Proyecto – Algoritmos probabilísticos

Jefferson Amado Peña Torres - Código 1425590, Jorge Leonardo Solis Banguera - Código 1432283,
Eduardo Saavedra Perafán - Código 1210076

Escuela de Ingeniería de Sistemas y Computación, Universidad del Valle Santiago de Cali, Colombia (2016)

jefferson.amado.pena@correounivalle.edu.co, jorge.solis@correounivalle.edu.co,

eduardo.saavedra@correounivalle.edu.co

Resumen: **El siguiente documento es el desarrollo del proyecto de simulación Computacional planteado por el docente Ing.Carlos Andrés Delgado Saavedra de la Universidad del Valle. El planteamiento es el la solución de un Sudoku a partir de la lectura de un archivo de texto, para este desarrollo se debe utilizar el algoritmo probabilítico de Las Vegas.**

I. Introduction

Sudoku es un juego rompecabezas numérico con una o varias soluciones que son combinatoriales, según los números iniciales en la matriz. El objetivo es llenar la matriz generalmente 9x9, con dígitos, donde cada columna, fila y submatriz 3x3 debe tener exactamente la numeración del 1 al 9, sin replicación, El planteamiento del proyecto es el de resolver un problema Sudoku dado una matriz con ceros y algunos valores válidos preestablecidos en un archivo de texto plano, donde estos ceros indican que esta es una posición que puede tomar un número válido según las restricciones del sudoku.

Las restricciones para una malla sudoku de 9x9 son:

1. Cada fila debe tener exactamente un número del 1 -9
2. Cada columna debe tener exactamente un número del 1 -9
3. Cada bloque o submatriz debe tener exactamente un número del 1 -9

La codificación se realizó con el lenguaje de programación Python, que permite la comprensión de los algoritmos, dado las características propias del lenguaje.

Este informe no busca hallar la solución, más rápida, ni presenta una nueva metodología para resolver el rompecabezas sudoku, solo establece una estrategia de solución a este problema, basado en un algoritmo probabilista, como Las Vegas, hallando la solución mediante decisiones aleatorias.

El documento se encuentra organizado de la siguiente manera: primero la definición de la estrategia de la lista **probador,** luego la definición del algoritmo probabilista utilizando esta estrategia y tomando decisiones al azar, posteriormente el análisis descriptivo de la ejecución del algoritmo con una matriz de prueba, la forma de ejecutar el aplicativo y finalmente las pruebas, análisis y conclusiones.

II. Estrategia de solución con el algoritmo las vegas

Una de las estrategias sorpresivas en el diseño de algoritmos es la incorporación de elementos o estados **aleatorios** en el proceso computacional del algoritmo, que en la práctica puede acelerar significativamente la velocidad (según la estrategia) de en la búsqueda de la solución.

Un caso propiamente dicho de la anterior afirmación es el algoritmo Las Vegas, que presenta dos tipos: el puramente aleatorio o el que aleatoriza métodos deterministas, la estrategia tomada para la solución del sudoku es el uso de un algoritmo puramente aleatorio.

A. Algoritmo

1. Se crea una lista de los números que están en la matrix y ordena según la ocurrencias de estas (**probador**), en este deben estar todos los dígitos del [1 -9], si no esta el

faltante se pone de último (tiene sentido, su ocurrencia es 0).
2. Mientras no halle solución haga:
3. Por cada "elemento" de **probador,** revise si este "elemento" ya está en **cada** fila de la matriz
4. Si está en esta en la fila continue (caso donde en esa fila ya está este número)
5. Si no está en la fila busque las posiciones donde este "elemento" es válido (válido es, que sea correcto según las restricciones)
6. Escoja al azar una de estas posiciones donde es válida, cambie el valor de esta posición y continúe con la siguiente fila (descrito en el paso dos como... cada fila de la matriz...)
7. Si llenaste se logró poner todos los "elementos" es un éxito, si no es un fracaso, vuelva al paso 2

B. Descripción completa

Lo primero es la lectura del archivo de texto plano (*txt) con los datos iniciales del Sudoku como se muestra en Fig.1

el archivo de texto

0 2 0 6 0 0 0 0 0
0 7 6 1 0 0 9 0 0
0 0 0 7 0 0 2 0 0
8 0 0 0 0 9 0 0 4
2 1 0 0 8 3 0 0 0
4 0 0 0 0 0 0 0 7
0 0 0 0 0 0 0 6 5
0 0 4 0 6 7 0 0 0
0 0 0 0 0 0 0 3 0

Fig. 1. Matriz inicial del Sudoku

Con base en esta matriz y siguiendo el algoritmo se crea una lista de números ordenados a partir del número de veces que aparece cada número (ocurrencias) en la matriz, por ejemplo, para una matriz como la de la Fig.1, el número 6 aparece 4 veces, el número 7 con 4 ocurrencias y así sucesivamente se obtiene un vector lista o vector con los números ordenados **(6,7,2,4,1,3,8,9,5)**, esta es una estrategia que permite hallar la solución, que fue discutida y analizada por el grupo, además de esa manera se asegura que en cada fila hay un dígito.
--Restricción No.1.

Con esta lista (**probador**), se inicia la validación de cada uno de los elementos del probador, para la Fig.1 , donde el probador es **(6,7,2,4,1,3,8,9,5),** se inicia con 6, siguiendo el algoritmo, las primeras dos filas son saltadas dado que esta ya tienen 6 ubicados, luego la tercera fila no tiene 6.

En la tercera fila de la Fig.1, obtiene todas posibles lugares que tenga un "0", que indica que esta posición puede ser modificada y que además el elemento del probador en esta posición sea válido, una posición válida es un estado basado en las restricciones donde un número no aparece más de una vez en una misma fila, ni en una misma columna y en un mismo cuadrante o submatriz de 3x3.

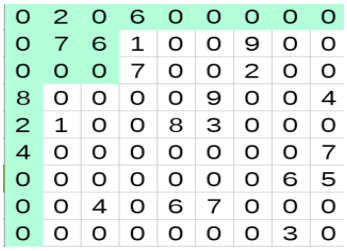

Fig. 2. Filas, columnas y cuadrantes de sudoku

Para el ejercicio de ejemplo que se está llevando a cabo, el elemento del **probador**, el número 6, donde se está obteniendo todas los posibles posiciones, pero para este caso hay una sola posicion valida en esta fila (color azul Fig.3), dado que en la primera submatriz, color amarillo, existe un 6 (color naranja) y en la segunda submatriz, color morado, existe un 6 (color naranja) y en la tercera submatriz solo se puede en la última columna, pues en la penúltima de esa fila la --Restricción No.2. no es una posición.

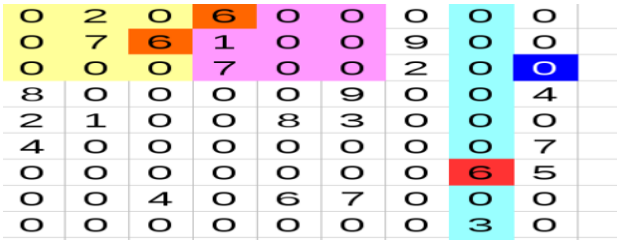

Fig. 3. Posiciones válidas para el elemento del probador

Esta es una situación particular para esta fila, el 6 se establece en la matriz, color gris en Fig.4, y continuando con la ejecución, puede existir más de una posición válida y es aquí donde este algoritmo usa el azar para hallar la solución o fracasar en el intento. Fig.4, para el elemento del probador "6" ahora en la fila 4, hay dos posibilidades,color azul y naranja, si escoge la posición, si escoge la posición (4,7), color naranja, en la próxima ejecución, cuando intenta asignar el elemento del probador "6" en la próxima fila **FALLA.**

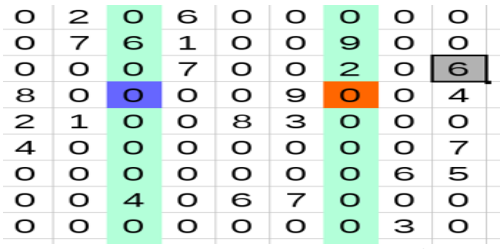

Fig. 4. Decisiones debido a decisiones desafortunadas

Como el algoritmo probabilista, toma decisiones al azar, que pueden ser o no desafortunadas ya que con una combinación aleatoria no encuentra una solución, lo admite, pero vuelve a iniciar la búsqueda de la solución, pero es de notar que dado que el algoritmo no tiene memoria (no tiene histórico de resultados), puede volver por ese "camino".

Si se analiza con detenimiento una vez se escoge al azar un posición, esta describe una camino, como un grafo que puede ser tan amplio como sea la cantidad de posiciones válidas en una fila, hay una variedad de caminos y posibilidades, dado que es un problema combinatorial y solo algunas combinaciones llevan al éxito, como se planteaba anteriormente, si al escoger de manera aleatoria la posición (4,7), en lugar la posición (4,2) vemos que conduce al **FALLO**, la misma situación se presenta cuando se escoge el número 7 del **probador,** se piden las posiciones válidas en la fila 4 de la matriz, [(4,3), (4,5), (4,5)] y se escoge al azar la primer a posición llevando el algoritmo al **FALLO**. Fig.5

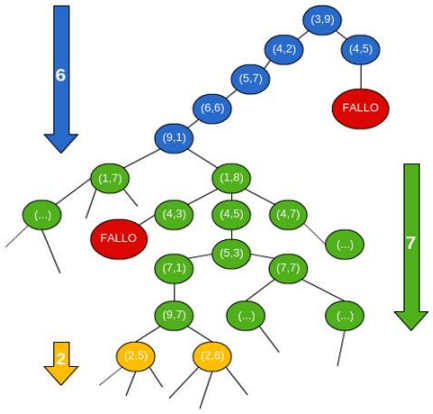

Fig. 5. Descripción de caminos posibles.

Si aleatoriamente, el algoritmo halla la combinación de la solución y reporta un **ÉXITO**, termina su ejecución.

Algoritmos como el que se plantea toma decisiones al azar para encontrar una solución al sudoku antes que un algoritmo determinista, como por ejemplo backtracking, restricciones, entre otros, sacrificando eficiencia, claro que este problema ha sido discutido ampliamente y se han propuesto metodologías con algoritmos genéticos, estocásticos que tienen un metodología donde el azar tiene un papel importante y lo que hay que destacar es que estas soluciones pueden llegar a ser más rápidas en algunos casos.

III. EJECUCIÓN DEL ALGORITMO

Como se ha indicado anteriormente la aplicación ha sido codificado con python, para ambas versiones v2.7 o v3.0, en necesario tener instalado el compilador de este lenguaje, la aplicación cuenta con un diseño orientado a objetos, posee dos módulos principales, en dos clases, un manejador de archivos (filehandle.py) y la implementación de la estrategia (sudoku.py), que se presenta en este informe.

Adicionalmente cuenta con una interfaz gráfica para la carga y despliegue de resultados, para esta es necesario contar con las librerías Pygame (http://pygame.org/hifi.html) y Thinker (https://wiki.python.org/moin/TkInter) de python.

La interfaz Fig.6 carga el sudoku propuesto en la Fig.1

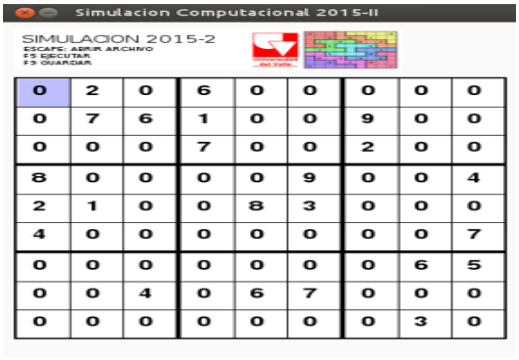

Fig. 6. Interfaz gráfica de usuario

La ventana de la aplicación, es simple pero útil para la carga de los datos, en la parte posee una descripción de sus funciones principales (cargar, ejecutar y guardar), la carga se realiza pulsando la tecla ESPACIO Fig.7

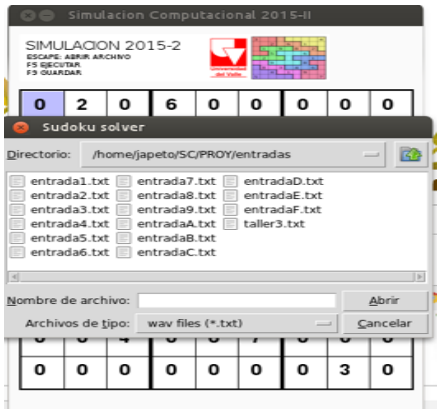

Fig. 7. Carga de documentos en la aplicación

Luego se despliega el cuadro de diálogo que permite la carga de las entradas, una vez seleccionado gráficamente se verá la actualización de la matriz, para resolver el rompecabeza se pulsa la letras F5, una vez terminado cambia los ceros de la interfaz (siempre haya la respuesta, se puede demorar mucho, pero la haya), si la solución se desea conservar, se pulsa la tecla F8 y guarda la matriz de la interfaz en un archivo de texto plano con un cuadro de diálogo similar al de abrir el archivo.

El aplicativo cuenta con la documentación del código fuente donde se especifica la características de cada función y otros comentarios para los programadores Fig.8

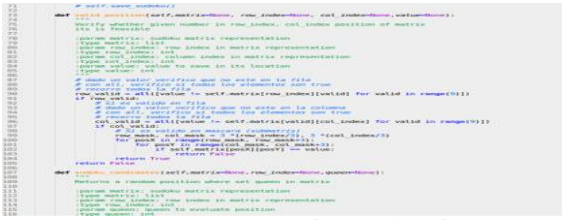

Fig. 8. Documentación y codificación

Las pruebas y ejecuciones se hicieron bajo un sistema operativo Linux, Ubuntu 4GB de ram y un procesador Core i3, para lo cual se detalla en este documento un capítulo completo.

IV. PRUEBAS DEL ALGORITMO

Las pruebas se realizaron con diversas entradas obtenidas de varias fuentes de internet, algunos detalles en las referencias, con diversos niveles de complejidad, un ejemplo de esta es la siguiente matriz.

A. *Prueba 1*

Matriz de entrada

2	0	0	0	0	4	3	7	0
0	6	3	0	0	0	0	2	0
0	0	0	0	0	2	1	0	4
0	0	5	1	0	7	0	0	0
8	7	0	0	0	0	0	4	1
0	0	0	5	0	6	7	0	0
3	0	2	4	0	0	0	0	0
0	8	0	0	0	0	4	6	0
0	4	9	8	0	0	0	0	3

Fig. 9. Conjunto de prueba 1

Una vez ejecutado el algoritmo veces para la entrada, esto con el fin de observar los tiempos aleatorios que se demora el algoritmo.

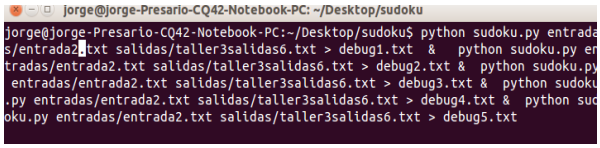

Fig. 10. Evidencia de ejecución del algoritmo para la entrada 2

Los resultados de la ejecución del algoritmo:

Debug 1: El algoritmo falla 21591 veces hasta encontrar la solución.

Debug 2: El algoritmo falla 917 veces hasta encontrar la solución.

Debug 3: El algoritmo falla 54630 veces hasta encontrar la solución.

Debug 4: El algoritmo falla 145395 veces hasta encontrar la solución.

Debug 5: El algoritmo falla 53980 veces hasta encontrar la solución.

Analizados estos datos se obtiene un promedio de falla de 50600 veces en promedio antes de encontrar la solución al problema del sudoku. Otra variable a considerar es el tiempo de esta se sacó un promedio el cual corresponde a 2 minutos promedio.

Debug 1: El algoritmo se demora 72 segundos hasta encontrar la solución.

Debug 2: El algoritmo se demora 12 segundos hasta encontrar la solución.

Debug 3: El algoritmo se demora 42 segundos hasta encontrar la solución.

Debug 4: El algoritmo se demora 122 segundos hasta encontrar la solución.

Debug 5: El algoritmo se demora 69 hasta encontrar la solución.

Promedio en encontrar la solución 63 segundos.

La solución del sudoku es:

2	1	8	9	5	4	3	7	6
4	6	3	7	1	8	9	2	5
5	9	7	6	3	2	1	8	4
9	2	5	1	4	7	6	3	8
8	7	6	2	9	3	5	4	1
1	3	4	5	8	6	7	9	2
3	5	2	4	6	9	8	1	7
7	8	1	3	2	5	4	6	9
6	4	9	8	7	1	2	5	3

Fig. 11. Solución del Sudoku conjunto de prueba 1

La Probabilidad de tener éxito en la primera ejecución con esta entrada sería 1 / cantidad promedio que el algoritmo falla, en este caso: 1/ 50600 es decir con esta entrada en este caso es de el 0,0178571%.

El número Esperado de casillas a llenar en caso de fracaso: Según la ejecución del algoritmo en promedio de casillas que llena son 3, esto se puede observar en el archivo adjunto veces_que_falla_entrada_1.txt.

Número de casillas que se deben llenar en caso de éxito: Se deben de llenar 51, esto se calcula contando las casillas que se están en 0.

1) *Análisis del tiempo esperado del cálculo de Las Vegas*

Sea s(x) tiempo promedio que demora en fallar y f(x) tiempo que demora en encontrar la solución. Se tiene los siguientes tiempos que el algoritmo falla.

Mensaje	Tiempo	Milisegundos
FALLE!	2016-02-29 01:05:07.479237	0
FALLE!	2016-02-29 01:05:07.480087	850
FALLE!	2016-02-29 01:05:07.480360	273
FALLE!	2016-02-29 01:05:07.480629	269
FALLE!	2016-02-29 01:05:07.481261	632
FALLE!	2016-02-29 01:05:07.481532	271
FALLE!	2016-02-29 01:05:07.481802	270
FALLE !	2016-02-29 01:05:07.482084	282

Tabla. 1. Tiempos cuando el algoritmo falla prueba 1

En promedio 355 milisegundos

$s(x) = 0.355\ s$

$f(x)$
$= Tiempo\ promedio\ en\ encontrar\ la\ solución. Corresponde\ a\ 63\ segundos$

$$E(x) = p(x) * s(x) + (1 - p(x)) * f(x)$$

$$E(x) = (0.000178571 * 0.355) + (1 - 0.000178571) * 63$$

$$E(x) = 62{,}98881342\ s$$

2) Análisis del tiempo de generación de los pseudoaleatorios

Realizando un análisis del comportamiento del algoritmo con esta entrada, observamos que el promedio de valores que ingresa es 7 cuando falla, y el tamaño del rango de generación de pseudoaleatorio máximo es 3, es decir un número entre 0 y 2.

Para obtener el tiempo de generación del pseudoaleatorio se plantea la siguiente prueba:

```
print "Inicial ", datetime.datetime.now()
random.randint(0,3)
print "Final ",datetime.datetime.now()
```

Fig. 12. Código de tiempo de generación pseudoaleatorio en Python

Esto arroja como resultado lo siguiente:

Inicial 2016-02-2 9 01:41:30.420673
Final 2016-02-29 01:41:30.420693

Generando un número pseudoaleatorio, el computador demora 0,020 seg aproximadamente, si se dice que en un caso de falla realiza 7 veces llamados a numeros pseudoaleatorio se obitene 0,14s en la generación de estos números; el tiempo promedio de este algoritmo en fallar es 0.355s lo que concluimos que al generar los números pseudoaleatorios toma el 39,43% del tiempo de ejecución del algoritmo en caso de falla.

En el caso que el algoritmo tenga éxito este realiza 51 veces el llamado a la generación de pseudoaleatorios, esto se convierte en 1,02s pero el tiempo en encontrar la solución es de 63s aproximadamente lo que el algoritmo en este caso emplea solo 1,6% de su tiempo en la generación de los números pseudoaleatorios.

B. Prueba 2

Fig. 13. Conjunto de prueba 2

4	8	3	9	2	1	6	5	7
9	6	7	3	4	5	8	2	1
2	5	1	8	7	6	4	9	3
5	4	8	1	3	2	9	7	6
7	2	9	5	6	4	1	3	8
1	3	6	7	9	8	2	4	5
3	7	2	6	8	9	5	1	4
8	1	4	2	5	3	7	6	9
6	9	5	4	1	7	3	8	2

Fig. 14. Solución del Sudoku conjunto de prueba 2

Los resultados de la ejecución del algoritmo

Debug 1: El algoritmo falla 37202 veces hasta encontrar la solución y demora 8s

Debug 2: El algoritmo falla 20610 veces hasta encontrar la solución 5s

Debug 3: El algoritmo falla 50211 veces hasta encontrar la solución 11s

Debug 4: El algoritmo falla 31832 veces hasta encontrar la solución 9s

Debug 5: El algoritmo falla 3676 veces hasta encontrar la solución 1s

Promedio de veces que falla: 28706

Promedio de tiempo en encontrar solucion:

Probabilidad de tener éxito en la primera ejecución con esta entrada sería del 1 / cantidad promedio que el algoritmo falla, en este caso: 1/28706 es decir con esta entrada en este caso el 0,000034836 o el 0,0034836%.

El número esperado de casillas a llenar en caso de fracaso: Según la ejecución del algoritmo en promedio de casillas que llena son 3, esto se puede observar en el archivo adjunto veces_que_falla_entrada_2.txt

Número de casillas que se deben llenar en caso de éxito: Se deben de llenar 53, esto lo calculamos contando las casillas que se están en 0.

1) Análisis del tiempo esperado del cálculo de Las Vegas

Sea s(x) tiempo promedio que demora en fallar y f(x) tiempo que demora en encontrar la solución. Se tiene los siguientes tiempos que el algoritmo falla para 8 datos.

Mensaje	Tiempo	Milisegundos
FALLE!	2016-02-29 03:42:23.798612	194
FALLE!	2016-02-29 03:42:23.798800	188
FALLE!	2016-02-29 03:42:23.798987	187
FALLE!	2016-02-29 03:42:23.799176	189
FALLE!	2016-02-29 03:42:23.799363	187

FALLE!	2016-02-29 03:42:23.799551	188
FALLE!	2016-02-29 03:42:23.799737	186
FALLE !	2016-02-29 03:42:23.799927	190

Tabla. 2. Tiempos cuando el algoritmo falla prueba 2

En promedio 188 milisegundos

$p(x) = 0{,}000034836$
$s(x) = 0.188\ s$

$f(x)$
$= Tiempo\ promedio\ en\ encontrar\ la\ solución.\ Corresponde\ a\ 6{,}8\ segundos$

$$E(x) = p(x) * s(x) + (1 - p(x)) * f(x)$$

$$E(x) = (0{,}000034836 * 0.188) + (1 - 0{,}000034836) * 6.8s$$

$$E(x) = 6{,}799769664\ s$$

2) Análisis del tiempo de generación de los pseudoaleatorios

Realizando un análisis del comportamiento del algoritmo con esta entrada, observamos que el promedio de valores que ingresa es 3 cuando falla, y el tamaño del rango de generación de pseudoaleatorio maximo es 3, es decir un número entre 0 y 2.

Generando un número pseudoaleatorio segun el experimento anterior demora 0,020 seg aproximadamente en el computador, si decimos que en un caso de falla realiza 3 veces llamados a numeros pseudoaleatorio tenemos 0,06s en la generación de estos números; el tiempo promedio de este algoritmo en fallar es 0,188s lo que concluimos que generar los números pseudoaleatorios toma el 31,91% del tiempo de ejecución del algoritmo en caso de falla.

En el caso que el algoritmo tenga éxito este realiza 53 veces el llamado a la generación de pseudoaleatorios, esto se convierte en 1,06s pero el tiempo en encontrar la solución es de 6,8s aproximadamente lo que el algoritmo en este caso emplea solo 23% de su tiempo en la generación de los números pseudoaleatorios.

C. Prueba 3 Entrada específica del proyecto

Este genera un DEBUG de 500.000 MB lo cual hace muy difícil su análisis en nuestro algoritmo.

0	2	0	6	0	0	0	0	0
0	7	6	1	0	0	9	0	0
0	0	0	7	0	0	2	0	0
8	0	0	0	0	9	0	0	4
2	1	0	0	8	3	0	0	0
4	0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	6	5
0	0	4	0	6	7	0	0	0
0	0	0	0	0	0	0	3	0

Fig. 15. Conjunto de prueba 3. Entrada específica del proyecto

9	2	1	6	4	8	5	7	3
5	7	6	1	3	2	9	4	8
3	4	8	7	9	5	2	1	6
8	6	3	5	7	9	1	2	4
2	1	7	4	8	3	6	5	9
4	9	5	2	1	6	3	8	7
7	3	9	8	2	1	4	6	5
1	5	4	3	6	7	8	9	2
6	8	2	9	5	4	7	3	1

Fig. 16. Solución del Sudoku conjunto de prueba 3

V. REFERENCIAS

[1] Tischer. Irene. (2015, Aug.). Material de Clase Simulacion Computacional Tema 3: Algoritmos probabilísticos

[2]. Algoritmos Probabilistas. Universidad de Guadalajara (1nd ed.) [Online]. Avaliable:http://dis.um.es/~domingo/apuntes/AlgProPar/0607/AlgoritmosProbabilistas.pdf

[3]. Marínez Edgardo, Análisis de algoritmos. (2015, Jul, 14). (1nd ed.) [Online].Avaliable:http://eafranco.com/docencia/analisisdealgoritmos/files/proyectofinal/ProyectoFinal.pdf