MANUAL TÉCNICO DE LA APLICACIÓN AUTÓMATAS FINITOS

ALEJANDRO CASTAÑOS ROJAS

ESTEFANY MURIEL CANO

UNIVERSIDAD DE ANTIOQUIA

2017

Página

2. Requisitos del sistema

Para lograr la ejecución adecuada de la aplicación de autómatas finitos, su equipo debe de tener sistema operativo Windows, además del IDE Netbeans, recomendablemente una versión reciente.

3. Descripción de métodos implementados

Paquete controlador:

En la clase ControladorAutomata tenemos los siguientes métodos:

Modifier and Type	Method and Description
void	`actualizarParticiones(java.util.ArrayList<java.lang.String> estadosAceptacion, java.util.ArrayList<java.lang.String> estados)` Este método actualiza las particiones de aceptación o de rechazo cuando el autómata es convertido a determinístico, dichas particiones son utilizadas en simplificar y en verificar la hilera.
java.util.ArrayList<java.util.ArrayList>	`convertirEnDeterministico()` Este método permite convertir un autómata no deterministico a deterministico, tomando el primer estado con sus transiciones del automata inicial, y a partir de este mediante la concatenación de estado y transiciones, contruir el autómata final.
int	`convertirEstados(java.lang.String a)` Este método permite retornar la posición del estado que se le entra por parámetro para saber su ubicación dentro del arreglo.
int	`convertirSimbolos(java.lang.String a)` Este método permite retornar la posición del simbolo que se le entra por parámetro para saber su ubicación dentro del arreglo.
java.lang.String[]	`convertirString(java.lang.String a)` Este método permite tomar un String y llevarlo a un arreglo con un caracter del String en cada posición.
boolean	`definirEstadoDeAceptacion(java.lang.String[] estados)` Permite decidir que estado de aceptación va a tener la unión de varios estados
boolean	`esDeterministico()` Permite verificar si el autómata ingresado es deterministico o no deterministico, mediante la revision de sus transiciones
boolean	`esEstadoDeAceptacion(java.lang.String estado)` Permite determinar si el estado ingresado es de aceptación.
void	`estadosAceptacion()` Aunque su nombre indique que selecciona los estados de aceptación, en realidad separa estados de aceptación de los estados de rechazo generando así 2 particiones; una partición P_0 con los estados de rechazo y una particion P_1 con los estados de aceptación.
boolean	`estaEnAceptacion(java.lang.String estado)` Este método determina si un estado es de aceptación o no
boolean	`existeEstado(java.util.ArrayList<java.lang.String> estados, java.lang.String estado)` Este método permite verificar si ya existe un estado, para evitar las repeticiones en el momento de agregar un nuevo estado.
boolean	`existeEstadoAceptacion(java.util.ArrayList<java.lang.String> estadosAceptacion, java.lang.String estado)` Este método analiza los estados de aceptación que se tienen en el momento y verifica que el estado que va a entrar no se encuentre repetido entre los estados de aceptación que ya están.
boolean	`existeSimbolo(java.lang.String simbolo)` Este método permite verificar si el simbolo es correcto, para evitar evaluar simbolos que no pertenecen al autómata.
java.util.ArrayList<java.util.ArrayList>	`guardarAutomata()` Este método trae de la tabla todas las transiciones y las guarda en un ArrayList de ArrayList con datos de tipo String
void	`imprimir(java.util.ArrayList<java.util.ArrayList> a)`
java.lang.String	`intercambiarEstados(java.util.ArrayList<java.lang.String> estados, java.lang.String estado)` Este método permite tomar un estado que se encuentra desordenado, y ordenarlo de la forma en que se define previamente, es decir, si existe un estado compuesto de más estados con un orden específico, los otros estados que contenga sus mismos estados, tambien van a estar con el mismo orden del estado

java.util.ArrayList<java.util.ArrayList>	**organizarAutomata**(java.util.ArrayList<java.lang.String> estados, java.util.ArrayList<java.util.ArrayList> automata) Toma el autómata que se construyó y reordena los estados de las transiciones,para que coincidan con los estados ya definidos, para evitar confusiones en el usuario,y sea más ordenado el proceso de reconocimiento.
javax.swing.table.DefaultTableModel	**ParticionesEnTabla**()
java.lang.String	**quitarEstadosRepetidos**(java.lang.String a, java.lang.String b) Este método compara dos estados y evita que hayan repeticiones de estados cuando se realice la unión de los mismos.
java.util.ArrayList<java.lang.String>	**revisarTransiciones**(java.util.ArrayList<java.lang.String> transiciones) Este método permite revisar las transiciones de un estado, y en caso de que un transición vaya a dos estados, los concatena.
javax.swing.table.DefaultTableModel	**Simplificar**() Este método usa las variables globales de particion para determinar si el automata en cuestión contiene estados equivalentes y reducirlo a su forma mínima.
java.util.ArrayList<java.lang.String>	**unirTransiciones**(java.lang.String[] estados) Este método permite hacer la unión de las transiciones de varios estados, tomando las transiciones de cada estado que se va unir y luego compara las transiciones que se van a unir para evitar la repetición de estados en las transiciones finales.
boolean	**verificarHilera**(java.lang.String hilera) Este método permite que al tener un autómata ya definido, podamos relaizar un reconocimiento de secuencias mediante el autómata, para saber si la secuencia es válida o no, nos ubicamos en el estado inicial, luego miramos que simbolo entra y avanzamos al estado que indique la transición, hasta llegar al final de la hilera que es denotada por el símbolo '*', cuando se llega al final se mira en que tipo de estado se finalizó y si es de aceptación se acepta la secuencia, de lo contrario se rechaza.

- **DESCRIPCIÓN MAS AMPLIA DE LOS MÉTODOS UTILIZADOS:**

- **ControladorAutomata**

- `public ControladorAutomata(AutomataF af, javax.swing.table.DefaultTableModel dtm)`

- ***Method Detail***

- **convertirSimbolos**

`public int convertirSimbolos(java.lang.String a)`

Este método permite retornar la posición del simbolo que se le entra por parámetro para saber su ubicación dentro del arreglo.

Parameters:

`a - String con el simbolo que se quiere buscar`

Returns:

`entero con la posición del símbolo del estado en el arreglo`

- **convertirEstados**

`public int convertirEstados(java.lang.String a)`

Este método permite retornar la posición del estado que se le entra por parámetro para saber su ubicación dentro del arreglo.

Parameters:

`a - String con el simbolo que se quiere buscar`

Returns:

entero con la posición del estado en el arreglo

- **guardarAutomata**

```
public java.util.ArrayList<java.util.ArrayList> guardarAutomata()
```

Este método trae de la tabla todas las transiciones y las guarda en un ArrayList de ArrayList con datos de tipo String

Returns:

un ArrayLsit con todas las transiciones del autómata

- **imprimir**

```
public void imprimir(java.util.ArrayList<java.util.ArrayList> a)
```

- **estadosAceptacion**

```
public void estadosAceptacion()
```

Aunque su nombre indique que selecciona los estados de aceptación, en realidad separa estados de aceptación de los estados de rechazo generando así 2 particiones; una partición P0 con los estados de rechazo y una particion P1 con los estados de aceptación.

- **esEstadoDeAceptacion**

```
public boolean esEstadoDeAceptacion(java.lang.String estado)
```

Permite determinar si el estado ingresado es de aceptación.

Parameters:

estado - String con el estado que se desea conocer si es de aceptación

Returns:

un booleano con la confirmación de si es un estado de acepetación o no.

- **esDeterministico**

```
public boolean esDeterministico()
```

Permite verificar si el autómata ingresado es deterministico o no deterministico, mediante la revision de sus transiciones

Returns:

un booleano con la confirmación de si es deterministico o no.

- **quitarEstadosRepetidos**

```
public java.lang.String quitarEstadosRepetidos(java.lang.String a,
                                               java.lang.String b)
```

Este método compara dos estados y evita que hayan repeticiones de estados cuando se realice la unión de los mismos.

Parameters:

`a - String con el primer estado a comparar`

`b - String con el segundo estado a comparar`

Returns:

`Un String con el nuevo estado sin repeticiones de estados en el mismo.`

- **unirTransiciones**

```
public java.util.ArrayList<java.lang.String> unirTransiciones(java.lang.String[] estados)
```

Este método permite hacer la unión de las transiciones de varios estados, tomando las transiciones de cada estado que se va unir y luego compara las transiciones que se van a unir para evitar la repetición de estados en las transiciones finales.

Parameters:

`estados - Arreglo de Strings con los estados a los cuales se les va a hacer la unión de transiciones.`

Returns:

`Un Array de Strings con las transciones previamente concatenadas y sin repeticiones en ellas.`

- **existeEstado**

- ```
  public boolean existeEstado(java.util.ArrayList<java.lang.String> estados,
                              java.lang.String estado)
  ```

Este método permite verificar si ya existe un estado, para evitar las repeticiones en el momento de agregar un nuevo estado.

Parameters:

`estados - Lista con los estados existentes hasta el momento.`

`estado - String con el estado a comparar.`

Returns:

`un booleano en true si el estado ya existe, o false de lo contrario.`

- **existeSimbolo**

```
public boolean existeSimbolo(java.lang.String simbolo)
```

Este método permite verificar si el simbolo es correcto, para evitar evaluar simbolos que no pertenecen al autómata.

Parameters:

`simbolo - String con el simbolo a verificar.`
```

Returns:

un booleano en true si el simbolo es correcto, o false de lo contrario.

- **definirEstadoDeAceptacion**

```
public boolean definirEstadoDeAceptacion(java.lang.String[] estados)
```

Permite decidir que estado de aceptación va a tener la unión de varios estados

Parameters:

estados - Arreglo con los estados a verificar su estado de aceptación

Returns:

un booleano en true si al menos uno de los estados es de aceptación, o false de lo contrario.

- **revisarTransiciones**

```
public java.util.ArrayList<java.lang.String> revisarTransiciones(java.
util.ArrayList<java.lang.String> transiciones)
```

Este método permite revisar las transiciones de un estado, y en caso de que un transición vaya a dos estados, los concatena.

Parameters:

transiciones - Array de String con las transciones a revisar.

Returns:

Un ArrayList de Strings con las transiciones concatenadas

- **existeEstadoAceptacion**

- ```
  public boolean existeEstadoAceptacion(java.util.ArrayList<java.lang.St
  ring> estadosAceptacion,
                                        java.lang.String estado)
  ```

Este método analiza los estados de aceptación que se tienen en el momento y verifica que el estado que va a entrar no se encuentre repetido entre los estados de aceptación que ya están.

Parameters:

estadosAceptacion - Lista que contiene los estados de aceptación que se tienen hasta el momento

estado - String con el estado a verificar.

Returns:

un booleano en true si el estado ya existe en la lista o false si no existe

- **actualizarParticiones**

- ```
  public void actualizarParticiones(java.util.ArrayList<java.lang.String
  > estadosAceptacion,
  ```

```
java.util.ArrayList<java.lang.String> estados)
```

Este método actualiza las particiones de aceptación o de rechazo cuando el autómata es convertido a determinístico, dichas particiones son utilizadas en simplificar y en verificar la hilera. Para actualizar se toman los estados que no pertenecen a la lista de estado de aceptación y se asigan en la partición de estados de rechazo.

Parameters:

`estadosAceptacion - Lista con los estados de aceptación del autómata.`

`estados - lista con los estados del autómata.`

- **convertirEnDeterministico**

```
public java.util.ArrayList<java.util.ArrayList> convertirEnDeterminist
ico()
```

Este método permite convertir un autómata no deterministico a deterministico, tomando el primer estado con sus transiciones del automata inicial, y a partir de este mediante la concatenación de estado y transiciones, contruir el autómata final.

Returns:

`un ArrayList de ArrayList de tipo String con el nuevo autómata.`

- **convertirString**

```
public java.lang.String[] convertirString(java.lang.String a)
```

Este método permite tomar un String y llevarlo a un arreglo con un caracter del String en cada posición.

Parameters:

`a - String que se quiere almacenar en un arreglo.`

Returns:

`Un arreglo con el String inicial.`

- **intercambiarEstados**

- ```
  public java.lang.String intercambiarEstados(java.util.ArrayList<java.l
  ang.String> estados,
                                              java.lang.String estado)
  ```

Este método permite tomar un estado que se encuentra desordenado, y ordenarlo de la forma en que se define previamente, es decir, si existe un estado compuesto de más estados con un orden específico, los otros estados que contenga sus mismos estados, tambien van a estar con el mismo orden del estado previamente definido.

Parameters:

`estados - Lista con los estados del autómata.`

`estado - String con el estado que se quiere reordenar.`

Returns:
  ```

String con el estado ordenado.

- **organizarAutomata**

- public java.util.ArrayList<java.util.ArrayList> organizarAutomata(java.util.ArrayList<java.lang.String> estados,

  java.util.ArrayList<java.util.ArrayList> automata)

  Toma el autómata que se construyó y reordena los estados de las transiciones,para que coincidan con los estados ya definidos, para evitar confusiones en el usuario,y sea más ordenado el proceso de reconocimiento.

  Parameters:

  estados - Lista con los estdos del autómata.

  automata - ArrayList con todas las transiciones del autómata.

  Returns:

  Un ArrayList con todas las transiciones ordenadas del autómata.

- **verificarHilera**

  public boolean verificarHilera(java.lang.String hilera)

  Este método permite que al tener un autómata ya definido, podamos relaizar un reconocimiento de secuencias mediante el autómata, para saber si la secuencia es válida o no, nos ubicamos en el estado inicial, luego miramos que simbolo entra y avanzamos al estado que indique la transición, hasta llegar al final de la hilera que es denotada por el símbolo '*', cuando se llega al final se mira en que tipo de estado se finalizó y si es de aceptación se acepta la secuencia, de lo contrario se rechaza.

  Parameters:

  hilera - String con la secuencia que se quiere verificar

  Returns:

  un booleano en true si la secuencia se acepta, o false de lo contrario.

- **Simplificar**

  public javax.swing.table.DefaultTableModel Simplificar()

  Este método usa las variables globales de particion para determinar si el automata en cuestión contiene estados equivalentes y reducirlo a su forma mínima.

  Returns:

  DefaultTableModel Un modelo de tabla para poder visualizarlo en pantalla

- **ParticionesEnTabla**

  public javax.swing.table.DefaultTableModel ParticionesEnTabla()

- **estaEnAceptacion**

```
public boolean estaEnAceptacion(java.lang.String estado)
```

Este método determina si un estado es de aceptación o no

Parameters:

`estado -`

Returns:

`boolean`

Paquete modelo:

En la clase AutomataF tenemos la siguiente implementación:

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
java.util.ArrayList<java.util.ArrayList>	getAutomataSinExtraños()
java.lang.String[]	getEstados()
java.lang.String[]	getEstadosAceptacion()
java.lang.String[]	getSimbolos()
java.util.ArrayList<java.util.ArrayList>	getTransiciones()
void	setAutomataSinExtraños(java.util.ArrayList<java.util.ArrayList> automataSinExtraños)
void	setEstados(java.lang.String[] estados)
void	setEstadosAceptacion(java.lang.String[] estadosAceptacion)
void	setSimbolos(java.lang.String[] simbolos)
void	setTransiciones(java.util.ArrayList<java.util.ArrayList> Transiciones)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

- ***Constructor Detail***

- **AutomataF**

```
public AutomataF()
```

- ***Method Detail***

- **getSimbolos**

```
public java.lang.String[] getSimbolos()
```

- **setSimbolos**

```
public void setSimbolos(java.lang.String[] simbolos)
```

- **getEstados**

```
public java.lang.String[] getEstados()
```

- **setEstados**

```
public void setEstados(java.lang.String[] estados)
```

- **getEstadosAceptacion**

```
public java.lang.String[] getEstadosAceptacion()
```

- **setEstadosAceptacion**

```
public void setEstadosAceptacion(java.lang.String[] estadosAceptacion)
```

- **getTransiciones**

```
public java.util.ArrayList<java.util.ArrayList> getTransiciones()
```

- **setTransiciones**

```
public void setTransiciones(java.util.ArrayList<java.util.ArrayList> T
ransiciones)
```

- **getAutomataSinExtraños**

```
public java.util.ArrayList<java.util.ArrayList> getAutomataSinExtraños
()
```

- **setAutomataSinExtraños**

```
public void setAutomataSinExtraños(java.util.ArrayList<java.util.Array
List> automataSinExtraños)
```

Paquete Vista:

En la clase PantallaIngreso tenemos la siguiente implementación de métodos:

definirEstadoAceptacion

```
public boolean definirEstadoAceptacion(java.lang.String estado)
```

Este método permite saber si un estado es de aceptación para llenar la tabla cuando se convierte de no determinístico a determinístico

Parameters:

`estado - String con el estado que se quiere verificar.`

Returns:

`un booleano en true si el estado es de aceptación, o false de lo contrario.`

llenarTabla

```
public void llenarTabla(javax.swing.JTable tabla)
```

Este método permite mostrar el autómata en un tabla para que el usuario pueda interactuar dinámicamente con ella.

Parameters:

`tabla -`

main

```
public static void main(java.lang.String[] args)
```

Parameters:

`args - the command line arguments`