Universidad Nacional Autónoma de México

Facultad de Ciencias

Programación Declarativa

Reporte: Regex a AFD

Profesor: Manuel Soto Romero
Ayudante: Juan Pablo Yamamoto Zazueta

Alumno:
Monterrubio Acosta Demian Alejandro
No. de cuenta: 317529180

9 de junio de 2023

Índice

1. Análisis del Problema

El objetivo de este proyecto es crear un programa que pueda leer expresiones regulares de entrada, y construya el autómata finito determinista mínimo que reconozca el lenguaje generado por la expresión regular. El AFD nos puede servir para analizar el lenguaje, o para implementarlo, en particular en el área de compiladores se ocupan AFD's para construir los analizadores Léxicos.
El problema se divide en tres partes principales:

1. Leer la cadena de entrada y procesarla obtener una representación útil en el programa. Esta parte es similar a las primeras fases de un intérprete para un lenguaje pequeño; con la diferencia de que no va a ejecutarse nada, solo genera una representación interna de la expresión regular. Mi objetivo en esta parte es obtener el ASA de la expresión, pues necesito tener una representación interna de la expresión regular que pueda manipular.

2. Construir el AFD mínimo a partir de la expresión leída; existen muchos métodos para construir los AFD's de una expresión regular, en particular me interesa un método que me permita generar el AFD mínimo.

3. A partir del AFD construido, generar una imagen de el y que se guarde en un archivo. Generar la imagen de un AFD significa generar la imagen de una gráfica casi arbitraria. Aunque existan muchas maneras de representar gráficas, algunas de las cuales son muy básicas, una buena representación del AFD es muy útil para poder analizarlo, o para poder agregarlo como parte de un documento. Las representaciones visuales son muy útiles principalmente para el entendimiento humano.

2. Antecedentes

En el campo de la programación, las expresiones regulares son herramientas poderosas para manipular y analizar patrones en cadenas de texto.
La manipulación de expresiones regulares ha sido un área de interés durante muchos años, y se han desarrollado numerosas implementaciones y bibliotecas en varios lenguajes de programación para trabajar con ellas. Las expresiones regulares se utilizan ampliamente en aplicaciones como el procesamiento de texto, en el área de compiladores, la búsqueda y el análisis de datos.
Los Autómatas finitos deterministas (AFD) se basan en la teoría de autómatas y proporcionan una forma formal de describir y reconocer lenguajes regulares. Un AFD está compuesto por un conjunto finito de estados, una entrada de alfabeto, una función de transición y un estado de aceptación.
La construcción de un AFD a partir de una expresión regular es un proceso fundamental en el estudio de las expresiones regulares y los autómatas. Existen algoritmos y técnicas bien establecidas para transformar una expresión regular en un AFD equivalente. Estos algoritmos, como el algoritmo de Thompson o el algoritmo de construcción de subconjuntos, permiten generar un autómata que puede reconocer el mismo lenguaje que la expresión regular original.

Dada una expresión regular, no siempre es buena idea obtener su AFD si nuestro único propósito es reconocer cadenas de entrada; un AFD puede tener hasta tamaño exponencial respecto a la expresión regular a la que es equivalente; sin embargo, si nuestro objetivo es analizar la expresión, y el lenguaje, sí es de mucha utilidad.
Una vez que se ha generado el AFD, visualizarlo de manera gráfica puede facilitar su comprensión y análisis. La representación gráfica del AFD permite identificar los estados, las transiciones y los estados de aceptación de forma intuitiva. Existen diversas herramientas y bibliotecas que permiten dibujar un AFD de manera automatizada, algunas incluso se encuentran en línea.
La representación de un AFD es una gráfica dirigida; existen muchos métodos para dibujar gráficas, pues son ampliamente estudiadas en matemáticas, sin embargo muchos de los métodos piden pre-condiciones que no se pueden garantizar en un AFD, (por ejemplo) que el grado de los vértices sea máximo de 4. En su contra parte existen métodos muy generales para dibujar todo tipo de gráficas, como los métodos ortogonales; estos métodos no son la manera más común de representar las gráficas, pero si partimos de un autómata arbitrario, resulta muy útil tener un método genérico.

3. Metodología

Me basé en la bibliografía para resolver cada parte del problema, aunque con algunas adaptaciones.

- Procesamiento de la entrada:

  Para este apartado tenía la intención de implementar los métodos vistos en la clase de compiladores; sin embargo, me encontré con dos obstáculos: me parecía complicado traducir los métodos a un lenguaje declarativo, pues los métodos que vimos en la clase de compiladores ocupan fuertemente el estado del programa; y los métodos de compiladores no creaban el ASA de la expresión. Por lo tanto opté por un acercamiento más similar a la construcción del intérprete que vimos en Lenguajes de programación.

  El código de procesamiento de la entrada consta de distintas etapas, cada etapa revisa que se cumplan ciertas propiedades en la entrada, y convierten la expresión de entrada en tipos intermedios con los que es más fácil operar; la última etapa construye el ASA de la expresión.

  El programa solo funciona con expresiones regulares válidas, una expresión regular válida está definida por la siguiente gramática:

```
S -> S | S
S -> SS
S -> S+
S -> S*
S -> S?
```

```
S -> (S)
S -> terminal
```

Donde un terminal puede ser un símbolo del alfabeto 'a', un rango de símbolos denotado por [...], la negación de un símbolo ∼a , cualquier símbolo (denotado por un punto .), o ϵ.

Por comodidad, el alfabeto se definió como el conjunto de todas las letras minúsculas $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$ y al guión $-$ como épsilon.

Dada una cadena de entrada, primero tiene que pasar por el lexer que convierte la cadena en una lista de Tokens, al mismo tiempo que revisa que todos los caracteres sean válidos. Después va a tener que pasar por el parser, pero antes de esto, tenemos un paso intermedio (en el código PreParse), el paso intermedio va a convertir la lista de Tokens en un tipo de dato llamado PreRegex; PreRegex nos permite tener sub-expresiones, con lo cual podemos tener distintos contextos anidados recursivamente; su principal objetivo es eliminar los paréntesis , creando una sub-expresión con el contenido de esos paréntesis. De esta forma se garantiza que se respete el orden de precedencia.

Por último se manda el PreRegex al parser, que ahora sí, lo convierte en un Regex; el parser solo tiene que hacer la conversión de tipos, pues la verificación sintáctica se realiza en preParse.

El tipo de dato Regex se basa en el presentado en el artículo [4], es simplemente una extensión de este que agrega más operadores, y que separa los terminales de la expresión para poder tener distintos tipos de terminales.

- Construcción del AFD mínimo:

  Para construir el AFD mínimo uso el método presentado en [1] (pags 74 - 83); que a su vez es muy similar a la forma que usan para evaluar pertenencia a una regex en [4].

  Este método consiste en tener una expresión regular con posibles marcas en cada uno de sus caracteres terminales; se empieza con la expresión regular sin marcar, y por cada símbolo del alfabeto se repite el siguiente proceso: intentamos avanzar las marcas leyendo símbolo, si una marca no puede avanzar, desaparece; pero si puede avanzar, entonces avanza hasta el siguiente terminal al cual puede llegar leyendo ese símbolo.

  La expresión regular marcada de una manera concreta va a representar un estado q, y si leyendo un símbolo c se puede llegar a otra configuración de las marcas p, significa que el estado q tiene una arista que leyendo c llega al estado p.

  Usando este método, guardando todos los estados en un diccionario, y todas las aristas en una lista, podemos obtener la descripción completa del AFD.

Después de aplicar el método, revisamos cuales son los estados finales (los que tengan una marca en algún terminal que pueda ser el final de la regex), y se devuelve una lista de tuplas con el índice del estado y un valor booleano que indica si es final.

- Dibujo del AFD:

  Para dibujar el autómata usé el método de tres fases para graficación ortogonal de una gráfica [2]. El método en cuestión es el siguiente:

  1. Primero se necesitan cumplir las siguientes pre-condiciones: se separan los componentes conexos en distintas gráficas, se eliminan las aristas reflexivas.
  2. Este método trabaja sobre un sistema de coordenadas cuadrado, como si fuera una rejilla.
  3. Posicionamiento de nodos: se le asigna una coordenada a cada nodo, (es decir un cuadro de la rejilla), de manera que no haya dos nodos en la misma coordeada; cualquier posicionamiento que cumpla esta propiedad es válido.
  4. Enrutamiento de aristas: se crean las aristas que conecten los nodos, las aristas no van a tener curvaturas, van a ser una línea quebrada con una única vuelta. En este paso no importa si las aristas pasan sobre otros vértices.
  5. Asignación de puertos: se aumentan las dimensiones de los nodos, de manera que sea de $(n \times m)$, donde n es en número de aristas incidentes, y m el número de aristas que salen del nodo; a cada arista se le asigna un puerto, es decir una entrada, de manera que no haya más de dos en la misma entrada.

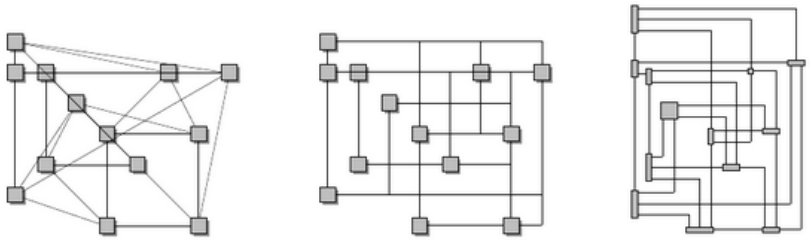

Figure 4: A graph shown after each of the three main phases.

Figura 1: [2] pag. 6

El artículo también presenta métodos específicos para llevar a cabo cada uno de los pasos, algunos de estos garantizan ciertas propiedades, como que minimizar la longitud de las aristas; pero, estos métodos eran bastante complejos, por lo que no las usé.

Todos los nodos siguen el mismo estándar, las aristas de entrada tienen que entrar por arriba o por abajo, y las de salida tienen que salir por los costados; no hay flechas que indiquen la dirección de las aristas. Todos los vértices finales están coloreados.

El método permite que haya más de un nodo en la misma columna o mismo renglón, pero decidí que cada nodo estaría en un renglón y columna separados; de esta forma, no se empalmaría ningún nodo, y además, no habría ninguna arista recta, todas estarían quebradas, por lo que podría manipular a todas las aristas de la misma forma.

Todos los vértices estaban etiquetados con un número que corresponde al estado que representan, ese número será el número de columna donde los posicionó. Por esta razón el estado inicial siempre es el nodo de más a la izquierda. En una parte del código se meten todos los nodos en un diccionario, el diccionario tiene la característica de que no garantiza el orden de sus elementos, decidí que para elegir el renglón de cada vértice tomaría el orden en el que los acomoda el diccionario; así no tengo que recurrir a bibliotecas externas para aleatorizar el proceso. Por último decidí que los vértices solo podrían estar en columnas y renglones pares, así se espaciarían, y aunque haya vértices cercanos habría espacio suficiente entre ellos; además, así las aristas solo van a pasar por renglones y columnas impares, de esta forma nunca chocarán con un vértice intermedio.

El paso de enrutar aristas está implícito en el código, pues las aristas tienen las coordenadas de sus dos extremos, pero no las dibujo hasta que se asignen los puertos. Para asignar los puertos, simplemente ordeno las aristas, y les asigno su puerto por orden de aparición; llevando un contador de desplazamiento, que es el valor que se van a tener que desplazar todas las aristas y vértices después de la actual, (recordemos que para agregar puertos es necesario crear nuevas filas y columnas).

Hay que señalar que, aunque una pre-condición es quitar los lazos, el programa quita los lazos hasta después de asignar puertos, pues se deben contar los lazos en el conteo de puertos y desplazamientos. Luego de asignar puertos separo a los lazos de las otras aristas; a cada lazos los divido en dos aristas, la unión de ambas aristas dibujan el lazo completo, pero al separarlos puedo se convierten en una arista quebrada con un único doblez, y puedo usar las mismas funciones para dibujar a todas las aristas.

Después de llevar a cabo este proceso se dibujan las aristas con sus etiquetas, y los vértices con sus id's. Por último se guarda en un archivo.

4. Resultados

Respecto al reconocimiento de expresiones regulares:
Creo que este es un punto débil del proyecto, aunque es funcional, me parece que hubiera sido mejor implementar métodos más elegantes del área de compiladores; principalmente porque escalar la gramática definida es complica; aunque la manera que está implementada no es la

más rápida, para los propósitos de su uso, funciona a la perfección.

Respecto a la construcción del AFD:
Me parece que en este apartado brillaron las virtudes que provee haskell, al poder definir nuestros propios tipos, y al ser un lenguaje funcional, construir el AFD fue muy sencillo. Funciona bien en tiempo, y el código es simple y legible. Le doy el crédito a los autores de [4] porque ellos escribieron el código en el que me basé. Y al creador del método para transformar regex en afd, pues es muy bueno.
En este apartado, el tiempo si se puede ver perjudicado, pues usé diccionarios para guardar los estados; en Haskell los diccionarios no tienen accesos en tiempo constante, entonces, tiene peor complejidad que en otros lenguajes impuros; pero, aún así, el acceso a diccionarios sigue siendo muy bueno, pues es logarítmico, y no afecta la complejidad, porque de todas formas se ve opacado por el tiempo que toma el método de construcción del afd, entonces, en la práctica esto no importa.

Respecto al dibujar el AFD:
Los resultados son buenos, el programa funciona conforme a lo esperado y se imprime el AFD; de manera subjetiva puedo decir que el método para generar gráficas ortogonales es bastante bueno en cuanto a generar gráficas legibles. A continuación se muestran tres ejemplos de gráficas generadas con el programa:

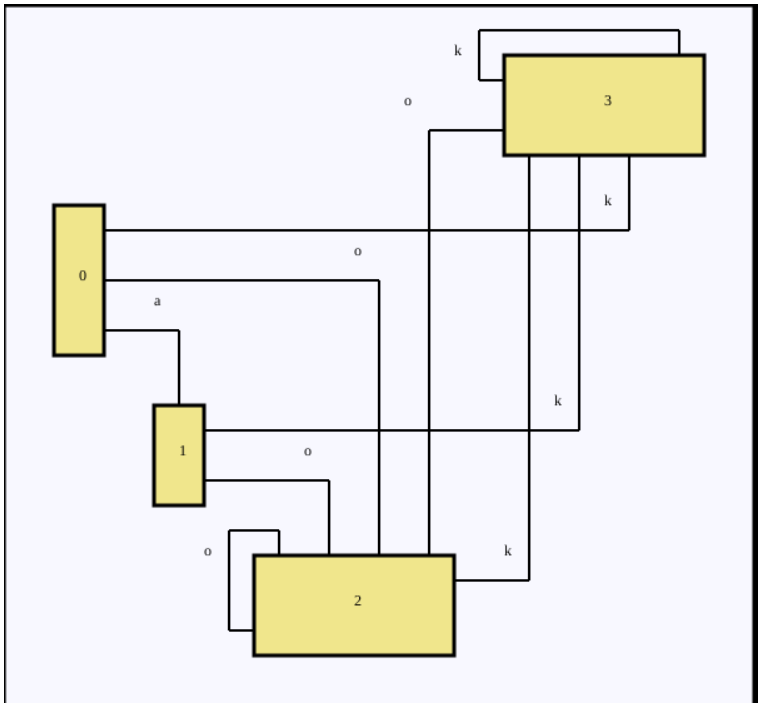

Figura 2: Gráfica de la regex: a?(o—k)*

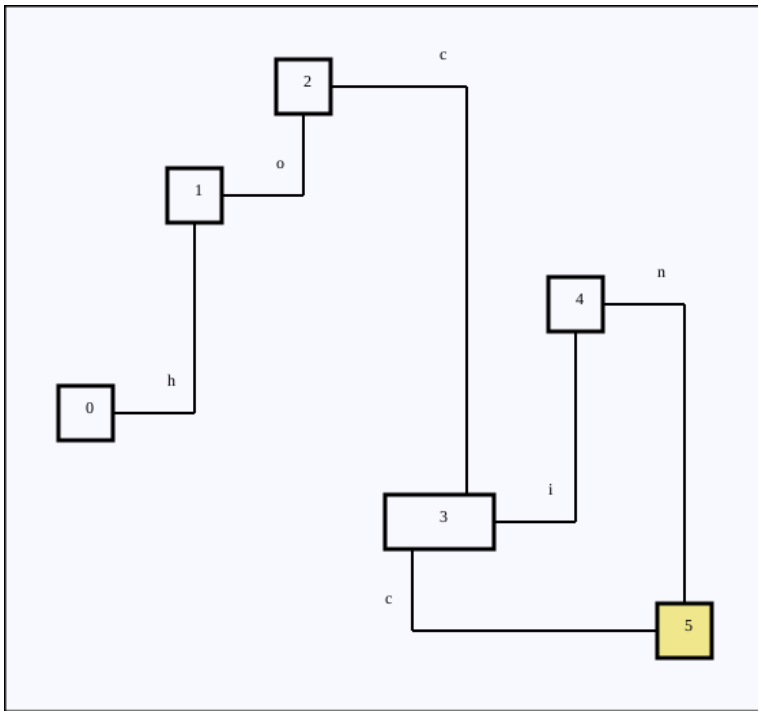

Figura 3: Gráfica de la regex: ho-(cin)+

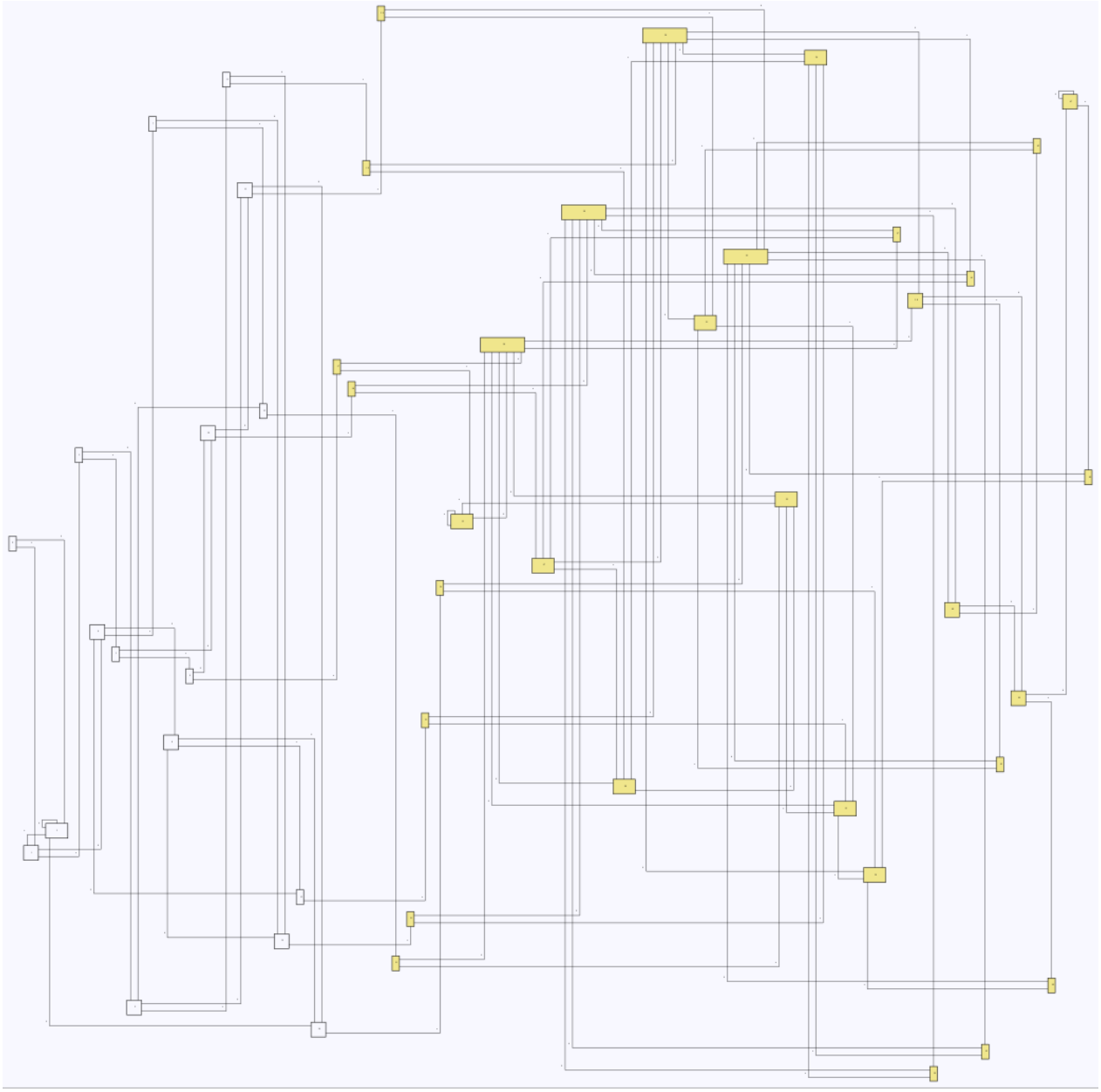

Figura 4: Gráfica de la regex: (a|b)*a(a|b)(a|b)(a|b)a(a|b)*

En método de tres fases para graficación ortogonal funciona para cualquier gráfica arbitraria, no fue diseñado específicamente para autómatas; pero, a partir de las imágenes generadas, me parece que el método funciona muy bien, y es capaz de generar autómatas muy agradables a la vista.
Debido a que la asignación de columnas fue en orden de su id, y que el id se asignó en orden de aparición del estado; de manera que el dibujo del autómata, tiende a ir de izquierda a derecha en sus transiciones, lo que resulta bastante intuitivo. Por ejemplo para expresiones regulares como:

```
lineas+
```

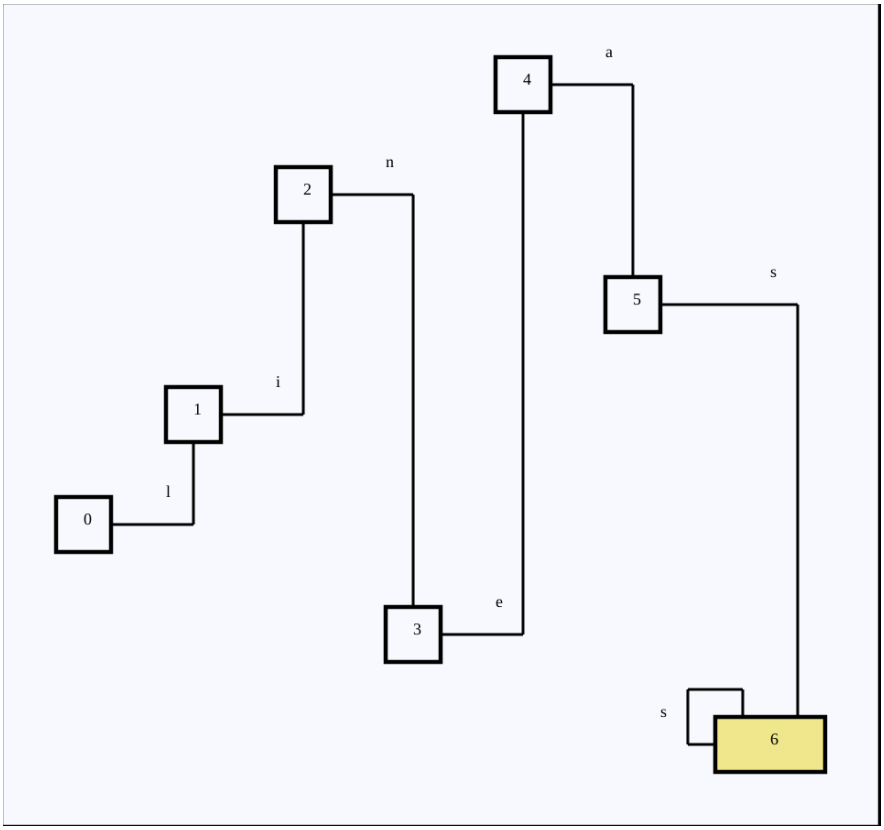

Figura 5: Gráfica de la regex: lineas+

Si bien el problema con este tipo de expresiones es que las aristas están quebradas y no son líneas rectas, aún así el dibujo es bastante entendible. Y al quebrar todas las aristas los dibujos resultantes tienen a tomar espacios más cuadrados, en vez de ser dibujos rectangulares con mucha anchura.

Se estén ocupando más renglones y columnas extra para espaciar los vértices, y ningún estado repite renglón o columna con otro, por lo que se está desperdiciando espacio, sin embargo, no perjudica al entendimiento del dibujo, y en gráficas con muchos estados ayuda mucho tener un buen espaciamiento entre ellos.
Claro que esta valoración es meramente subjetiva, de manera objetiva solo puedo decir que el método funciona. Me parece que es un muy buen método para graficar por su simpleza, y porque tiene muchas formas diferentes de implementarse, que pueden volverlo más elegante.

5. Propuestas a Futuro

Existen muchos detalles del proyecto que se pueden mejorar, o agregar; los que me parecen más notables son:

- Una interfaz gráfica para todo el programa, incluso se podría escalar a una página web.
- Expandir la gramática para que pueda trabajar con todos los operadores de expresiones regulares.
- Permitir que el usuario defina su propio alfabeto, y su propia épsilon.
- Permitir la negación de expresiones regulares, no solo de caracteres, (esto probablemente implicaría una modificación al método).
- Agregar flechas al dibujo para que sea más claro el camino de las aristas.
- Cambiar el método de distribución de los vértices a uno que no los separe tanto, y que permita líneas rectas.
- Internamente el programa genera todo el texto y luego lo escribe en el archivo, esto es mucho más ineficiente que escribirlo conforme es generado.
- Manejo de errores más elaborado, con mensajes más claros.
- Hay varias optimizaciones que se pueden meter al código, y modificaciones para que sea más legible.

6. Bibliografía:

1. Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. 2012. Modern Compiler Design (2nd. ed.). Springer Publishing Company, Incorporated.
   Link de consulta.

2. Biedl, Therese & Madden, Brendan & Tollis, Ioannis. (2000). The Three-Phase Method: A Unified Approach to Orthogonal Graph Drawing.. Int. J. Comput. Geometry Appl.. 10. 553-580. 10.1142/S0218195900000310.
   Link de consulta.

3. Eiglsperger, Markus & Fekete, Sandor & Klau, Gunnar. (1999). Orthogonal Graph Drawing. Drawing Graphs: Methods and Models, 121-171 (2001). 2025. 10.1007/3-540-44969-8_6.
   Link de consulta.

4. Fischer, S., Huch, F., & Wilke, T. (2010). A play on regular expressions: functional pearl. ACM SIGPLAN International Conference on Functional Programming.
   Link de consulta