Algoritmos Avanzados: Crear contraseñas de 8 caracteres mediante *Backtracking*

FELIPE IGNACIO MATURANA GUERRA[1]

[1] *Universidad de Santiago de Chile, Santiago, 2017.*
[*] *Correspondencia del autor: felipe.maturana.g@usach.cl*

Compilado April 16, 2017

Existe un sistema de seguridad el cual posee implementado la generación de contraseñas de acuerdo a combinaciones de letras y números de un largo de 8 caracteres en total, sin embargo, existen ciertos criterios que deben cumplir las contraseñas. La primera condición consiste en que dentro de una combinación no pueden existir 3 letras o 3 números de forma consecutiva, en segundo lugar si la contraseña comienza con un número, ésta no puede terminar con un número, por último, la contraseña no puede iniciar con una vocal. En esta experiencia a medida que se van realizando las combinaciones, éstas son sometidas a un análisis de criterios para verificar si cumple con las condiciones anteriormente expuestas, de ésta forma si la combinación es inválida se detiene la iteración hasta ese punto y se continúa con la siguiente potencial combinación, esta técnica es conocida como *Backtracking*. De esta forma se mejora un problema presente en la experiencia anterior cuando se utilizó *Fuerza Bruta* donde se generaban TODAS las combinaciones y luego se les aplicaba el filtro, lo cual se traducía en un alto costo computacional y, por lo tanto, en lo que respecta a orden computacional. Los caracteres a utilizar son el alfabeto (en minúsculas), excluyendo la ñ, y los números del 0 al 9. El código fuente está implementado en Lenguaje C desarrollado en ambiente Linux, respetando el estándar ANSI C.

URL Github: El código fuente de la publicación y los códigos del programa se encuentran alojados en Git-Hub.

https://github.com/Felipez-Maturana/AlgoritmosAvanzados_L2_Backtracking

1. INTRODUCCIÓN

En el mundo de las contraseñas, los niveles de seguridad cobran cada vez más importancia ya que nuestras distintas credenciales en las páginas web, redes sociales y nubes virtuales poseen información sensible de nuestras vidas privadas las cuales en las manos equivocadas pueden destruir la vida de una persona. Es por esto, que es necesario aumentar los niveles de seguridad de nuestras contraseñas, incluyendo una mayor cantidad de caracteres distintos o simplemente haciendo las contraseñas más extensas. En la criptografía y el mundo del hacking existen distintos métodos de generar y descifrar contraseñas mediante distintos tipos de ataques, uno de ellos es el **Método Backtracking** o **búsqueda con retroceso**, una estrategia utilizada para buscar soluciones que satisfacen restricciones (como nuestro caso). Se dice que es un algoritmo de búsqueda en profundidad y si en ésta búsqueda encuentra una combinación que no satisface las restricciones, el algoritmo **retrocede** y toma la siguiente opción. Generalmente los algoritmos implementados son de carácter recursivo, sin embargo, también se pueden encontrar de forma no recursiva. Dado este algoritmo no se crearán todas las combinaciones posibles, sino, sólo las combinaciones válidas, dada una lista de ***x caracteres*** y de un ***largo*** $n = 8$, originando a x^n combinaciones totales. Es por esto, que contraseñas que sólo utilicen dígitos numéricos ($x = 10$) son más fáciles de descifrar que otras que incluyan el alfabeto ($x = 25$). A lo largo de la experiencia se buscará comprobar si la técnica de **Backtracking** presenta mejoras a la anterior experiencia donde se utilizó **Fuerza Bruta**.

2. DESCRIPCIÓN DEL PROBLEMA

La compañía Cuídate S.A crea sistemas de seguridad basadas en combinaciones de Letras y Números de un tamaño fijo de 8 caracteres los cuales deben cumplir ciertos criterios.

1. Las contraseñas generadas no pueden contener 3 letras consecutivas o 3 números consecutivos.
2. Si la contraseña termina con un número no puede terminar con un número, mientras que si empieza con letra puede terminar en número o letra.
3. La combinación no puede empezar por una vocal.

Además, se debe considerar que las letras a utilizar son todas minúsculas y los números corresponden a los dígitos, es decir, del 0 al 9. Para resolver este problema se debe utilizar el **lenguaje de programación C**, mediante la utilización de la técnica **Backtracking**.

A. Entrada

En un archivo de texto titulado "Entrada.in" se encuentran listados todas las letras y números a utilizar para formar las contraseñas.

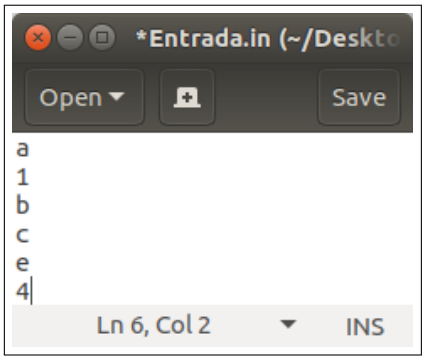

Fig. 1. Ejemplo del alfabeto ingresado en el archivo de entrada.

B. Salida

Al compilar el programa y ejecutarlo se debe generar un archivo de salida "Salida.out" el cual contiene las combinaciones válidas generadas (luego de aplicar los filtros mencionados anteriormente) enumeradas.

3. MARCO TEÓRICO

A. Backtracking

Backtracking o también conocida como **Búsqueda con retroceso**, el método consiste en resolver problemas que deben satisfacer un determinado tipo de restricciones, donde el orden de los elementos de la solución no importa. Estos problemas consisten en un conjunto de variables[2], en este caso cada posición, entre los 8 caracteres, a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema. De esta forma se va creando una especia de árbol en profundidad y en cada ramificación (o hijo) se analiza la nueva potencial combinación, de esta forma si la combinación es inválida se devuelve y se prueba con otra potencial combinación, por esto, por lo cual surge una especie de poda de la rama por lo tanto el tiempo de ejecución se ve reducido, no así su orden de complejidad.

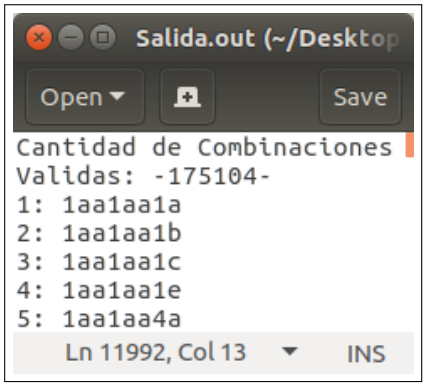

Fig. 2. Ejemplo del alfabeto ingresado en el archivo de entrada.

- Normalmente posee un menor tiempo de ejecución que la técnica de **Fuerza Bruta**.
- En el peor de los casos se comporta como fuerza bruta (orden de complejidad).

4. DESCRIPCIÓN DE LA SOLUCIÓN

La solución del problema se puede enumerar de la siguiente manera.

1. Se lee el archivo de entrada: ***Entrada.in*** y se guardan los caracteres en ***lista caracteres***. $O(n) = n$.
2. Se comienzan a crear las combinaciones de forma recursiva y en profundidad, es decir, se inserta el *i-ésimo* carácter en la posición p.
3. Se verifica si la combinación parcial (en caso que el largo sea menor a 8) cumple con las condiciones expuestas en el enunciado. De ser así, se realizan **n** llamados recursivos con cada uno de los caracteres en la posición $p + 1$. Si no cumple con las 3 condiciones propuestas se deshecha la combinación.
4. Si la combinación posee un largo igual a 8, se verifica que ésta cumpla con las condiciones si lo cumple ésta es escrita en el archivo de salida ***Salida.out***, de lo contrario, la combinación es eliminada.

A. Diseño de la solución

Para efectos de este laboratorio el enunciado solicita que los caracteres permitidos a utilizar sean letras minúsculas y números del 0 al 9 (dígitos) es por esto que se elige utilizar la representación ASCII de los caracteres en la implementación en el lenguaje de programación C, donde un carácter es interpretado internamente como un entero según su representación en el *código estándar Estadounidense para el intercambio de información*, el cual posee una serie de caracteres imprimibles según la figura 3.

En la imagen 3 se aprecia que los caracteres en su representación ASCII son los *elementos* mayores que 48 y menores que 57 (dígitos) y por otro lado los mayores a 97 y menores que 122 (letras minúsculas). Es importante aclarar esto para entender de mejor forma el código fuente donde se encuentra implementada la solución de este problema.

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Fig. 3. Tabla de Caracteres imprimibles ASCII.

B. TDA Utilizado: Listas

El primer problema que surge es obtener cada uno de los caracteres a ocupar en la generación de contraseñas y ocupar un TDA (Tipo de Dato Abstracto) adecuado en la implementación del código en C, cabe destacar que la implementación del código se respeta el estándar ANSI C y no se utiliza ninguna librería distinta a *stdio.h* y *stdlib.h* correspondientes al manejo de entrada-salida y a la librería de un correcto manejo de memoria respectivamente, ésta última cobra una gran importancia ya que es gracias a ella que es posible implementar una programación dinámica respecto a la asignación de memoria ya que resulta imposible calcular a *priori* la cantidad de combinaciones válidas que resultarán, por lo cual es necesario reasignar memoria antes de agregar cada lista a nuestra *listas de listas* de esta forma se optimiza el uso de memoria. Se utiliza una lista doblemente enlazada implementada mediante cursor, la implementación del TDA se encuentra en el archivo de código fuente *2Enlazadas-Cursor.c* adjunto a este informe (también disponible en github) ya que éste TDA permite un manejo adecuado para los *strings*, los cuales son manejados como una lista de caracteres de un tamaño fijo igual a 8, donde cada elemento contenido en la lista representa a un carácter partiendo desde la posición 0 hasta la última posición correspondiente a la número 7, es decir, la *lista combinación* es una lista de caracteres que si se imprime de forma concatenada, da origen a una combinación generada por el algoritmo que se explicará en la sección D.

C. Lectura de datos de entrada

Los caracteres a utilizar se encuentran enlistados uno por cada línea como se puede apreciar en 1 lo cual facilita captar cada uno de los caracteres, en primer lugar se lee y se analiza si éste es una letra minúscula o si es un dígito, de no ser ninguno de los casos anteriormente mencionados el programa finaliza e imprime en el archivo de salida (Salida.out) que no se puede utilizar el alfabeto ingresado en el archivo de entrada, en caso de que sí sea un carácter permitido se comprueba que éste no se haya agregado previamente a la lista con los caracteres a utilizar, si ya se encuentra el programa finaliza e imprime el mismo mensaje anteriormente mencionado. Si cumple con las dos condiciones anteriormente mencionadas (que sea un carácter permitido y que éste no se encuentre ya en la lista de caracteres a utilizar) es agregado a la lista de caracteres a utilizar y procede a preguntar por el siguiente carácter en la lista.

D. Algoritmo de Backtracking

El algoritmo consiste en un caso base al cual se debe llegar, el cual consiste en tener un largo igual a 8, si el largo de la contraseña es igual a 8 y cumple con las condiciones es escrita en el archivo de salida. Si no cumple con la condición, es desechada. Por otro lado, si la contraseña posee un largo menor a 8 se hace lo siguiente: Para cada carácter **c** en la lista Alfabeto, se inserta en la posición **p** que corresponda (inicialmente $p = 0$). Si la combinación parcial que se generó cumple con las condiciones de filtro, se realiza un llamado recursivo con $p = p + 1$. El algoritmo realiza n llamados recursivos y cada llamado reduce en 1 el problema y por último, el orden del problema no recursivo es de orden $O(n) = 1- > k = 0, a = n, b = 1$. Por lo tanto, el teorema fundamental indica que el orden de complejidad en el peor de los casos es de: $O(n) = n^n$.

Algorithm 1. Algoritmo Backtracking Passwords

```
1: procedure BACKTRACKINGPWD(lista Alfabeto, lista combinacin, entero p)   ▷ largoCaractes = n
2:     if p<8 then
3:         for i = 0; i < n; i++ do
4:             combinacion[p] = Alfabeto[i]
5:             if aplicarFiltro(Combinación) then
6:                 BacktrackingPwd(alfabeto,combinación,p + 1)
7:     elseif aplicarFiltro(combinacion)
8:         imprimirSalida(combinacion)
```

5. ANÁLISIS DE RESULTADOS

En el siguiente capítulo se pondrá a prueba si efectivamente el código funciona, además de la robustez del código respecto a la respuesta a ***eventos inesperados***, como que el alfabeto a utilizar esté vacío, contenga un caracter repetido o contenga un caracter inválido como una letra mayúscula o la letra ñ. En cada subcapítulo se mostrarán las imágenes correspondientes a la consola luego de compilar y ejecutar el programa además de la entrada y salida de los archivos.

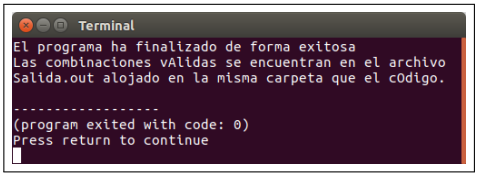

Fig. 4. Consola para el caso del camino feliz

A. Camino Feliz

Se realiza una prueba con 6 caracteres = {a,1,b,c,e,4}. Por lo tanto se esperan $6^8 = 1.679.616$ combinaciones totales (a priori), sin

embargo, Backtracking no realizará todas las combinaciones e irá desechando las combinaciones que no sean viables según las condiciones dadas. La prueba finaliza con éxito, el número

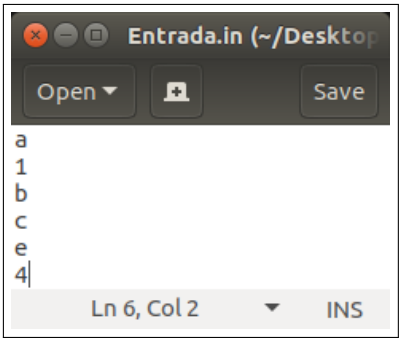

Fig. 5. Archivo de entrada para el caso del camino feliz

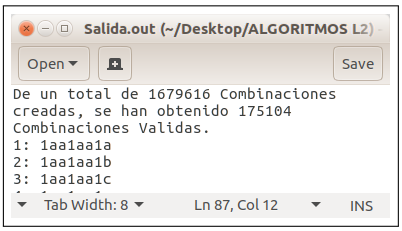

Fig. 6. Archivo de salida para el caso del camino feliz Backtracking

de combinaciones válidas coincide con el número de combinaciones válidas generadas por *Fuerza Bruta* en la experiencia del laboratorio anterior, sin embargo, a priori se experimenta una mejora en tiempo de ejecución en la creación de contraseñas. Además, mediante una simple inspección visual en la figura 6 se puede apreciar que ninguna de las 3 condiciones impuestas por el enunciada son violadas dentro de las combinaciones válidas impresas en el archivo *Salida.out*.

B. Caracter no permitido en el archivo de entrada.

En esta subsección se mostrará el comportamiento del programa cuando encuentra un caracter que no es válido según las condiciones que se dieron al problema, en este caso para cualquier caracter distinto a una letra minúscula o a un número distinto a un dígito (0-9).

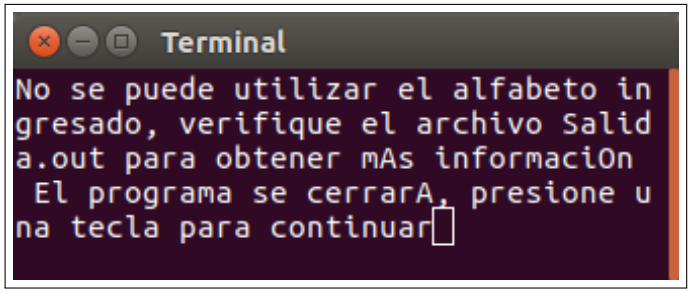

Fig. 7. Consola cuando existe un caracter inválido en el archivo de entrada

Como se puede apreciar en la figura 7 y 9 el código responde de buena forma, terminando de forma inmediata al encontrar el

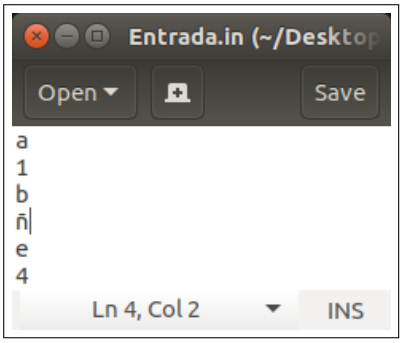

Fig. 8. Archivo de entrada cuando existe un caracter inválido en el archivo de entrada

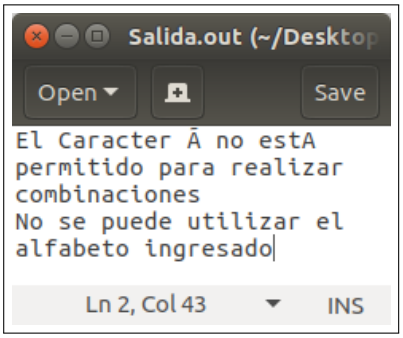

Fig. 9. Archivo de salida cuando existe un caracter inválido en el archivo de entrada

primer elemento que no corresponda a los caracteres permitidos, sin embargo en la figura 9 se aprecia que al imprimir el caracter en el archivo de salida no lo hace de forma correcta ya que estamos trabajando en ASCII y el caracter ñ corresponde al caracter número 164 por lo tanto corresponde a ASCII Extendido, es por esto que no es posible imprimirlo de forma correcta.

C. Caracter duplicado en el archivo de entrada

En esta subsección se pondrá a prueba el código para ver su comportamiento en el caso que exista un caracter duplicado en el alfabeto, ya que si éste lo permite se formarán combinaciones repetidas lo cual perjudicaría la calidad de nuestra solución.

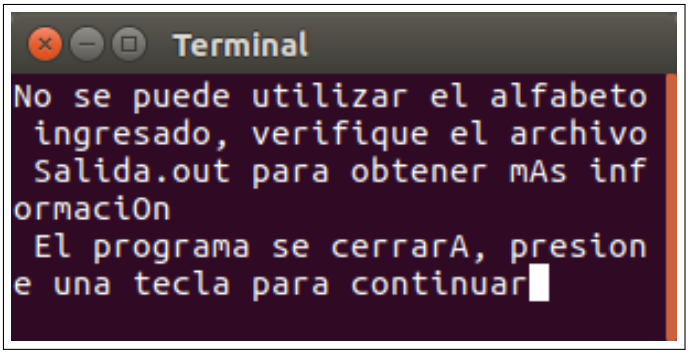

Fig. 10. Consola cuando existe un caracter duplicado en el archivo de entrada

En la figura 12 se aprecia que el caracter si se puede imprimir de forma correcta ya que éste si corresponde a el código ASCII

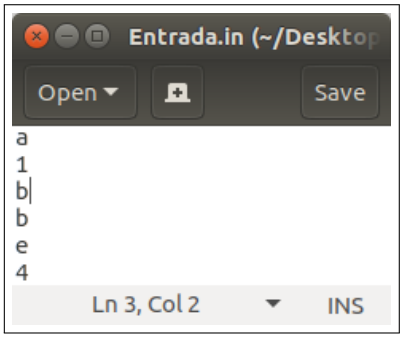

Fig. 11. Archivo de entrada cuando existe un caracter duplicado en el archivo de entrada

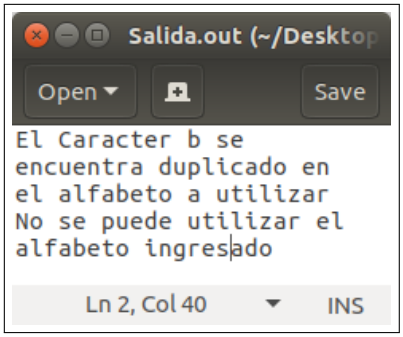

Fig. 12. Archivo de salida cuando existe un caracter duplicado en el archivo de entrada

normal (y no al extendido) como en el caso de la figura 9 donde el caracter ñ corresponde a ASCII Extendido.

6. TRAZA DE LA SOLUCIÓN

En esta sección se mostrará la traza de la solución generada por el archivo de entrada que se muestra en la figura 13, es necesario acotar el problema ya que la cantidad de combinaciones totales crece de forma exponencial como se ha visto durante el desarrollo de la experiencia, y 3 caracteres da un número de combinaciones viables para analizar y visualizar en la terminal de nuestro computador. De acuerdo al algoritmo 1 (Backtracking) en la combinación inicialmente vacía se inserta el primer elemento del Alfbeto, en este caso una *a* y si esta combinación parcial cumple con las restricciones se realiza un llamado recursivo con las siguientes posiciones, sin embargo, como en la posición 0 hay una vocal se viola la restricción que la combinación no puede empezar con una vocal, por lo tanto se elimina esta rama y se deja de combinar, como se puede apreciar en la figura 14 en la cual se pasa directamente al siguiente carácter, en este caso el 1. Sigue buscando candidatos de forma recursiva hasta que se encuentra con la combinación '1aaa' donde nuevamente se viola una restricción (3 letras consecutivas), por lo tanto se poda esta rama del árbol y se procede con el siguiente carácter, en este caso '1'. El retroceso puede observarse en la ante-penúltima salida en consola, en la combinación '1aa1aab' ya que se encuentra con una combinación en la cual hay 3 letras de forma consecutiva **aab** (las últimas 3) y como la b es el último elemento hijo de esa combinación retrocede y cambia el elemento padre al siguiente carácter, en este caso la transición es de a->1 y sigue buscando el resto de combinaciones.

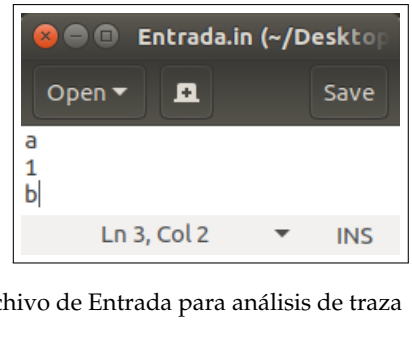

Fig. 13. Archivo de Entrada para análisis de traza

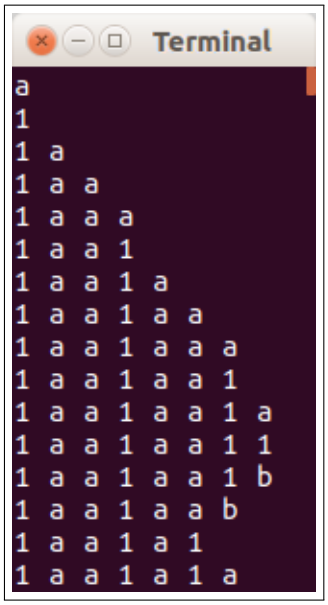

Fig. 14. Combinaciones Posibles sin filtrar

Se puede apreciar que bastaron 11 llamados a la función para encontrar la primera combinación válida. Esto habla muy bien del método ya que a diferencia de su antecesor (Fuerza Bruta) éste realizó todas las combinaciones posibles con el caracter 'a' al comienzo de las combinaciones lo cual era innecesario ya que si el primer elemento invalidaba la combinación, todas las combinaciones que nazcan de él serán igual de inválidas lo cual perjudica el tiempo de ejecución del algoritmo.

Se puede apreciar como en la primera segunda y tercera iteración varía el elemento en la posición 8, y como son sólo 3 caracteres luego varía el de la posición séptima, y luego la octava hasta que no queden caracteres por iterar en la séptima posición y así consecutivamente hasta que se acaben los caracteres a iterar en la primera posición. Cabe destacar que estas son el total de las combinaciones, luego en el código fuente se aplica un filtro con las 3 condiciones descritas al comienzo del informe y da orígen a las ***combinaciones válidas***, las cuales se muestran en la figura 15.

En la figura 15 se puede ver con claridad que no hay ninguna combinación que posea 3 letras o 3 números de forma consecutiva, o alguna combinación que comience con vocal o que comience con un número y termine con un número, es decir, ninguna combinación inválida. En la figura 16 se puede ver de mejor forma el comportamiento del algoritmo de backtracking o retroceso hasta encontrar las primeras tres combinaciones válidas. Las línea rojas equivalen a un retroceso en el algoritmo ya

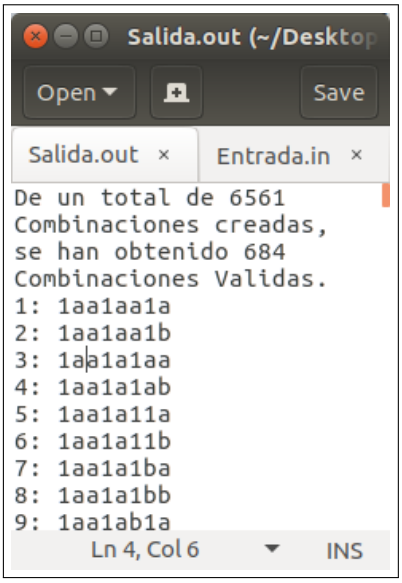

Fig. 15. Combinaciones Válidas, luego de aplicar el filtro

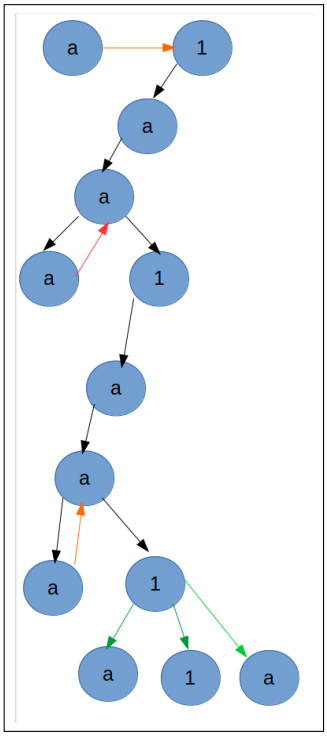

Fig. 16. Traza solución como un árbol

que nos encontramos frente a un algoritmo inválido, las líneas negras equivalen a los resultados de los llamados recursivos, es decir, el siguiente elemento que es insertado en la combinación, cada nivel del árbol hace referencia a la posición en la cual es insertado, finalmente, las líneas verdes equivalen a las combinaciones válidas encontradas.

7. CONCLUSIONES

Si recordamos la descripción de la solución, en el capítulo 4, se calcula cada uno de los órdenes de las etapas principales para el desarrollo de la experiencia, es decir, $O(n) = n + n^n + n^2 + 1$. Sin embargo, el orden que predomina es el del paso correspondiente a la generación de contraseñas, ya que el orden de complejidad mayor predomina sobre los menores, entonces la solución posee un orden de complejidad $O(n) = n^n$. El orden de $O(n) = n^n$ es muy alto, esto significa que mientras más caracteres posea nuestro archivo de entrada el tiempo de ejecución también aumentará, no de forma exponencial como la versión de Fuerza bruta, ya que como se vio durante el desarrollo de la experiencia, si los caracteres ingresados son, por ejemplo, una vocal el algoritmo se ahorrará todas las ramas que comiencen con ella. Lo cual se considera una mejora frente a fuerza bruta ya que en el caso de las contraseñas de nuestras credenciales utilizan 25 caracteres, además, de los 10 dígitos obtendremos un total de $35^8 = 2,251 \cdot 10^{12}$ combinaciones de 8 caracteres, en otras palabras, más de 2,25 trillones de combinaciones potenciales a formar, En este caso Fuerza Bruta genera y analiza cada una de ellas, por otro lado, Backtracking utiliza su capacidad de podar y evitar crear ramas que sean innecesarias porque en cierto momento la combinación se invalida tras la inserción de un carácter por lo tanto no tiene sentido seguir agregando ni analizando ésta potencial combinación porque no habrá nada que la vuelva a convertir en válida y justamente esto es lo que hace este algoritmo, deshecha la combinación y procede con las restantes. De éste método mejora su tiempo de ejecución, sin embargo, en el peor de los casos su orden de complejidad empeora hasta igualar el orden de complejidad de **Fuerza Bruta**. Por otro lado, queda demostrado que si mejoramos nuestros niveles de seguridad en nuestras contraseñas (aumentando el largo de las contraseñas, o la cantidad de caracteres distintos, alfa-numérico, etc) dificultando la posibilidad de obtener nuestras credenciales por medio de uno de estos ataques (Fuerza Bruta o Backtracking) debido a que la complejidad de estos problemas depende de n, donde n es la cantidad de caracteres, en teoría una contraseña de 6 dígitos formada por números y letras podría tardar hasta 20 días, mientras que una de largo 10 podría tardar hasta siglos en romperla [1]. A pesar de que resulta un poco más complejo en su implementación, respecto a **Fuerza Bruta** se recomienda utilizar este método frente a su contrincante, sin embargo se recomienda tener cuidado ya que al ser recursivo éste se expande dejando estados pendientes lo cual se traduce en ser más costoso en términos de memoria.

REFERENCES

1. P. Gutiérrez, "¿Cuanto tardaría un hacker en reventar nuestra contraseña?" Opt. Express **22**, (2012).
2. J. Mañas, "Análisis de Algoritmos: Complejidad" Departamento de Ingeniería en Sistemas Telemáticos **1**, (1997).
3. M. Villanueva, "Algoritmos: Teoría y aplicaciones" Departamento de Ingeniería Informática, Universidad de Santiago de Chile.**42-43**, (2012).