DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico: tlengrep

12 de octubre de 2023 — Teoría de lenguajes

Grupo: BTS

Estudiante	LU	Correo electrónico
Luciana Skakovsky	131/21	lucianaskako@gmail.com
Florencia Rosenzuaig	118/21	f.rosenzuaig@gmail.com
Alan Roy Yacar	174/21	alanroyyacar@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

El propósito de este proyecto consiste en desarrollar una herramienta capaz de detectar las ocurrencias de una expresión regular en un texto. En esta fase inicial, nos enfocamos en la creación de un motor que representará las expresiones regulares como objetos y, además, implementará la funcionalidad de búsqueda haciendo uso de autómatas finitos.

2. Solución

La solución ideada para este problema se divide en 3 partes principales:

- **Implementación de la función `to_afnd()` para cada subclase de RegEx:** En la primera fase de nuestro enfoque, nos centramos en la creación de la función `to_afnd()` para cada subclase de expresiones regulares (RegEx). Esto implica la conversión de las expresiones regulares en autómatas finitos no deterministas (AFND). Cada subclase de RegEx tiene su propio proceso de transformación en un AFND, lo que nos permite abordar una amplia gama de expresiones regulares de manera efectiva. Cada AFND lo creamos de la siguiente manera:

  - **Regex $\emptyset$:**
    Creamos un AFND con un único estado inicial que no es final
  - **Regex λ:**
    Creamos un AFND con un único estado que es inicial y final
  - **Regex Carácter α:**
    Para la regex que denota el lenguaje determinado por un único carácter α creamos un AFND con un estado q_0 inicial y un q_1 final unidos por una única transición por el carácter
  - **Regex Concatenación:**
    Para la regex que denota el lenguaje determinado por la concatenación de dos lenguajes $L_3 = L_1.L_2$ lo que hacemos es crear los AFNDs de L_1 y L_2 y creamos un AFND nuevo para L_3. Luego agregamos todos los estados y transiciones de L_1 y L_2 a L_3 solo marcando como finales los estados finales de L_2, tras hacer eso, conectamos los estados finales de L_1 con el estado inicial de L_2 con una transición λ y hacemos que el estado inicial de L_1 sea el estado inicial de L_3. Para finalizar mergeamos los alfabetos de L_1 y $_2$ para obtener el de L_3.
  - **Regex Unión:**
    Para la regex que denota el lenguaje determinado por la Unión de dos lenguajes $L_3 = L_1|L_2$ lo que hacemos es crear los AFNDs de L_1 y L_2 y creamos un AFND nuevo para L_3. Luego agregamos todos los estados y transiciones de L_1 y L_2 a L_3 manteniendo los estados finales de L_1 y L_2 como estados finales de L_3, tras hacer eso agregamos un nuevo estado inicial a L_3 y conectamos ese estado a los estados iniciales de L_1 y L_2 con una transición λ a cada estado.
  - **Regex L^*:**
    Para la regex que denota la clausura de Kleene para otra regex lo que hacemos es pasar L a AFND y luego agregamos un nuevo estado inicial q_i y un nuevo estado final q_f. Conectamos q_i al estado inicial original de L y a q_f con transiciones λ y unimos también todos los estados finales originales de L a q_i y q_f con transiciones λ
  - **Regex L^+:**
    Para la regex que denota la clausura positiva de otra regex, usamos la concatenación y la expresión estrella definidas previamente y hacemos $L^+ = L.L^*$

- **Desarrollo de la función de Determinación:** En la siguiente etapa, implementamos una función de determinación, siguiendo el enfoque enseñado en clase. Esta función tiene como objetivo transformar los autómatas finitos no deterministas (AFND) en autómatas finitos deterministas (AFD). La determinación es un paso crucial para lograr una búsqueda eficiente de las ocurrencias de la expresión regular en el texto de entrada.
- **Creación de la función de Minimización:** En la tercera parte de nuestra solución, diseñamos una función de minimización. Esta función se encarga de reducir la complejidad de los autómatas finitos deterministas (AFD) resultantes, optimizando así el rendimiento y la eficiencia de nuestras búsquedas. Seguimos el algoritmo presentado en las clases prácticas para garantizar que nuestros autómatas tengan la menor cantidad posible de estados.

Además, otra parte esencial de nuestra solución involucra la función `match`. Esta función descompone el proceso de verificación de si una palabra de entrada cumple con la estructura definida por la expresión regular. Para lograr esto, adoptamos un enfoque eficiente: durante la primera lectura de la expresión regular, construimos el autómata correspondiente. Luego, reutilizamos este autómata para recorrer los estados y verificar si se alcanza un estado final con la palabra siguiente. Esta estrategia nos permite ahorrar tiempo y realizar cálculos costosos de manera más efectiva, evitando repeticiones innecesarias y optimizando el proceso de coincidencia.

3. Experimentación

En esta sección, llevaremos a cabo un análisis exhaustivo y una comparación de la eficiencia de la implementación que hemos desarrollado en comparación con la versión naive proporcionada por la cátedra. Para este propósito, hemos diseñado varios casos de experimentación.

Por un lado, examinaremos el rendimiento de ambas implementaciones al variar el tamaño de la cadena de entrada. Además, también investigaremos cómo se comportan al variar la cantidad de cadenas de entrada.

Para medir la eficiencia lo haremos en términos de tiempo de ejecución utilizando la librería de `time` de python. Utilizamos la expresión regular dada por la cátedra en el archivo `r29`, que denota la expresión regular $(a|b|c) + d|e|f$, y generamos distintos archivos con las cadenas de texto correspondientes.

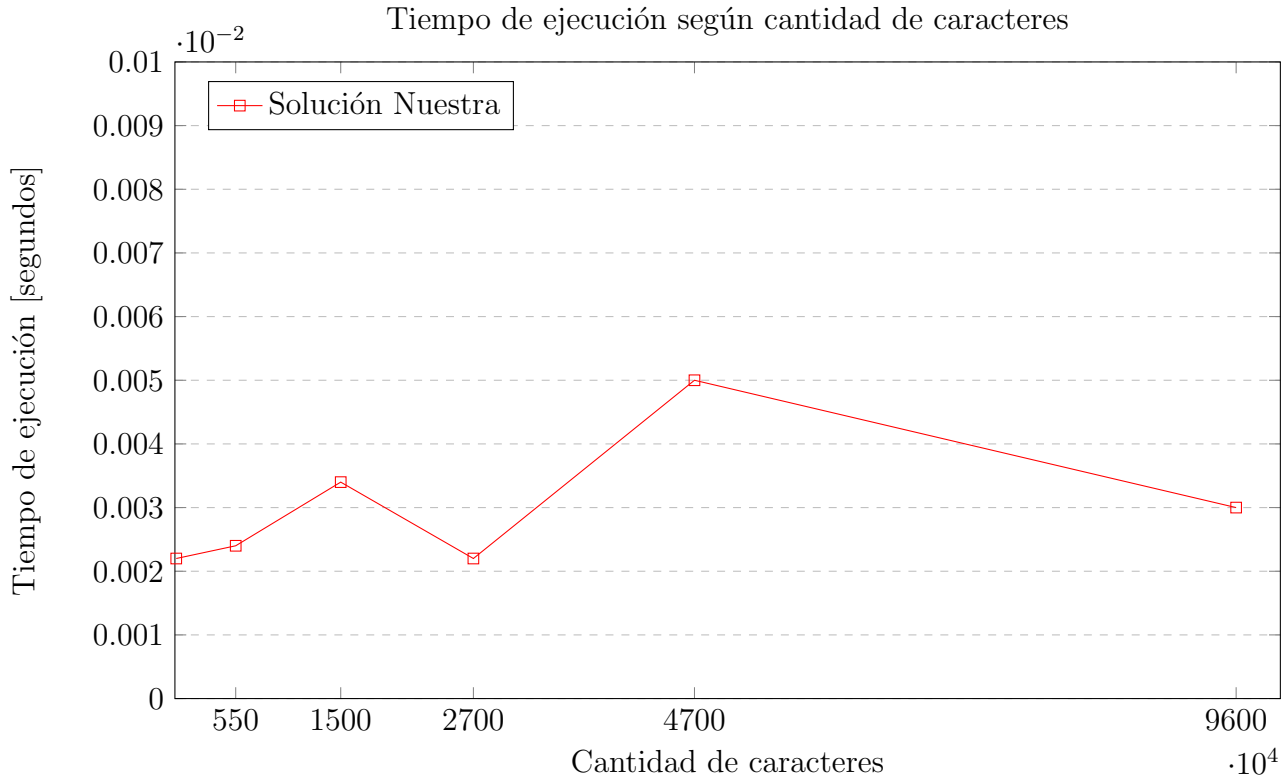
3.1. Experimento 1: Variación del tamaño de la cadena

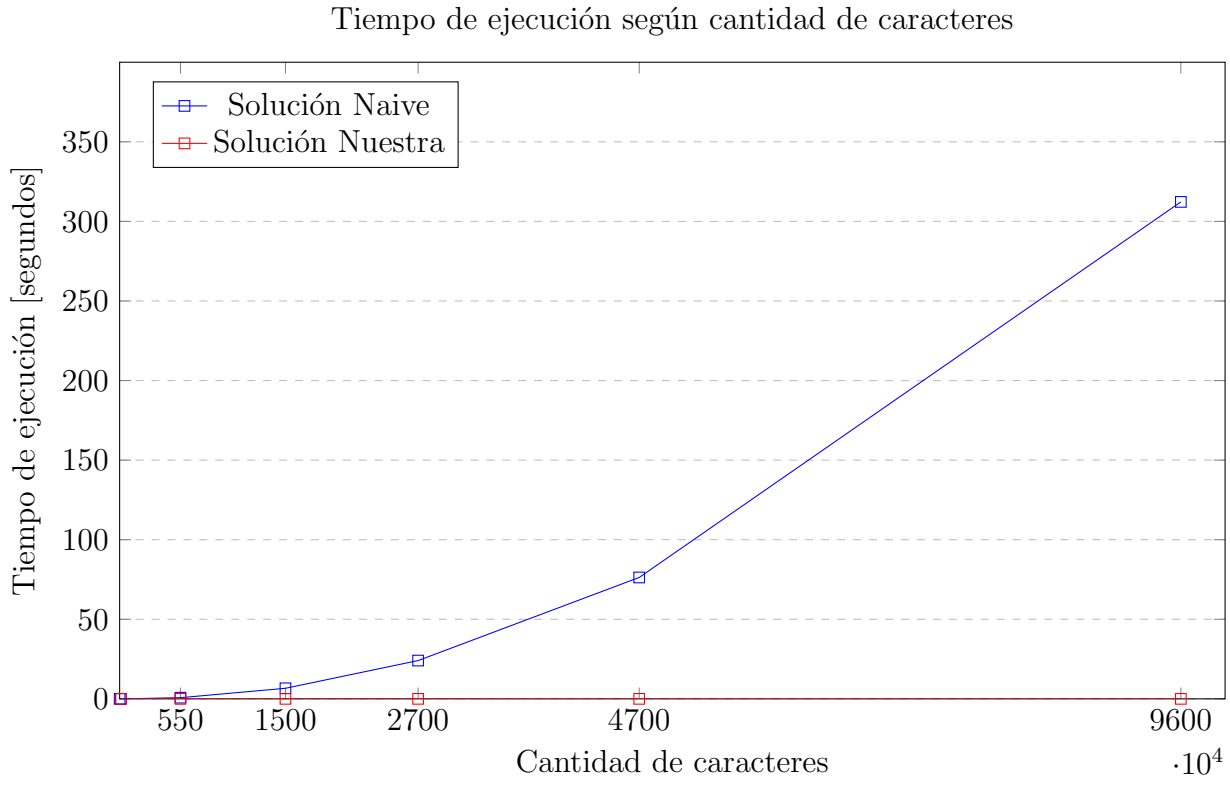
Como mencionamos previamente, en esta sección, nos centraremos en la comparación del tiempo de ejecución de ambas implementaciones a medida que variamos el tamaño de la cadena de entrada. Para llevar a cabo este experimento, hemos creado seis archivos diferentes. Cada uno lleva el nombre de `abc_n.txt` donde cada uno contiene una cadena que consiste en la repetición de 'abc' con un total de n caracteres.

Con el propósito de obtener resultados confiables y representativos, ejecutamos ambos programas 10 veces con cada uno de los archivos. Esta estrategia nos permitirá analizar el rendimiento de manera más sólida, reduciendo la influencia de variaciones aleatorias en los tiempos de ejecución y proporcionando un promedio de estos tiempos.

En los siguientes gráficos, se representa el comportamiento de ambas implementaciones en relación al tamaño de la cadena de entrada. Como se observa claramente, la implementación naive exhibe una significativa lentitud y un crecimiento exponencial del tiempo, en contraste con nuestra implementación que corre en milésimas de segundo. En los casos analizados, nuestros resultados indican un tiempo de procesamiento notablemente reducido.

Un ejemplo ilustrativo de esta diferencia se encuentra en la entrada de 9,600 caracteres. La implementación naive demanda aproximadamente 5 minutos para completar su ejecución, mientras que nuestra implementación lo hace en menos de 3 milésimas de segundo. Esta disparidad en el rendimiento resalta la eficiencia de nuestra solución y su capacidad para manejar cadenas de entrada significativamente más grandes de manera rápida y efectiva.





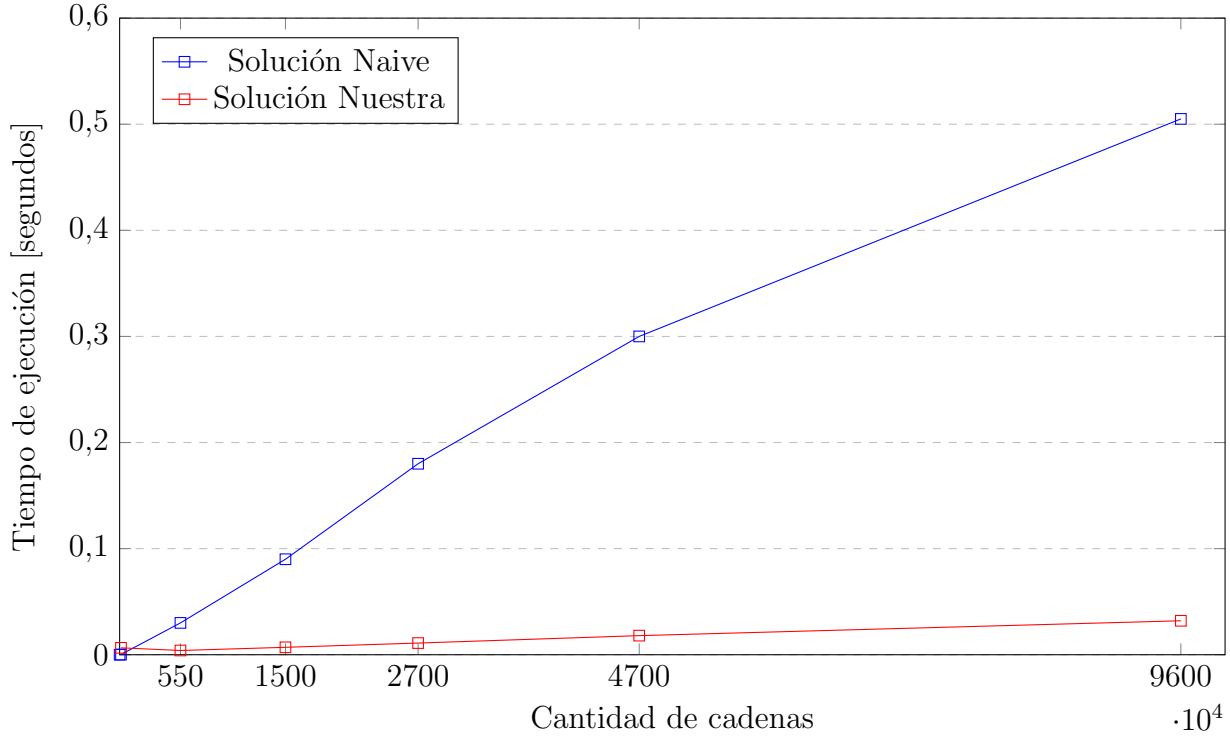
Cantidad de caracteres	12	550	1500	2700	4700	9600
Solución Naive	0.0001	0.74	6.64	24.06	76.27	312.24
Solución Nuestra	0.0022	0.0024	0.0034	0.0022	0.005	0.003

3.2. Experimento 2: Variación de la cantidad de cadenas

Este segundo experimento consiste en variar la cantidad de cadenas de entrada a detectar. Nuevamente creamos distintos archivos con diferentes cantidades de cadenas. Estos archivos llevan el nombre `wordsN.txt` donde N representa la cantidad de cadenas a detectar.

Nuevamente, puede observarse que la solución naive es más lenta que la nuestra. En éste caso, el crecimiento de la solución Naive se aproxima a la función lineal. Nuestra solución también tiene crecimiento lineal, pero crece más lentamente.

Tiempo de ejecución según cantidad de cadenas



Cantidad de caracteres	12	550	1500	2700	4700	9600
Solución Naive	0.0002	0.03	0.09	0.18	0.3	0.505
Solución Nuestra	0.0065	0.004	0.007	0.011	0.018	0.032