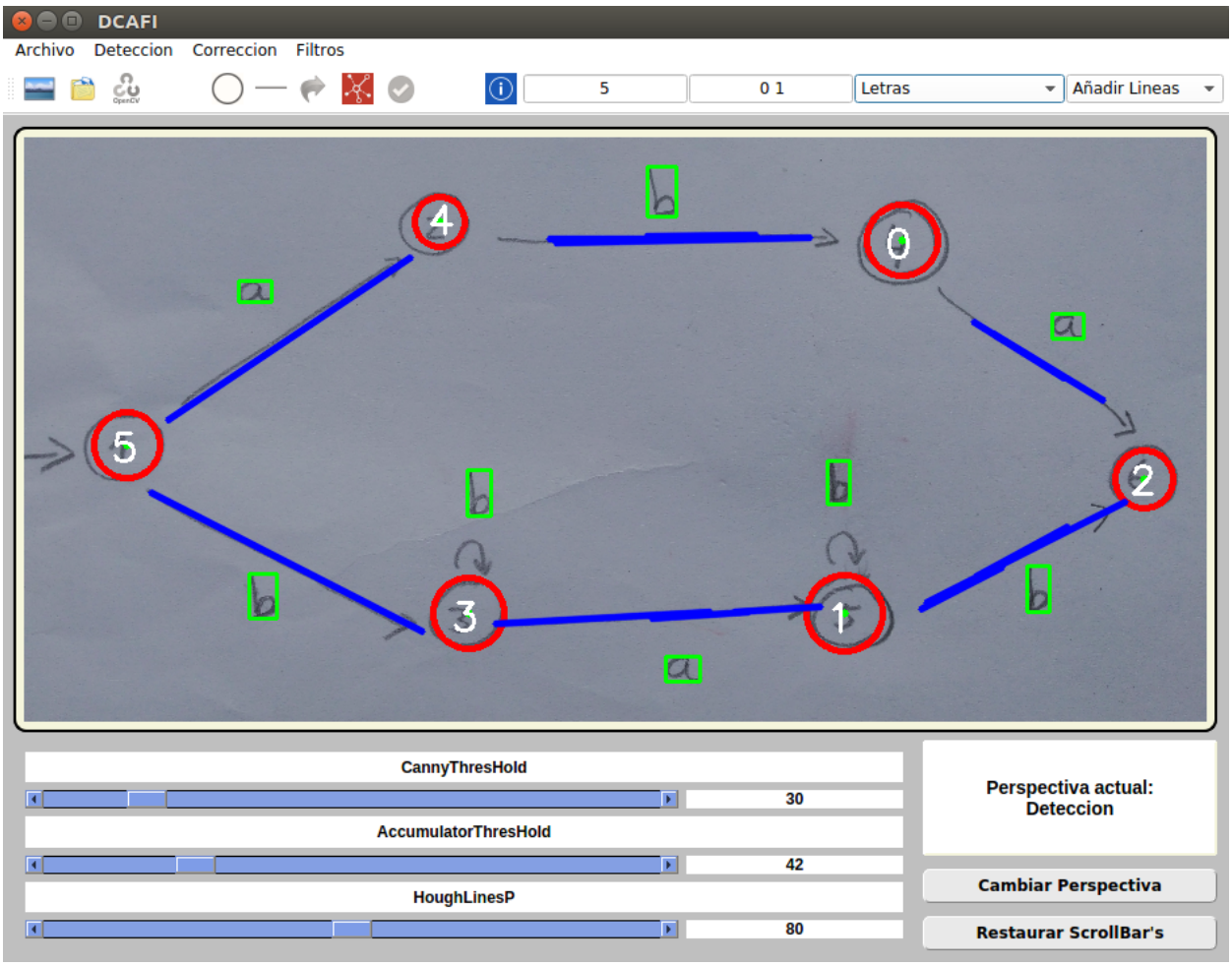


Figura 4.16: Transiciones detectadas imagen

Confirmar la detección. Esto se podrá hacer mediante:

- Opción del menú: *Detección - Confirmar detección*.
- Icono número 8 de la barra de herramientas: *Confirmar detección* (véase la Figura 4.3).

Esta opción intenta buscar un orden entre todos los elementos detectados y abre un asistente con la codificación resultante.

Este asistente de codificación permite variar cualquier característica de la imagen pudiendo cambiar el valor de las transiciones, eliminar, añadir y cambiar el sentido de alguna de ellas 4.18. De esta forma, se garantiza poder obtener el autómata correctamente si ocurriera algo con la detección y la interacción con la imagen.

Además, se puede:

- Cancelar la codificación.

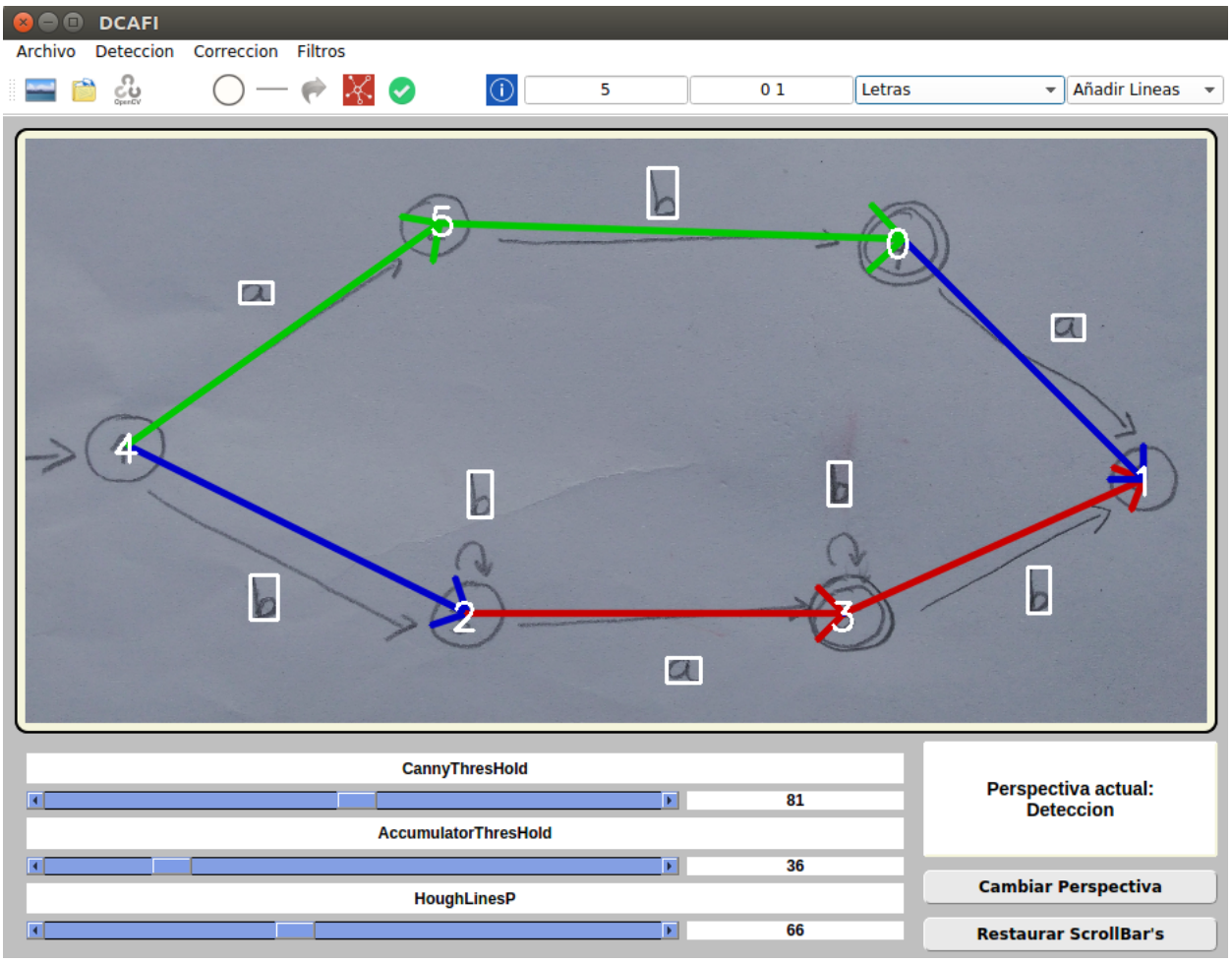


Figura 4.17: Sentidos Detectados Imagen

- Guardar el autómata como fichero en formato de texto plano.
- Cargar la codificación en la perspectiva corrección para su evaluación.

4.2.2. Detección automática

Opción para la codificación del autómata representado en la imagen que hace la detección de círculos, líneas, transiciones y sentidos de forma automática. Es posible que alguno de los elementos encontrados no sea correcto. Esta opción se encuentra en:

- Opción del menú: *Detección - Procesar Imagen.*
- Icono número 3 de la barra de herramientas: *Procesar imagen* (véase la Figura 4.3).

Para mejorar la detección, en el proceso de codificación se han realizado unas serie de consideraciones:

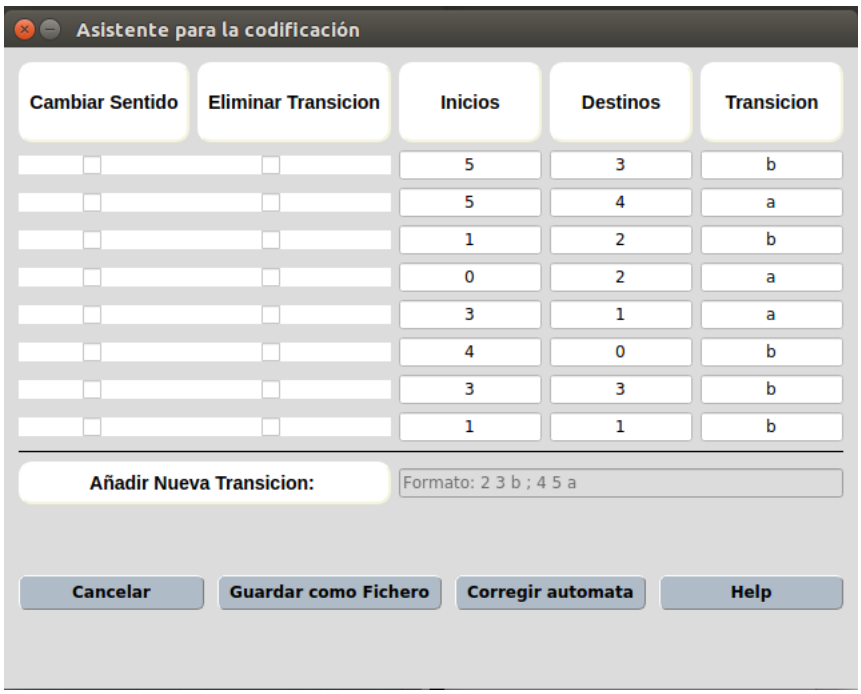


Figura 4.18: Asistente para la codificación de la imagen.

- La eliminación de las transiciones encontradas dentro de círculos (estados).
- Eliminar los círculos demasiado grandes.
- Realizar un filtrado de las líneas si hay varias de ellas superpuestas, quedándose con la más larga de todas ellas.
- Realiza la búsqueda de letras en el punto medio de dos círculos que tienen una línea en común.
- Para las transiciones que no queden clasificadas, busca si alguna de ellas se encuentra justo encima de algún estado reconociendo transiciones a sí mismo.
- Ordena de derecha a izquierda todo lo detectado. El sentido de las transiciones será de izquierda a derecha.

Al aceptar esta detección, el programa ofrece un asistente final como en la detección paso a paso, para confirmar todos los datos de la imagen permitiendo cambiar el valor de las transiciones, eliminar, añadir y cambiar el sentido de algunas de ellas. El resultado de la codificación podrá guardarse como fichero o cargarse en la perspectiva corrección para iniciar la evaluación del mismo.

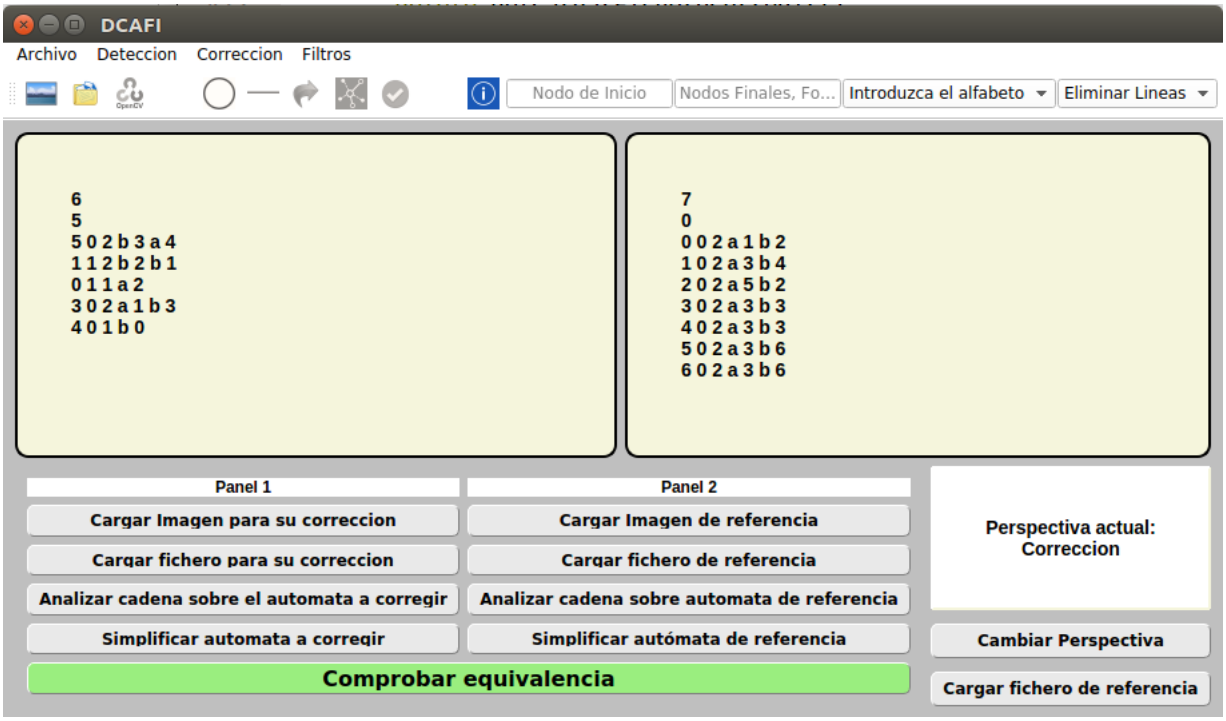


Figura 4.19: Fase de corrección.

4.3. Fase de corrección.

La fase de corrección comienza al completar la fase de carga de algún autómata en la ventana principal de la aplicación. En caso de introducirse un autómata en formato imagen, se realizará una fase de detección previa considerando la extracción y codificación del autómata.

Puede observarse la disposición de elementos de la fase de corrección en la Figura 4.19. En ella, se observan los dos paneles de la fase de carga con sus autómatas ya leídos. El primero corresponde con el autómata que se desea corregir y el segundo con el autómata correcto o de referencia. En el caso de que se haya cargado solo uno de los autómatas, se cargarán unicamente las opciones que puedan aplicarse a esta codificación.

A continuación se explican brevemente las opciones individuales que se pueden aplicar sobre un autómata en la fase de corrección:

- **Analizar una cadena**
  Analiza si una cadena es aceptada o no por el autómata. Puede verse un ejemplo de uso en la Figura 4.20. Además, esta opción indicará a través de una ventana, el alfabeto y estados de muerte del autómata.

- **Simplificar autómata**
  Simplifica el autómata aplicando el algoritmo de construcción de subcon-

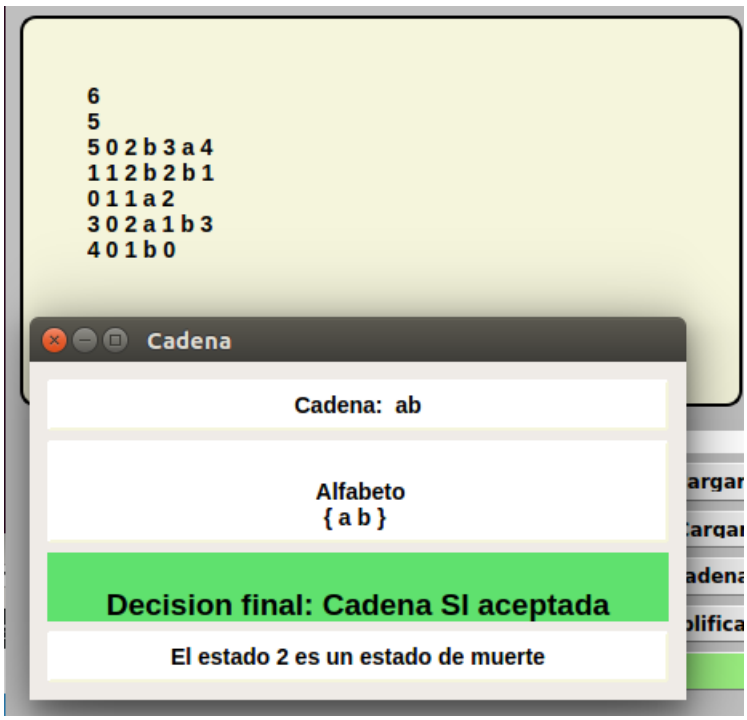


Figura 4.20: Análisis de cadena sobre un autómata.

juntos [2] para convertir el posible NFA en DFA. A continuación, aplica el algoritmo de minimización de estados [3] consiguiendo el DFA mínimo.

Si tanto el autómata a corregir como el de referencia han sido cargados, se habilita la posibilidad de comprobar la equivalencia entre ambos. Esta opción minimiza ambos NFA/DFA aplicando el algoritmo de construcción de subconjuntos y el algoritmo de minimización de estados. Posteriormente comprueba si los dos autómatas son iguales comunicando al usuario el resultado del análisis.

4.4. Menú de la aplicación DCAFI.

Por último, a continuación se resume la finalidad de todas las opciones del menú de la aplicación DCAFI.

1. Archivo:

   a) *Cargar desde imagen.* Abre una imagen para su detección y su posible corrección.

   b) *Cargar desde fichero.* Abre un autómata para corregirlo .

   c) *Crear nuevo fichero.* Abre un asistente para guardar o corregir el autómata que se escriba.

   d) *About Qt.* Información sobre el Framework utilizado.

e) *About.* Información sobre el Proyecto.

f) *Salir.*

2. Detección:

   a) *Detectar Círculos.* Detecta los círculos en una imagen.

   b) *Detectar Líneas.* Detecta las líneas en una imagen.

   c) *Detectar Transiciones.* Detecta las transiciones en una imagen.

   d) *Codificar Imagen.* Aplicado las detecciones anteriores, codifica el autómata.

   e) *Confirmar Imagen.* Confirma la detección hecha

   f) *Procesar Imagen.* Aplica las opciones a, b, c y d.

   g) *Cargar la Imagen Original.* Carga la imagen antes de haber detectado elementos.

3. Corrección:

   a) *Simplificar Autómatas*

   b) *Corregir Autómata*

4. Filtros: Repaso de filtros utilizados en el desarrollo del proyecto.

   a) *Filtro Gray*

   b) *Filtro Gaussiano*

   c) *Filtro Mediana*

   d) *Filtro Sobel*

   e) *Filtro Laplaciano*

   f) *Generar Histograma*

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

Con el presente Trabajo Fin de Grado se ha conseguido crear una aplicación que automatiza el proceso de corrección de autómatas finitos partiendo de una imagen inicial dibujada a mano. De esta manera, la aplicación es capaz de analizar imágenes extrayendo y codificándolas como autómata finito. Para ello se detectan estados, líneas, transiciones y sentidos en un autómata.

En la fase de detección tiene que introducirse el nodo inicial, los nodos finales y el alfabeto utilizado en las transiciones. Además, un asistente confirma todos los datos con el usuario antes de guardarlo y/o prepararlo para una posible fase de corrección, minimización o prueba con cadenas.

El proceso de detección ha ocupado gran parte del proyecto siendo bastante complicado conseguir la extracción y codificación de algunas imágenes debido a una mala iluminación o una gran cantidad de estados y poca distancia entre los elementos. Sin embargo, y para disminuir estos problemas se han añadido diferentes asistentes como la interacción directa con la imagen para añadir y eliminar elementos. Asimismo, se ha hecho complejo filtrado de los contornos detectados para descartar datos discordantes consiguiendo finalmente un sistema correcto capaz de adaptarse y solventar problemas en la detección de contornos y la eliminación de falsos elementos.

Para poder encontrar la metodología correcta se han estudiado diferentes herramientas utilizadas para la simulación de autómatas finitos. Además, se ha investigado en profundidad la biblioteca OpenCV para el tratamiento de imágenes y generación de clasificadores de contornos [20, 36, 38, 43, 45].

El proceso de corrección de autómatas ha resultado ser algo más sencillo. Se ha implementado un *algoritmo de construcción de subconjuntos* para convertir el posible autómata no determinista (NFA) en autómata determinista (DFA). Posteriormente, se ha implementado el *algoritmo de minimización de estados* para minimizarlo. La fase de corrección permite además, analizar cadenas sobre autómatas y finalmente, objetivo final del presente Trabajo Fin de Grado, comprobar si dos autómatas son equivalentes o no.

En este TFG he aprendido muchísimo sobre la visión por computador y he conocido todas las oportunidades de OpenCV. Por otra parte, se ha trabajado muchísimo sobre ventanas y eventos sobre el lenguaje de programación C++.
En general, estoy bastante contento con el trabajo realizado aunque me hubiera gustado tener más tiempo para poder diseñar una detección para máquinas de Turing. Sin embargo, estoy bastante satisfecho. Ha sido un TFG largo y costoso.

5.2. Líneas futuras

No obstante, el presente Trabajo Fin de Grado ofrece variedad de propuestas y futuras mejoras. Una de las propuestas que no se han podido finalmente implementar es la detección de máquinas de Turing a partir de imágenes. De esta manera, podrían considerarse otro tipo de lenguajes formales modificando ligeramente la detección y el formato actual de DCAFI.
También se propuso el diseño de una versión App. Está no se llevó a cabo por la escasa documentación para la utilización de OpenCV para Android y el poco conocimiento previo para la utilización de las funciones necesarias. Actualmente, y tras haber estudiado en profundidad esta librería de visión por computador me siento capacitado para hacerla.
Otras mejoras podrían ser la incorporación de una base de ejemplos bien definida que ayude al alumnado con el aprendizaje para la simulación de autómatas finitos y lenguajes formales. Por último, la utilización de una base de datos, podría servir para almacenar el resultado de las correcciones de los diferentes autómatas diseñados por el alumnado. DCAFI ofrece la posibilidad de seguir mejorando y llegar a convertirse en una herramienta para el apoyo a la docencia de lenguajes formales y la corrección automática a partir de imágenes o capturas de un diagrama de estados.

Capítulo 6

Conclusions and futures work

6.1. Conclusions

With the given Final Degree Project , it was achieved the creation of an application that automates the process of correcting finite automata starting from an initial handmade drawn image. This way, the application is able to analyze images extracting and encoding it as a finite automaton. In order to this, states, lines, transitions and senses will be detected in an automaton.
During the detection phase, the initial node, the final nodes and the alphabet used in the transitions, have to be introduced. Moreover, an assistant confirms all the data to the user before saving and/or preparing it for a possible correcting, minimasing phases or chain tests.
The detection process has been a huge part of the project, being quite hard to achieve the extraction and coding of some images due to bad lighting or large amount of state and little distance between the elements. However, and to reduce these problems, different assistants have been added as the direct interaction with the image for adding or removing elements. Likewise, a complex filter of the detected edges has been done in order to dismiss conflicting data and finally get a proper system able to adapt and solve any problems regarding the detection of edges and deletion of false elements.

In order to find the right methodology, different tools used for simulating finite automata have been studied. Futhermore, a deep research has been carried out on the OpenCV library for the image treatment and generation of edge classifying [20, 36, 38, 43, 45].
The automata correction process has turned to be quiet simpler. A construction algorithm of subsets to turn the possible non-deterministic automaton (NFA) into deterministic automaton (DFA) has been developed. Afterwards, a minimization algorithm of states has been implemented to minimize it. The correction

phase also allows to analyze chains on automata and finally get the goal of this Final Degree Project, to confirm if two automata are equivalent or not.

In this work, I have enormously learned about the computer vision and got to know all the possibilities of OpenCV. On the other hand, a lot of work has been done on windows and events about programming language C++. Overall, I am quite glad with the achieved work although I would like to have had more time to be able to design a detection for Turing machines. However, I can say I am satisfied. It has been a long and hard-working project.

6.2. Future work

Nevertheless, the given Final Degree Project offers a huge variety of proposals and future improvements. One of the suggestions that finally was not able to be introduced is the detection of Turing machines based on images. This way, other types of automata and formal languages could have been considered, slightly modifying the detection and the present format of DCAFI. The design of an App version was also proposed. This was not finally carried out since the lack of documentation on the use of OpenCV for Android and little background knowledge for using the needed functions. Currently, and after having deeply studied this library on computer vision, I feel capable to do it.

Other improvements could have been the insertion of a well-defined example base that helps to the students with the learning of simulation of finite automata and formal languages. Finally, the use of a database could be used to store the result of the corrections of the different automata designed by the students. DCAFI offers the opportunity to keep improving and get to be a tool for supporting teaching of formal languages and automatic correction starting from images or captures of a state diagram.

Capítulo 7

Presupuesto

A continuación se presenta un presupuesto orientativo para el desarrollo de la aplicación DCAFI.

Componente	Cantidad	Coste Unitario	Coste Total
Mano de Obra			
Analista desarrollador de Software	270h	12,00€	3240,00€
Hardware			
Depreciación o uso informático de computadora	270h	0,50€	135,00€
Software			
Libreria OpenCV	1	0,00€	0,00€
Qt-Creator	1	0,00€	0,00€
Doxygen	1	0,00€	0,00€
CMake	1	0,00€	0,00€
Gedit	1	0,00€	0,00€
Git	1	0,00€	0,00€
JFLAP	1	0,00€	0,00€
LaTeX	1	0,00€	0,00€
MikText 2.9	1	0,00€	0,00€
Servicios y suministros			
Energía Eléctrica	45kW/h	0,20€	5,00€
Internet	1 plan	17,00€	51,00€
Transporte	0	0,00€	0,00€
		Sub Total	3431,00€
		15 % de imprevistos	514,65€
		Total	3945,65€

Tabla 7.1: Presupuesto

Bibliografía

[1] A Collection of Tools for Making Automata Theory and Formal Languages Come Alive. https://www.cs.duke.edu/csed/rodger/papers/cse97flap.pdf. [Online; 02-Julio-2017].

[2] Algoritmo Construccion de Subconjuntos. https://es.wikipedia.org/wiki/Construcci%C3%B3n_de_subconjuntos. [Online; 02-Julio-2017].

[3] Algoritmo Minimización de Estados. http://www.sc.ehu.es/jiwnagom/MAC1-ALF/MAC-archivos/Tema2-parte3.pdf. [Online; 02-Julio-2017].

[4] Android Studio. https://developer.android.com/studio/index.html?hl=es-419. [Online; 02-Julio-2017].

[5] Cocoa. https://es.wikipedia.org/wiki/Cocoa_(inform%C3%A1tica). [Online; 02-Julio-2017].

[6] Core OpenCV. http://docs.opencv.org/2.4/modules/core/doc/core.html. [Online; 02-Julio-2017].

[7] dk.brics.grammar. http://www.brics.dk/grammar/. [Online; 02-Julio-2017].

[8] DOXYGEN. http://www.stack.nl/~dimitri/doxygen/. [Online; 02-Julio-2017].

[9] FSM. https://en.wikipedia.org/wiki/Finite-state_machine. [Online; 02-Julio-2017].

[10] Funcion Canny OpenCV. http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html. [Online; 02-Julio-2017].

[11] Funcion Find Contours. http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html. [Online; 02-Julio-2017].

[12] Funcion HoughCircles OpenCV. http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghcircles#houghcircles. [Online; 02-Julio-2017].

[13] Funcion HoughLinesP OpenCV. http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghlines. [Online; 02-Julio-2017].

[14] Github. https://github.com/. [Online; 02-Julio-2017].

[15] Graphviz. http://www.graphviz.org/. [Online; 02-Julio-2017].

[16] Highgui OpenCV. http://docs.opencv.org/2.4/modules/highgui/doc/highgui.html. [Online; 02-Julio-2017].

[17] HoughTransform. https://es.wikipedia.org/wiki/Transformada_de_Hough. [Online; 02-Julio-2017].

[18] IDE Eclipse. http://www.eclipse.org/users/. [Online; 02-Julio-2017].

[19] ImgProc OpenCV. http://docs.opencv.org/2.4/modules/imgproc/doc/imgproc.html. [Online; 02-Julio-2017].

[20] Introducción a la visión por computador: desarrollo de aplicaciones con OpenCV. https://courses.edx.org/courses/course-v1:UC3Mx+ISA.1x+1T2016/info. [Online; 02-Julio-2017].

[21] Knearest OpenCV. http://docs.opencv.org/2.4/modules/ml/doc/k_nearest_neighbors.html. [Online; 02-Julio-2017].

[22] ML OpenCV. http://docs.opencv.org/2.4/modules/ml/doc/ml.html. [Online; 02-Julio-2017].

[23] Octave. https://www.gnu.org/software/octave/. [Online; 02-Julio-2017].

[24] OpenCV. http://opencv.org/. [Online; 02-Julio-2017].

[25] Qt-Creator. https://www.qt.io/ide/. [Online; 02-Julio-2017].

[26] SimpleCV. http://simplecv.org/. [Online; 02-Julio-2017].

[27] Simplecv tutorial. http://tutorial.simplecv.org/en/latest/. [Online; 02-Julio-2017].

[28] Miguel y Blanes García José Antonio Alberto de la Encina Vara, Ballesteros Martínez and Samer Nabhan Rodrigo. TALFi, una herramienta para teoría de autómatas y lenguajes formales. http://eprints.ucm.es/9771/. [Online; 02-Julio-2017].

[29] Anders Møller at Aarhus University. dk.brics.automaton. `http://www.brics.dk/automaton/index.html`. [Online; 02-Julio-2017].

[30] J.G. Brookshear. *Teoría de la Computación: Lenguajes Formales, Autómatas y Complejidad.* Addison-Wesley, 1993.

[31] Iván García Campos. DCAFI (Detector y Corrector de Automatas Finitos en Imágenes. `https://github.com/alu0100693737/Codigo-TFG`. [Online; 02-Julio-2017].

[32] Iván García Campos. Documentación Proyecto DCAFI. `https://alu0100693737.github.io/Codigo-TFG/html/`. [Online; 02-Julio-2017].

[33] Iván García Campos. Script Instalacion OpenCV. `https://github.com/alu0100693737/Codigo-TFG/blob/master/instalacion_opencv.sh`. [Online; 02-Julio-2017].

[34] Jose Jesus Castro-Schez, Jose Jesus Castro-Schez, Julian Hortolano, and Alfredo Rodriguez. Designing and Using Software Tolls for Educational Purposes: FLAT, a Case Study . `https://www.researchgate.net/publication/224355194_Designing_and_Using_Software_Tools_for_Educational_Purposes_FLAT_a_Case_Study`. [Online; 02-Julio-2017].

[35] José Jesús Castro Sánchez. Software para la enseñanza y aprendizaje de la materia Teoría de Autómatas y Lenguajes Formales. `http://portal.esi.uclm.es/selfa/`. [Online; 02-Julio-2017].

[36] Sohil Dalal, Jayneil Patel. *Instant OpenCV Starter (1).* Packt Publishing, May 2013.

[37] Adán Dzib-Tun, Cinhtia González, and Michel García. Una herramienta didactica interactiva para la enseñanza-aprendizaje de los autómatas finitos deterministas . `https://www.researchgate.net/publication/273671521_Una_herramienta_didactica_interactiva_para_la_ensenanza-aprendizaje_de_los_automatas_finitos_deterministas`. [Online; 02-Julio-2017].

[38] Noelia Vállez Enano; Ismael Serrano Gracia; Jesus Salido Tercero; José Luis Espinosa Aranda; Oscar Deniz Suarez; Gloria Bueno García. *Learning Image Processing with OpenCV.* Packt Publishing, March 26, 2015.

[39] Akhilesh Hegde, Akshay G Joshi, and Viraj Kumar. Ranking Student Ability and Problem Difficulty Using Learning Velocities . `https://www.researchgate.net/publication/282739118_Ranking_Student_Ability_and_Problem_Difficulty_Using_Learning_Velocities`. [Online; 02-Julio-2017].

[40] R. Motwani J. E. Hopcroft and J. D. Ullman. *Introducción a la teoría de Autómatas, Lenguajes y Computación.* Addison-Wesley, 2002.

[41] D. Kelley. *Teoría de Autómatas y Lenguajes Formales.* Prentice-Hall, 1995.

[42] MathWorks. Matlab. `http://es.mathworks.com/products/matlab/?requestedDomain=es.mathworks.com`. [Online; 02-Julio-2017].

[43] Amgad Muhammad. *OpenCV Android Programming By Example.* Packt Publishing, December 15, 2015.

[44] Susan Roger. JFlap. `http://www.jflap.org/`. [Online; 02-Julio-2017].

[45] Joseph Howse; Steven Puttemans; Quan Hua; Utkarsh Sinha. *OpenCV 3 Blueprints.* Packt Publishing, November 10, 2015.

[46] Bjarne Stroustrup. C++. `https://es.wikipedia.org/wiki/C%2B%2B`. [Online; 02-Julio-2017].

[47] Juan José Tamagnini, Salvador Valerio Cavadini, Pablo Luis Berdaguer, Diego Alejandro Cheda, Fernando Manuel Pachado Sánchez, and Mario Petersen. Sepa Project, Software para la Enseñanza de Parsing . `http://www.ucse.edu.ar/fma/sepa/`. [Online; 02-Julio-2017].