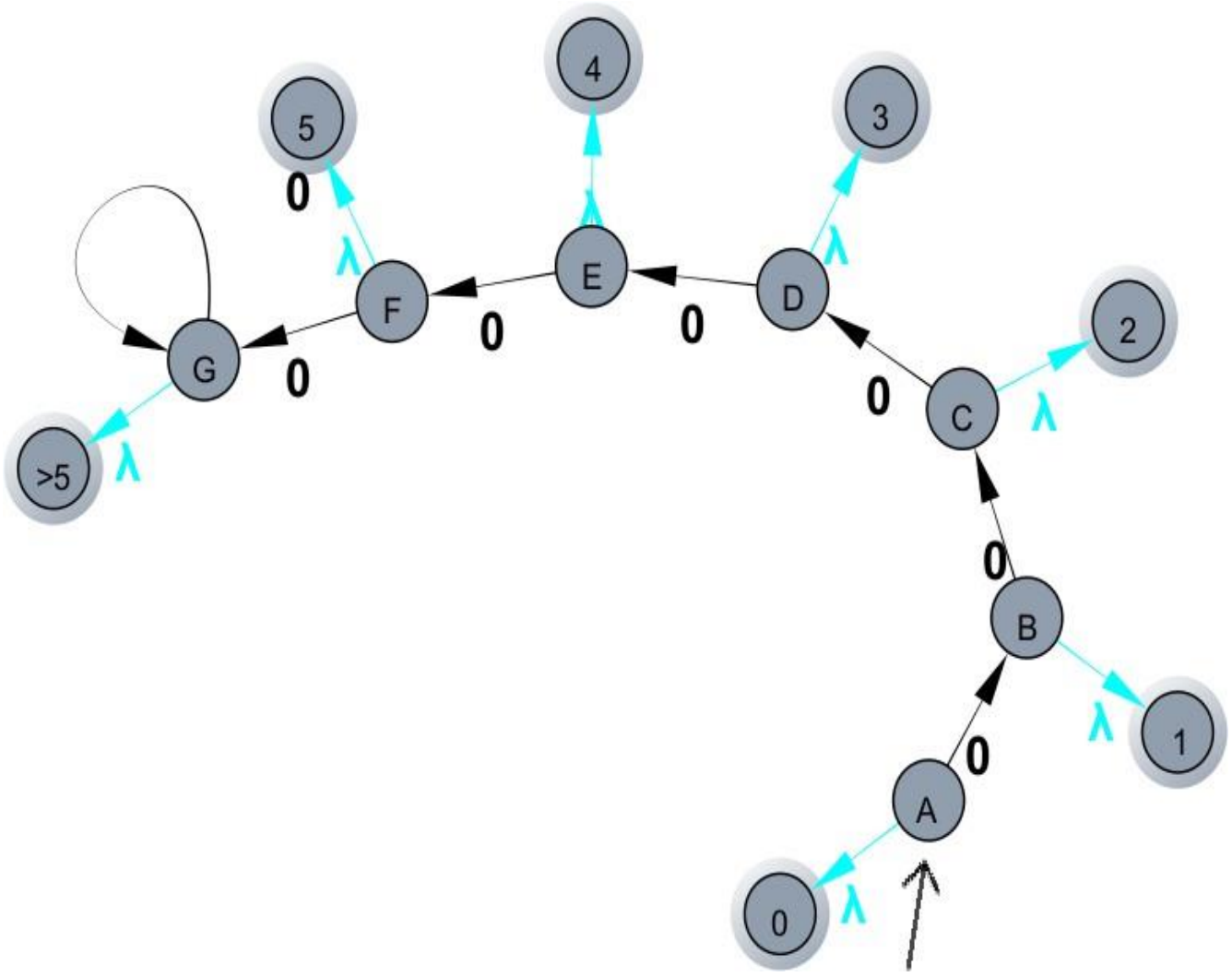
SAS (Simple Automaton Simulator)

JTorr



ÍNDICE

Para la realización de esta práctica, se ha empleado el patrón de diseño “Modelo-Visa-Controlador”, simplemente, por la organización que nos aporta y, de cuya estructura, nos valdremos para organizar la explicación esta memoria:

Empezaremos explicando las clases que componen el ‘Modelo’ y que constituyen la implementación básica para el control de autómatas. Finalmente, se explicará de forma breve los distintos elementos de la “Vista” y del “Controlador”.

Terminaremos exponiendo algunos ejemplos que deberían cubrir la totalidad de situaciones posibles mostrando las soluciones que arrojan frente a diferentes ‘*inputs*’.

[1] Serán accesibles al tratar de importar un autómata, durante la ejecución del programa.

El Modelo

Para la realización de esta práctica y, en vista a tener 2 clases de autómatas diferentes, primero se ha hecho una clase abstracta 'AbstractAutomata.java' que implementa las estructuras de datos y funciones básicas:

- Las listas de estados y sus transiciones ("Transactions"):

```
private HashSet<String> initialStates;
private HashSet<String> finalStates;
private HashSet<Transaction> transactionsList;
```

(Nótese que el acceso a dichas estructuras solo será posible mediante dichas funciones; pues son 'private')

- Los métodos para manipular dichos datos: GETTERs y SETTERs

```
// SETTERs ------------------------------------------------------------------
public void addInitialState(String initialStateX){...}
public void addFinalState(String finalStateX){...}
protected boolean addTransaction(String initialState, Character command, HashSet<String> nextStates){...}
public boolean addTransaction(String initialState, Character command, String nextState){...}

// GETTERs ------------------------------------------------------------------
public HashSet<String> getInitialStates(){...}
protected HashSet<String> getNextStates(String initialState, Character command){...}
protected HashSet<Transaction> getTransactionsList(){...}
private Transaction getTransaction(String initialState, Character command){...}

// UTILs --------------------------------------------------------------------
public boolean isFinalState(String state){...}
```

Así como el soporte necesario para ejecutar el autómata:

```
// KORE ---------------------------------------------------------------------
public boolean runAutomata(String inputString){...}
protected abstract boolean validate(String inputString);
```

Además, a medida que se ha ido avanzando en el desarrollo de la práctica, se ha agregado el soporte necesario para guardar (en memoria) los pasos (entre estados) que sigue el autómata, del que se distinguen las dos modalidades en la que se puede mostrar el resultado:

- Directamente:

```
private HashMap<Integer, String> finalSolutionDict;
protected void updateFinalSolution(int stepIndex, String commandsSequence){...}
public String getFinalSolution(){...}
```

- Paso a paso ("step by step"):

```
private String operationsList_str = "";
protected void addOperation(String operationCode){...}
protected void addOperation_transactionCommand(char command){...3 lines }
public ArrayList<String> popOperationsList(){...}
```

Con el fin de facilitar la limpieza se ha creado un 'Lenguaje' [2] que permite conocer qué ha ocurrido en cada 'step by step' de un autómata:

```
// Step by Step Iteration-'Language'
public static final String OPERATION_CODE__COMMAND_INVALID = "<CX>";
public static final String OPERATION_CODE__FINAL_NODE_INVALID = "<BX>";
public static final String OPERATION_CODE__FINAL_NODE_VALID = "<VALID>";
```

(Los símbolos/"command"s que determinan el desplazamiento entre nodos se guardan entre corchetes.)

Ambas estructuras, tendrán que ser empleadas por las clases que hereden de 'AbstractAutomata.java' para un correcto funcionamiento.

Con la clase 'Abstracta' definida, se construyeron los dos tipos de autómatas pedidos:

- AutomataDeterminista.java (Autómata Finito Determinista):

  Que simplemente "capa" o restringe la flexibilidad de agregar más de un estado inicial y de tener múltiples transiciones mediante un mismo símbolo ("Command"); partiendo del mismo estado.

```
// GETTERs ------------------------------------------------------------------
private String getInitialState(){...}
private String getNextState(String initialState,char command){...}
```

(Los GETTERs solo devolverán el primer elemento de la lista que devuelve 'AbstractAutomata')

```
// SETTERs ------------------------------------------------------------------
@Override
public boolean addTransaction(String initialState, Character command, String nextState){...}
```

("addTransaction" no permite agregar una lista de 'nextStates' a la vez)

- AutomataNoDeterminista.java (Autómata Finito No Determinista):
  - Incorpora soporte para las transiciones lambda que lo caracterizan ("Lambda Transactions"):
    Y las agrupa en una estructura de datos específica; filtrado que realiza 'saveLambdaTransactions()':

```
private HashSet<Transaction> lambdaTransactions;
private void saveLambdaTransactions(){...}
```

    Este filtrado es facilitado por el hecho de que al importar un autómata, las transiciones lambda se guardan usando un método específico que guardan el valor de su símbolo/'command' como 'null':

```
public void addLambdaTransaction(String initialState, HashSet<String> finalStates) {...}
public void addLambdaTransaction(String initialState, String finalState) {...}
```

[2] Ver las "Observaciones Finales" del "ANEXO" para una mejor comprensión sobre la elección de este 'Lenguaje'

La implementación de dicho 'soporte' para transiciones lambda se vertebra mediante el siguiente método:

```
protected void addLamdaStates(HashSet<String> statesList){...}
```

Que permitirá incluir todos aquellos "estados lambda" (fruto de transiciones lambda) en base al estado inicial (no lambda) en cada iteración entre estados, sin tener que "gastar" o "consumir" un símbolo/'command' en el proceso, tal y como ocurre entre transiciones normales.

- Por último, cabe mencionar, que para esta clase fue necesario crear un método recursivo que inicia la función abstracta 'validate()' y de la que podemos observar que el primer parámetro es el estado inicial, mientras que el segundo, son los símbolos que quedan, de las iteraciones entre estados, que realiza el autómata.

```
private boolean validateRecursive(String initialState, String partialTextImput){...}
```

Finalmente, las transiciones entre estados son representadas por la clase 'Transaction.java', de la que podemos destacar:

- Sus atributos:

```
private String initialState;
private HashSet<String> finalStates;
private Character command;
```

Donde:

  - El símbolo lo hemos llamado 'command' y es de tipo 'Character' para poder asignarle el valor 'null' y así diferenciar las transiciones lambda del resto.
  - Solo puede existir un estado inicial por transición, mientras que pueden hacerlo más de un estado final (Esta lista está "capada" para el 'AutomataDeterminista' que en cuyo caso siempre cogería el primer elemento).

- Su método 'toString()', cuyo uso se restringe al ámbito de la depuración de código.

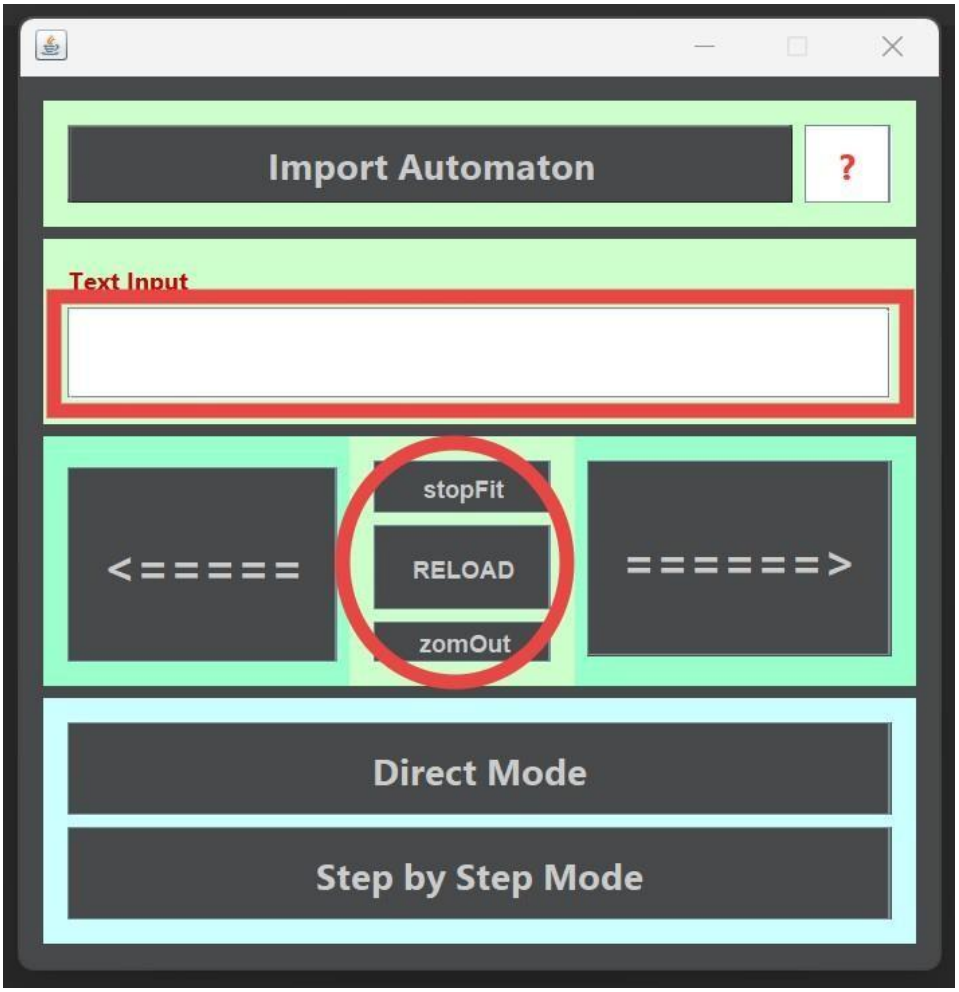
Con esto, termina la explicación del desarrollo y funcionamiento de las clases principales que permiten la ejecución de un autómata finito determinista y no determinista.

A continuación, se explicarán brevemente el resto de las clases que integran la práctica:

VISTA

Además de las especificaciones requeridas de la práctica, se destacan 2 partes especiales:

(Cabe mencionar, que las flechas del modo "paso a paso" también se pueden controlar con las fechas del teclado)



(ControlPane.java)

La primera consiste en un '*textInput*' que, además, durante el modo "paso a paso" remarcará el símbolo de la iteración actual. La segunda, consiste en una serie de botones que, mejoran la visualización de los grafos que representan al autómata, pues la librería ('*GraphStream*'), a veces, no cuadra correctamente el grafo que representa al autómata.

(NOTA: En caso de querer modificar el 'textInput' durante el modo "paso a paso", es necesario hacerle doble clic)

Controlador

Se han creado dos controladores: el principal y aquel que se encarga del 'textInput' mencionado anteriormente.

- "*ControladorTextInput.java*"[3] : encargado de alterar el modo en el que se encuentra el 'textInput' (como un "display" o como un "input"), así como del estilo (color) con el que aparecerá el 'input' en dicho 'textInput' durante el modo "paso a paso".
- "*ControladorAutomáta.java*": es el controlador principal; se encarga de gestionar las acciones que pueden ocurrir en la vista, entre las que encuentran:
    - El modo "paso a paso" o el modo "directo" (que muestra la solución directamente).
    - La representación gráfica del autómata y de diferentes estilos[4] (color, imágenes, etc.) que se aplican a sus componentes (haciendo uso de la librería 'GraphStream').
    - Determinar el símbolo por el que se encuentra durante el modo "paso a paso" y transmitírselo al "ControladorTextInput.java" para su coloreado.
    - Importar[5] y ejecutar el autómata.

Hay que destacar que la creación del "*ControladorTextInput.java*" tiene como propósito: una mayor claridad estructural, para no sobrecargar (más de lo que ya está) el controlador principal.

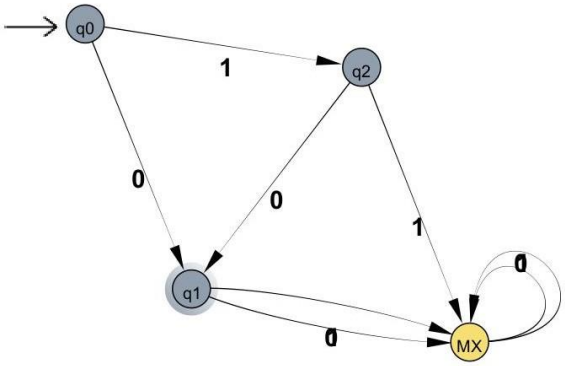
[3] Si no introducimos nada de texto en el 'textInput', éste no se bloqueará. Esto evita error cuya causa se desconoce y que impide escribir texto al volver a desbloquearlo.
[4] Se localizan en la carpeta "StyleSheets" del proyecto.
[5] Desde la carpeta "dataset" del proyecto.

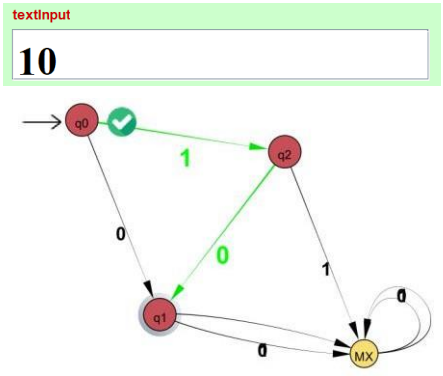
EJEMPLOS

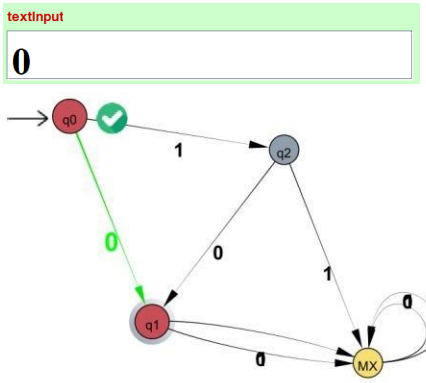
A continuación, se expondrán diferentes ejemplos que, cubrirán la totalidad de casos que se plantean en la práctica (se podrá la solución “directa” y no la solución “paso a paso”), para ello, empezaremos con los ejemplos propuestos y terminaremos con otros más complejos.

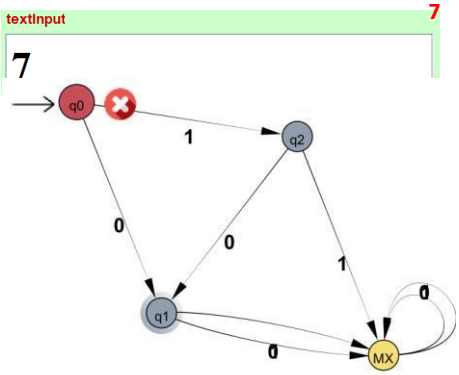
Empezaremos por los ejemplos:

Ejemplo 1 [6]





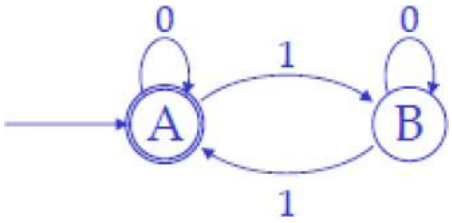


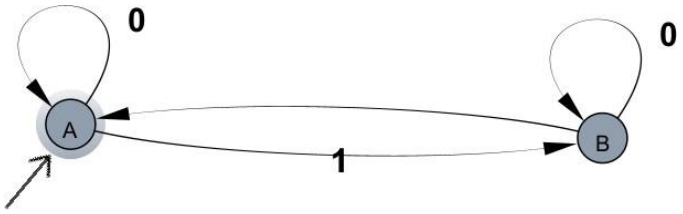


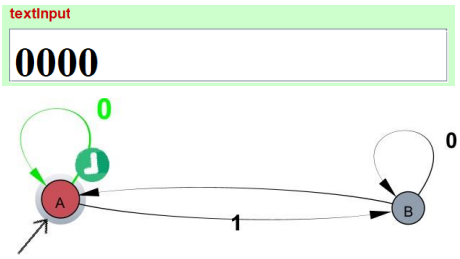
[6] La superposición es inevitable, a menos, que se haga de forma manual lo cual afectaría a su escalabilidad. Se ha optado por remarcar el camino por el que pasa y, en caso de duda, siempre se podrá ver con claridad en el modo “paso a paso”.

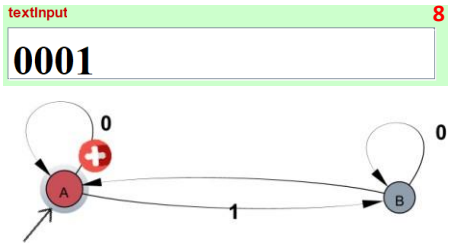
[7] Si no hay una solución válida, simplemente muestra una “X” en el estado inicial. Para conocer hasta qué estado llegó, se tendría que usar el modo “paso a paso” mediante las flechas de la ventana de control o con las del teclado.

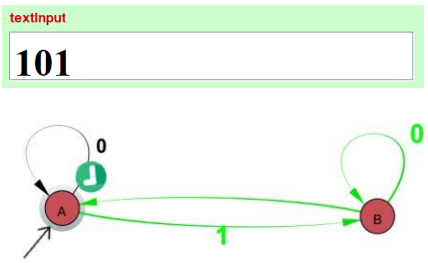
Ejemplo 2



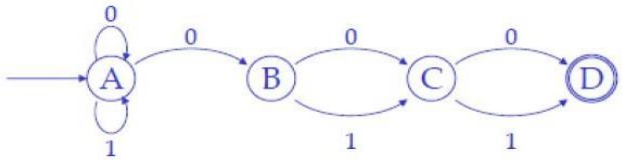


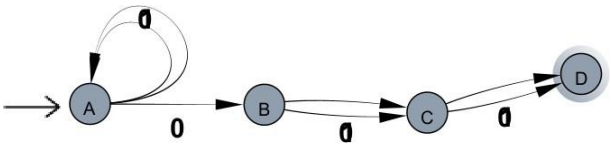


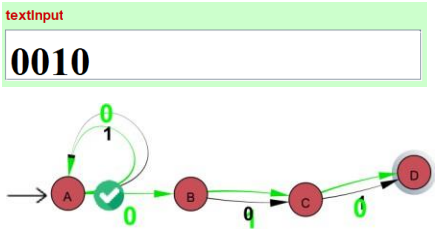


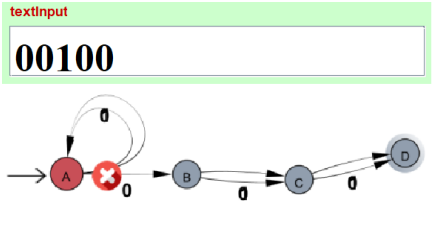


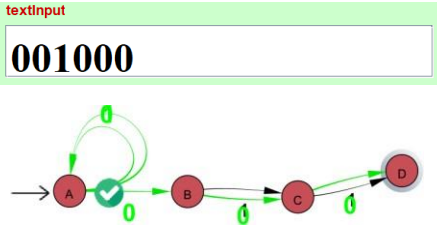
Ejemplo 3





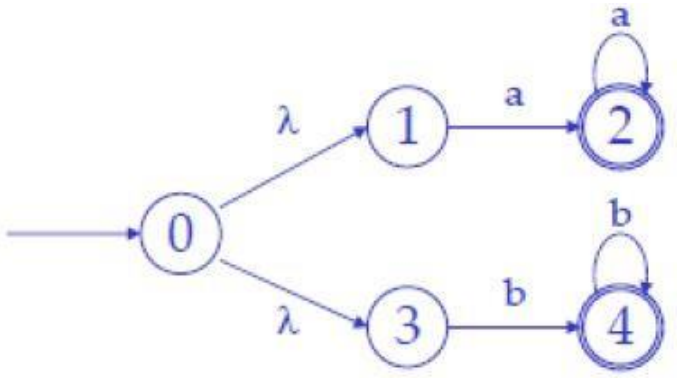


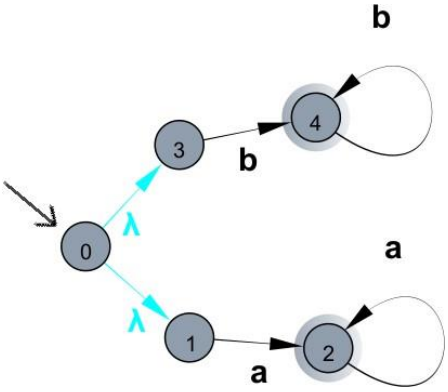




[8] El símbolo “X” se ve como un “+” debido a se ha girado la cámara para asemejarse al ejemplo (para girar la vista se podrán usar la flechas del teclado: “↑“ y “↓”).

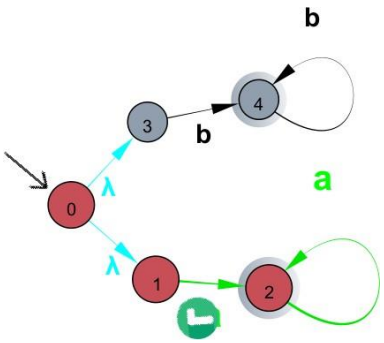
Ejemplo 4





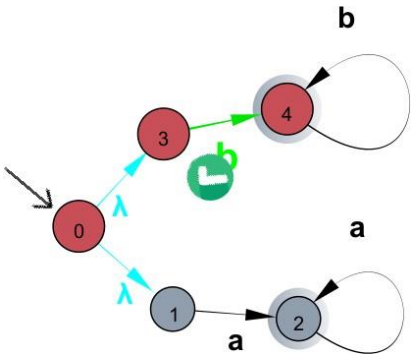
textInput

aaaaa



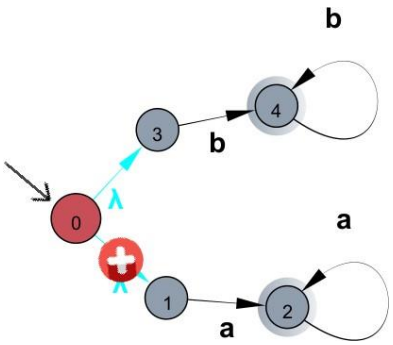
textInput

b

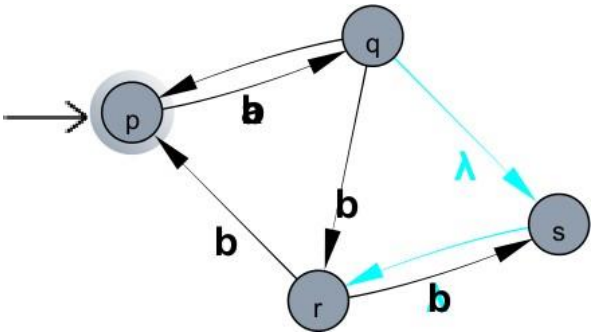


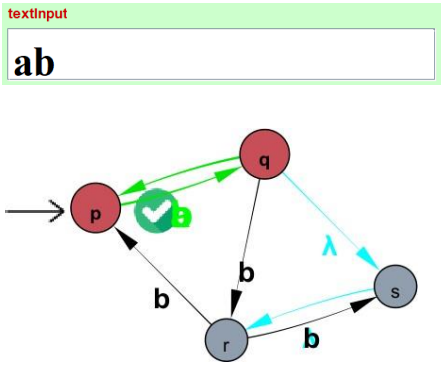
textInput

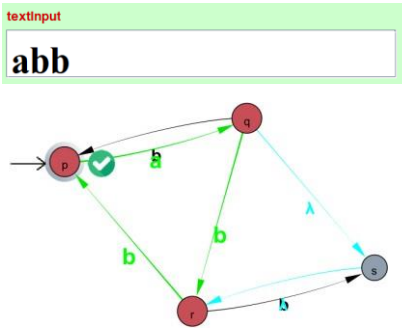
aba

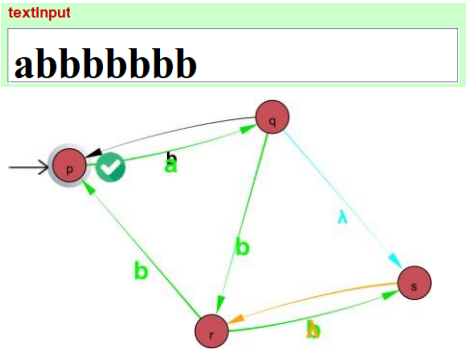


EjemploPropuesto1

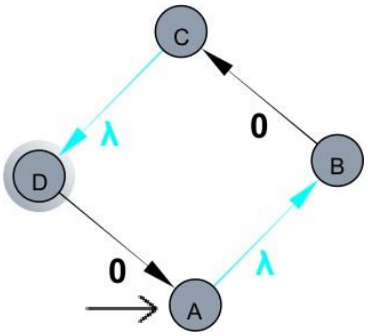






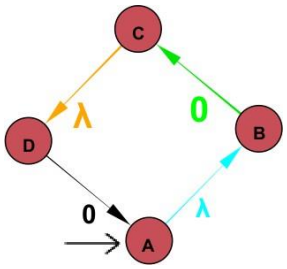


EjemploPropuesto2



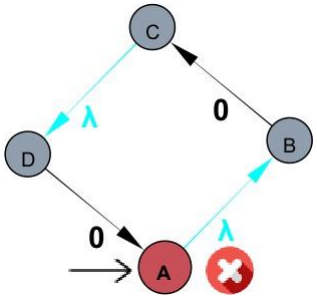
textInput

0



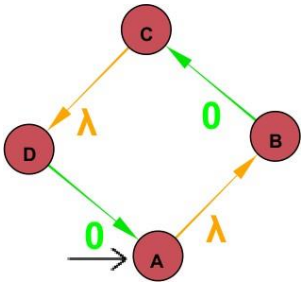
textInput

00

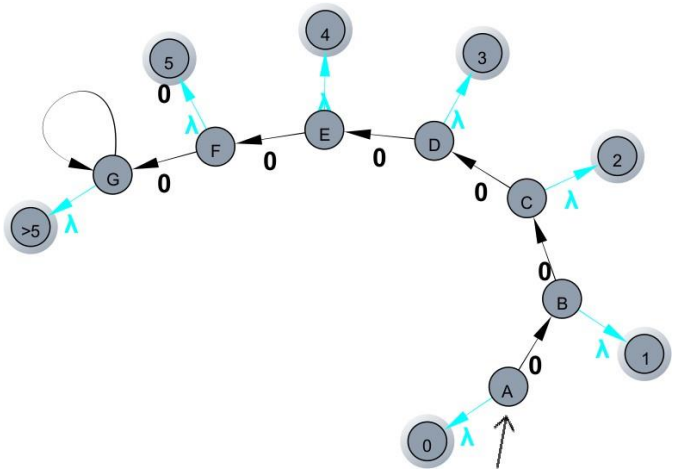


textInput

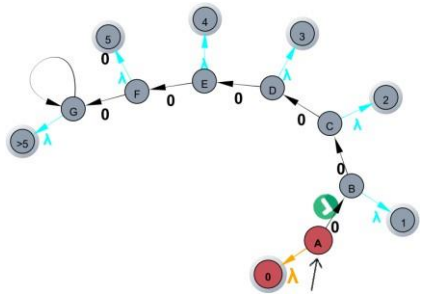
000



EjemploPropuesto3_contador[9]

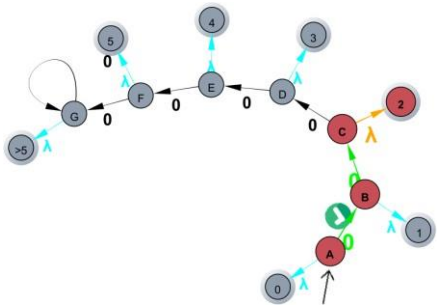


textInput



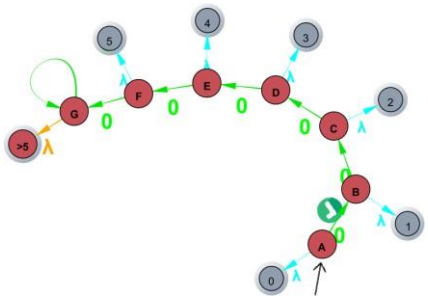
textInput

00



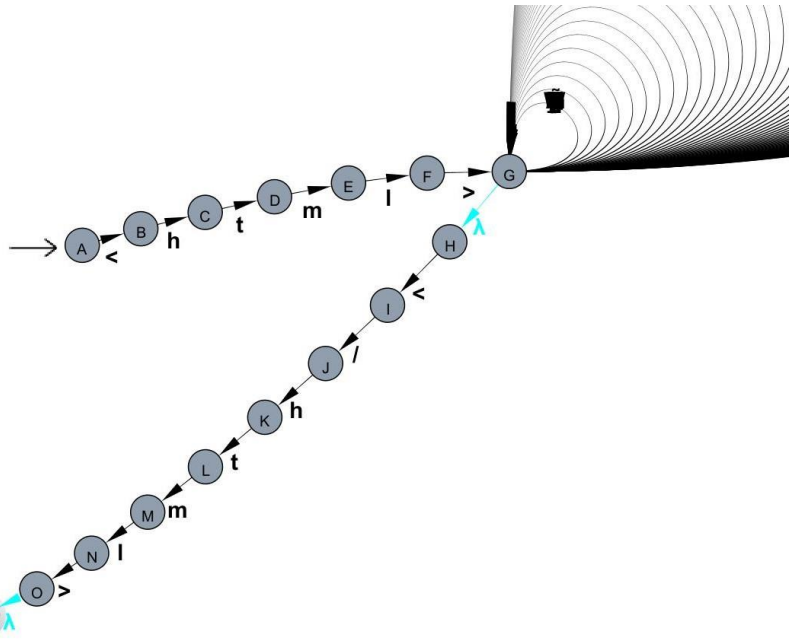
textInput

0000000000



[9] No cuenta la cantidad sino la veces que se repite de forma seguida, pero solo pudiendo introducir ceros.

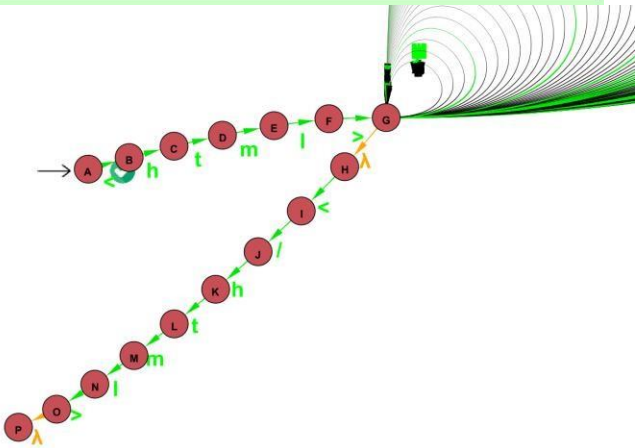
EjemploPropuesto4_regexpHTML



textInput

<html><body>Hola_Mundo</body></html>

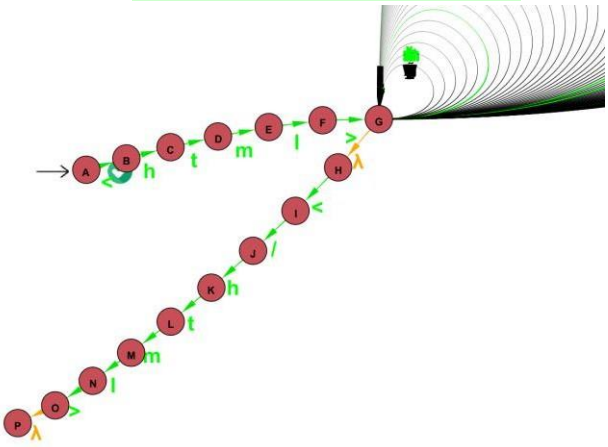
10



textInput

<html></htm</html>

11



[10] No es posible usar el espacio en blanco (‘ ’) como un símbolo para las transiciones, en cambio, se podría usar, como sustituta, la barra baja (‘_’). Esto se debe al uso (algo abusivo) de la funcion ‘String.trim()’ que borra los espacios y que se ha usado para “proteger” los símbolos de falsos negativos ser comparados.

[11] Como se puede apreciar, no se modela la totalidad de los casos; basta con que haya o no texto entre “<html>” y “</html>”.

ANEXO

Carpetas del Proyecto

Este proyecto incluye, además de la arquitectura MVC, dos carpetas: una con los autómatas en modo texto preparados para ser importados y otra que contiene iconos y el estilo que se emplea para dotar a 'GraphStream' de estilo.

Formato del Autómata[12]

```
TIPO: { AFND | AFD }
ESTADOS: estado1 [... estadoN]
INICIAL: inicial1 [ … inicialN] [13]
FINALES: final1 [… finalN]
TRANSICIONES:
{ estadoA 'símboloX' estadoB [... estadoC] }
...
TRANSICIONES LAMBDA:
{ estadoX estadoY [... estadoZ] }
...
FIN
```

Librerías Externas: GraphStream

Para la representación gráfica se ha empleado la librería 'GraphStream': https://graphstream-project.org/doc/

[12] Los corchetes indican opcionalidad, las llaves agrupan y los "|" sirven para agregar mas de una posible opción.

[13] Aunque no se ha probado, no deberían ocurrir errores en caso de agregar más de un estado inicial en cuyo caso el autómata habrá de ser de tipo AFND (Autómata Finito No Determinista)

Principales Problemas Encontrados

- Modelar todas las situaciones posibles del modo “paso a paso” ha sido extenuante: el modo “paso a paso” se ha tenido que rehacer varias veces (“ControladorAutómata.java”), dado que requirió mucha “prueba y error”.
- Ser capaz de mostrar correctamente el símbolo utilizado en cada transición (por “ControladorTextInput.java”) y que, finalmente se solucionó usando un diccionario.

Limitaciones Conocidas

- Es posible que tanto transiciones como imágenes (iconos) se vean entrecortados y no se muestren en su totalidad, esta es una de las limitaciones de la librería ‘GraphStream’. Para “solucionarlo” basta usar el botón de ‘zoomOut’.
- El ‘*ControladorTextInput.class’* puede presentar errores al tratar de ir hacia atrás; el resto de operaciones funciona correctamente.

Observaciones Finales

- También es posible rotar la escena con las flechas arriba y abajo (al pulsar el controlPanel Jframe).
- Ha sido la programación de los algoritmos, que permiten visualizar gráficamente al autómata, lo que más coste tanto de esfuerzo como de tiempo ha supuesto, no tanto por su complejidad, sino por la falta de experiencia para atajarlo: la parte básica la terminé en un fin de semana; el resto, dos o tres semanas.
- Aunque la documentación esté en inglés, no lo están la totalidad de los comentarios presentes en el código fuente.
- Se define una transacción (o transición) como el paso de un estado A (inicial) a otro estado B (final) mediante un comando (o símbolo) C (el cual se pone entre corchetes para una mejor diferenciación visual). De aquí se infieren múltiples comandos de error como <CX> o <BX> donde no existe transición para un comando determinado o para cuando el nodo actual no es final, respectivamente.