

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

COMPLEJIDAD COMPUTACIONAL

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

Reporte

La dificultad de jugar Sudoku

Profesor:

Oscar Hernández Constantino

INTEGRANTES:

Clemente Herrera Karla

Cruz Gonzalez Irvin Javier

Sánchez Sánchez Jorge Angel

Ugalde Flores Jimena

Índice

1. Introducción	1
2. Problema Sudoku	1
3. Latin Square Completion (LSC)	2
3.1. Complejidad:	4
4. Reducción de LSC al problema del sudoku	4
5. Algoritmo Verificador	6
6. Conclusiones	10
7. Referencias consultadas	10

1. Introducción

El Sudoku se crea a partir de los trabajos del matemático suizo Leonhard Euler (1707-1783) cuando se encontraba trabajando en la demostración de un conjunto de teoremas acerca del cálculo de probabilidades. Sin embargo, el Sudoku visto como un juego, tiene su origen en Indianapolis en 1979. Para entonces no se llamaba Sudoku sino simplemente Number Place (el lugar de los números), siendo publicado en la revista Math Puzzles and Logic Problems de la empresa especializada en puzzles Dell.

Posteriormente Nikoli, una empresa japonesa especializada en pasatiempos para prensa, lo exportó a Japón publicándolo en el periódico Monthly Nikolist en abril de 1984 bajo el título "Suji wa dokushin ni kagiru", que se puede traducir como "los números deben estar solos", y que posteriormente se abreviaría a Sudoku (su=número, doku=solo).

En el ámbito de las matemáticas es visto como un problema de satisfacción de restricciones. Los juegos de puzzle como el Sudoku son muy populares debido a que se caracterizan por la simplicidad de sus reglas, simplicidad que necesariamente no significa no complejo, pues por el contrario se necesita de un razonamiento metódico para llegar a una solución. Cada instancia de puzzle tiene una solución única.

El análisis de la complejidad del Sudoku ya ha sido revisada por muchos autores, entre ellos quien postula que el Sudoku es un problema puzzle de tipo ASP-completo (Another Solution Problem) a partir del cual, mediante inferencias formales, demuestra que un Sudoku de $N \times N$ casillas, es también NP-completo.

2. Problema Sudoku

Dado que el problema de Sudoku fue generalizado a ASP completo, lo que a su vez implica que es NP completo; por lo tanto, es teóricamente tan difícil como cualquier problema del conjunto NP de problemas de decisión para los cuales se puede certificar una solución positiva en tiempo polinomial. Tenga en cuenta que, aunque existen variantes de versiones más generales del Sudoku (como versiones rectangulares), la variante cuadrada, donde N es un cuadrado perfecto, es suficiente para completar NP. Por lo tanto, durante el resto de este reporte se supondrá que nos limitaremos a considerar la variante cuadrada.

El problema de Sudoku generalizado es un problema NP-completo que, efectivamente, requiere un cuadrado latino que satisface algunas restricciones adicionales:

- Se presenta una cuadrícula $n \times n$ parcialmente llena que debe rellenarse con números del 1 al n .
- Ninguna fila, columna o subcuadrícula puede tener un número repetido, es un proceso que se puede hacer rápidamente, ya que implica recorrer cada fila, columna y bloque y comprobar la validez de cada celda.
- ¿Dada una cuadrícula parcialmente llena, existe una solución válida?

Este último punto lo podemos ver como un problema de solución, aunque encontrar la solución puede requerir probar múltiples combinaciones (y en el peor de los casos, puede llevar tiempo exponencial), la verificación de cualquier solución propuesta es rápida. Esto es lo que caracteriza a un problema en NP .

Los problemas que son NP-Hard y están también en NP se llaman NP-Completo. Esto significa que si encontramos una solución en tiempo polinómico para uno de estos problemas NP-Completo, podríamos resolver todos los problemas en NP en tiempo polinómico.

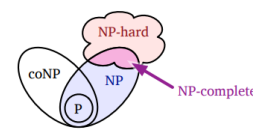
Sudoku NP

El juego del sudoku puede considerarse NP si se toma la notación $M(G, s)$, G representa el tablero de Sudoku (con algunas celdas llenas y otras vacías), y s es el certificado, es decir, la solución propuesta para completar el tablero. El tamaño de s es menor que n^2 donde n es la longitud de un lado del tablero (en el caso de un Sudoku de 9×9 , $n = 9$, esto significa que la solución completa no necesita más que n^2 valores. Verificar cada valor en el tablero de Sudoku requiere $3n$ operaciones, ya que cada valor debe cumplir tres restricciones (fila, columna y subcuadro de 3×3 en el caso de Sudoku estándar). lo cual se simplifica a una complejidad de $O(n^3)$. Esto significa que el proceso de verificación es polinómico, lo que confirma que el problema de verificar una solución de Sudoku está en NP es decir se puede verificar en ese tiempo una vez que tenemos una solución candidata.

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

NP-Hard

Un problema es NP-completo si es a la vez NP – hard y un elemento de NP. Los problemas NP completos son los problemas más difíciles en NP. Si alguien encuentra un algoritmo de tiempo polinomial para incluso un problema NP completo, entonces eso implicaría un algoritmo de tiempo polinomial para cada problema NP completo.



Intuitivamente, si pudiéramos resolver rápidamente un problema NP – Hard particular, entonces podríamos resolver rápidamente cualquier problema cuya solución sea fácil de entender, utilizando la solución de ese problema especial como una subrutina. Los problemas NP – Hard son al menos tan difíciles como cualquier problema en NP, aunque el problema NP-Hard en sí no necesariamente debe estar en NP.

Si podemos tomar cualquier problema en NP y transformarlo de manera eficiente en una instancia de Sudoku, entonces Sudoku sería al menos tan difícil como cualquier problema en NP.

3. Latin Square Completion (LSC)

Para poder demostrar que el problema del Sudoku es un problema NP-completo ocuparemos una estructura matemática llamada cuadrado latino.

Un cuadrado latino es una matriz de tamaño $n \times n$ cuyos elementos pertenecen a un conjunto finito A de cardinalidad n y cada uno de ellos aparece exactamente una vez en cada renglón y en cada columna. Tenemos que tener en cuenta que un cuadrado latino no se forma sólo con números, sino que vale para cualquier tipo de símbolos, por ejemplo, las letras del alfabeto latino que utilizó el matemático suizo Leonhard Euler, los colores utilizados por el pintor Richard Paul Lohse o los cuadrados grecolatinos.

1	2	3	4	5	
2	3	4	5	1	
3	4	5	1	2	
4	5	1	2	3	
5	1	2	3	4	

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>B</i>	<i>A</i>	<i>D</i>	<i>C</i>
<i>C</i>	<i>D</i>	<i>A</i>	<i>B</i>
<i>D</i>	<i>C</i>	<i>B</i>	<i>A</i>

Para el propósito de este proyecto se ocupara un cuadrado latino numérico. Entrando más en las definiciones que se ocuparan para nuestra reducción tenemos que: Un cuadrado latino parcial de tamaño n es una matriz M de $n \times n$ tal que sus entradas pertenecen al conjunto $\{1, \dots, n, \textcircled{S}\}$ y tal que en ninguna fila y en ninguna columna de M ningún número ocurre al menos dos veces. Por lo tanto, dado M un cuadrado latino parcial de tamaño n , diremos que M es completible si y sólo si existe un modo de reemplazar todas las ocurrencias del símbolo \textcircled{S} por elementos del conjunto $\{1, \dots, n\}$ de manera que tras realizar tales sustituciones se obtenga un cuadrado latino.

Para finalmente poder pasar a realizar nuestra reducción tenemos que demostrar que el cuadrado latino es NP-completo. Para saber esto tenemos que verificar si el problema esta en NP y si es NP-hard.

Cuadrado Latino

Primero definimos formalmente el problema del cuadrado latino:

Entrada: Un entero n , conjunto de k símbolos.

Pregunta: ¿Existe un cuadrado latino válido de tamaño $n \times n$?

Demostración

1. Problema en NP :

Para saber si esta en NP necesitamos saber si existe un verificador de solución que puede comprobar la validez de nuestra solución. Por lo tanto decimos que tenemos un verificador de nuestra cuadro latino tal que puede:

- Comprobar que cada fila contiene símbolos únicos
- Comprobar que cada columna contiene símbolos únicos

El tiempo de verificación para cada una de los dos puntos es $O(n^2)$, por lo tanto el tiempo de verificación total es de $O(n^2)$.

2. Problema en NP-hard:

En esta sección realizaremos una reducción del problema del SAT a Cuadrado Latino.

El problema de satisfacibilidad (SAT) consiste en determinar si existe una asignación de valores de verdad (verdadero o falso) a las variables de una fórmula booleana tal que la fórmula entera sea verdadera.

Dada una formula booleana en forma normal conjuntiva ϕ con n variables y m cláusulas, construiremos un cuadro latino que es válido si y solo si ϕ es satisfacible. El tamaño de nuestro cuadrado latino sera de $k \times k$ donde $k = n + m + 2$. El conjunto de símbolos que

se ocupan es $S = \{x_1, \neg x_1, x_2, \neg x_2, x_3, \neg x_3, \dots, x_v, \neg x_v, C_1, C_2, \dots, C^c, S\}$ Cada variable y su negación se representan como símbolos.

Para la estructura de nuestro cuadro tenemos:

- Bloques de las variables:
Donde n filas representan las variables y cada fila las asignaciones de las variables y sus respectivas negaciones. También tenemos el símbolo auxiliar S . Se distribuyen x_n y $\neg x_n$ de manera única.
- Bloques de cláusulas:
Tenemos m filas para verificar satisfacibilidad, donde cada fila representa una cláusula y debe contener al menos un literal verdadero. Usamos a S para completar restricciones.

Por lo tanto, si podemos construir un cuadrado latino a partir de la fórmula booleana tal que todas las cláusulas sean satisfechas, entonces existe una asignación satisfactoria para la fórmula original.

3.1. Complejidad:

- Construcción del cuadro: $O(n + m)$
- Verificación: $O(k^2) = O((n + m)^2)$

Conclusión:

El problema del cuadro latino es NP-Completo porque:

- a) Pertenece a NP
- b) SAT se reduce polinómicamente a él

4. Reducción de LSC al problema del sudoku

Para la reducción se utilizara la siguiente notación

- L es un cuadrado latino parcial de tamaño $n \times n$
- L_i Es la i -ésima fila de L
- $M(L)$ una matriz de $n^2 \times n^2$ (sudoku)
- F_i La i -ésima fila de $M(L)$
- $F_{i,j}$ El j -ésimo bloque de n entradas de la i -ésima fila de $M(L)$
- S^i matriz de $(n - 1) \times n$
- $S_{j,l}^i$ La entrada en la j -ésima fila y l -ésima columna de S^i
- F_j^i Es la j -ésima fila de la matriz S^i

A partir de un cuadrado latino parcial L de $n \times n$ se construirá un sudoku $M(L)$ de $n^2 \times n^2$ con entradas en $\{1, \dots, n^2, \textcircled{5}\}$

Primero se deben de crear las n matrices S^i

Cada una de las entradas de S^1 esta dada por $S^1_{i,j} = (i \cdot n) + j$. S^1 esta constituida por las filas C_1, \dots, C_n .

$\forall i \in \{2, \dots, n\}$, S^i es la matriz constituida por las columnas $C_{v^{i-1}(1)}, \dots, C_{v^{i-1}(n)}$, donde v es la permutación $v : [n] \rightarrow [n]$ tal que

$$v(i) = i + 1 \bmod(n)$$

Es decir, S^i es un reordenamiento de las columnas de S^1 .

Una vez con las n matrices S^i , ya se puede construir $M(L)$ con los siguientes pasos

1. Para todo $i \leq n$, en el bloque $F_{(i-1)n+1,1}$ se coloca L_i
2. Para todo $i \leq n$ y todo $j \in \{2, \dots, n\}$, en el bloque $F_{(i-1)n+j,1}$ se coloca FS^i_{j-1}
3. Para todo $i \leq n$ y todo $j \in \{2, \dots, n\}$, en el bloque $F_{(i-1)n+1,j}$ se coloca FS^i_{j-1}
4. Para todo $i \leq n$ y todo $j \in \{2, \dots, n\}$, en el bloque $F_{(i-1)n+j,n-(j-2)}$ se coloca L_i
5. Para todo $i \leq n$, todo $l \in \{2, \dots, n\}$, todo $j \in \{2, \dots, n\}$ y con la permutación $v : [n-1] \rightarrow [n-1]$ $v(i) := i + 1 \bmod(n-1)$
 - Si $l < n - (j - 2)$, entonces en el bloque $F_{(i-1)n+l,j}$ se coloca $FS^i_{v^{(j-1)}(l-1)}$
 - Si $l > n - (j - 2)$, entonces en el bloque $F_{(i-1)n+l,j}$ se coloca $FS^i_{v^{(n+j-3)}(l-1)}$

Notemos que con el paso 2 se colocan los S^i en el bloque 1. Los S^i se construyen de tal forma de que no tengan elementos repetidos y sean reordenamientos de columnas de S^1 , por lo que en el bloque 1 de $M(L)$ no se repite ni un número en fila ni en columna cuando se colocan los S^i . Con el paso 1 se colocan las filas de L en los espacios vacíos del bloque 1 de $M(L)$, es decir, se pone implícitamente L dentro de $M(L)$. Y con el pasos 3 al 5, se esta haciendo un reordenamiento en filas de los cuadrantes de $n \times n$ de los bloques $F_{(i-1)n+1,1}$ al $F_{(i-1)n+j,1}$ con $i \leq n$ y $j \in \{2, \dots, n\}$, dejando fija i . Y estos reordenamientos se colocan en los cuadrantes $n \times n$ de alado.

1	3	2	4	5	6	7	8	9
4	5	6	7	8	9	1	3	2
7	8	9	1	3	2	4	5	6
2	$\textcircled{5}$	3						
5	6	4						
8	9	7						
3	2	1						
6	4	5						
9	7	8						

L =	1	3	2
	2	$\textcircled{5}$	3
	3	2	1

S^1 =	4	5	6
	7	8	9

S^2 =	5	6	4
	8	9	7

S^3 =	6	4	5
	9	7	8

Figura 1: Ejemplo de lo que se realiza en los pasos 3 al 5 cuando $i = 1$

Así que $M(L)$ se construye con reordenamientos en columnas de S^1 y un reordenamiento en filas de S^i y L_i en cada cuadrante. Al ser puros reordenamientos, $M(L)$ no tendrá números repetidos en filas ni en columnas al menos que L , el cual se pone en los pasos 1 y 4 implícitamente en cada bloque de $M(L)$, tenga repetidos en filas o columnas. Por lo que el algoritmo de transformación cumple con la pertenencia.

Al copiar la información de matrices L y S^i en $M(L)$, lo cual se realiza en $O(n)$, al tener a lo más 3 variables (que van de 1 hasta n) sobre las que se itera para obtener que partes de las matrices L , S^i se colocan en el sudoku y las operaciones que se utilizan para obtener las filas y los bloques son operaciones constantes; sumas, multiplicaciones y modulo, entonces se tiene que el algoritmo transformador es de tiempo polinomial $O(n^4)$.

Como el algoritmo transformador cumple con la pertenencia y es de tiempo polinomial, entonces se tiene que el problema LSC se reduce al Sudoku. Y al ser el LSC un problema NP-hard que se reduce al Sudoku, entonces el Sudoku es NP-hard.

5. Algoritmo Verificador

Explicación Formal del Código

Este programa implementa la generación, validación y manipulación de tableros de Sudoku. A continuación, se explica cada componente del código:

Verificación de Validez del Sudoku

El código incluye una función llamada `verificar_sudoku`, que evalúa si un tablero de Sudoku cumple con las reglas establecidas. Estas reglas incluyen:

- Cada fila debe contener valores únicos entre 1 y 9, ignorando los ceros.
- Cada columna debe contener valores únicos entre 1 y 9, ignorando los ceros.
- Cada subcuadro de 3×3 también debe cumplir con la condición de unicidad.

```
# Función para verificar si el Sudoku es válido
def verificar_sudoku(tablero):
    for i in range(9):
        # Verificamos filas y columnas
        if not es_unico([tablero[i][j] for j in range(9)]) or not es_unico([tablero[j][i] for j in range(9)]):
            return False
    # Verificamos bloques de 3x3
    for fila in range(0, 9, 3):
        for col in range(0, 9, 3):
            bloque = [tablero[fila + i][col + j] for i in range(3) for j in range(3)]
            if not es_unico(bloque):
                return False
    return True
```

Para realizar esta verificación, se emplea una función auxiliar llamada `es_unico`, que evalúa si los elementos de una lista cumplen con la propiedad de unicidad.


```
# Función para verificar si los valores en una lista son únicos, ignorando los ceros
def es_unico(valores):
    valores_sin_ceros = [v for v in valores if v != 0]
    return len(valores_sin_ceros) == len(set(valores_sin_ceros))
```

Generación de un Sudoku Válido

La función `completar_sudoku` utiliza un algoritmo de *backtracking* para rellenar un tablero de Sudoku incompleto. Este algoritmo intenta colocar un número en cada celda vacía (0) de forma recursiva:

- Se generan opciones aleatorias entre 1 y 9 para cada celda.
- Se evalúa la validez del número mediante la función `es_valido`.
- Si el número es válido, se coloca en la celda, y la función continúa con el siguiente espacio vacío.
- Si en algún punto no hay números válidos disponibles, se retrocede y se intentan otras opciones.

La función termina cuando todas las celdas están llenas y el tablero cumple con las reglas del Sudoku.

```
# Función para rellenar el Sudoku con números válidos de manera aleatoria
def completar_sudoku(tablero):
    for i in range(9):
        for j in range(9):
            if tablero[i][j] == 0:
                opciones = list(range(1, 10))
                random.shuffle(opciones)
                for num in opciones:
                    if es_valido(tablero, i, j, num):
                        tablero[i][j] = num
                        if completar_sudoku(tablero):
                            return True
                        tablero[i][j] = 0
                return False
    return True
```

Validación de Números en Celdas Específicas

La función `es_valido` verifica si un número puede colocarse en una posición específica del tablero sin violar las reglas del Sudoku. Para ello:

- Comprueba que el número no se repita en la fila correspondiente.
- Verifica que el número no esté presente en la columna.
- Evalúa que el número no aparezca en el subcuadro 3×3 que contiene la celda.

```
# Función para verificar si un número puede colocarse en una posición sin violar las reglas
def es_valido(tablero, fila, col, num):
    if all(tablero[fila][j] != num for j in range(9)) and \
        all(tablero[i][col] != num for i in range(9)) and \
        all(tablero[fila // 3 * 3 + i][col // 3 * 3 + j] != num for i in range(3) for j in range(3)):
        return True
    return False
```

Introducción de Conflictos

La función `es_no_valido` manipula un tablero de Sudoku completado para introducir conflictos. Esto se logra seleccionando celdas aleatorias y asignándoles valores que pueden no cumplir con las reglas de unicidad. Se utiliza para generar tableros intencionalmente incorrectos.

```
# Función para hacer que el tablero no sea válido introduciendo número repetidos en alguna fila o columna
def es_no_valido(tablero, num_conflictos=3):
    for _ in range(num_conflictos):
        fila = random.randint(0, 8)
        col = random.randint(0, 8)
        tablero[fila][col] = random.randint(1, 9) # Colocamos un número aleatorio que puede no cumplir las reglas
```

Certificación de Validez

La función `generar_certificado` imprime un mensaje indicando si un tablero es válido o no. Esto se determina llamando a la función `verificar_sudoku`.

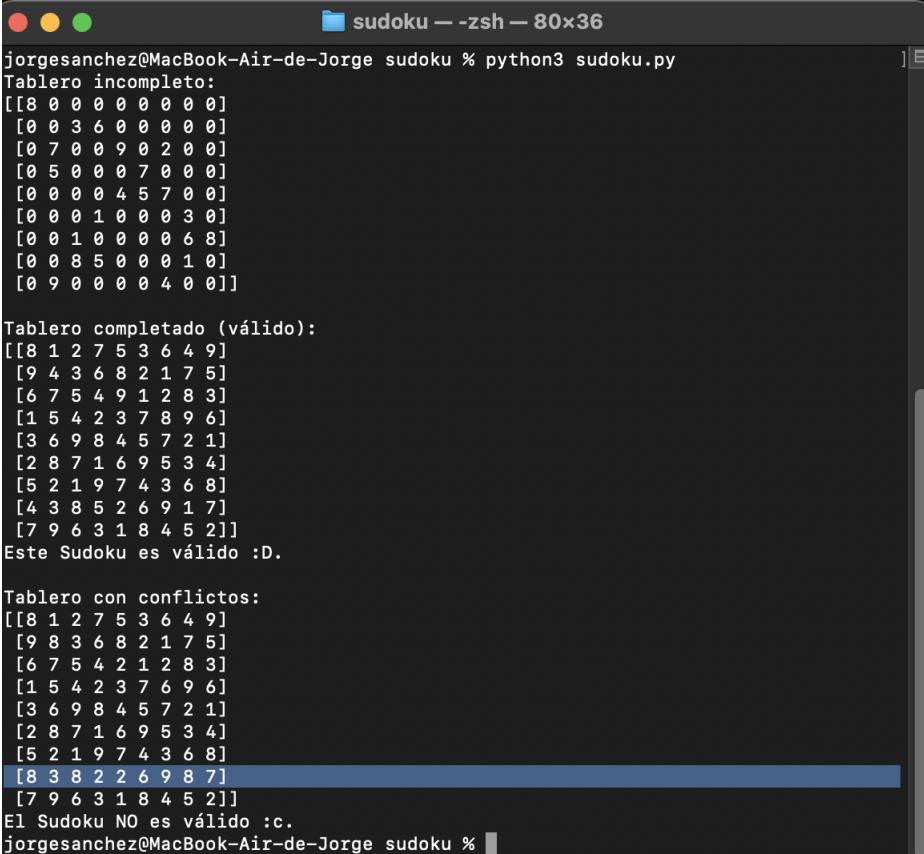
```
# Función para generar un certificado de validez e imprimirlo en la terminal
def generar_certificado(tablero):
    if verificar_sudoku(tablero):
        print("Este Sudoku es válido :D.")
    else:
        print("El Sudoku NO es válido :c.")
```

Ejemplares:

Demostración del Código

El programa incluye dos ejemplos prácticos:

1. En el primer ejemplo, se toma un tablero de Sudoku incompleto, se completa utilizando `completar_sudoku`, y se valida con `generar_certificado`.
2. En el segundo ejemplo, se introduce un número de conflictos en el tablero utilizando `es_no_valido`, y luego se valida nuevamente para confirmar que no cumple con las reglas del Sudoku.



```
sudoku — zsh — 80x36
jorgesanchez@MacBook-Air-de-Jorge sudoku % python3 sudoku.py
Tablero incompleto:
[[8 0 0 0 0 0 0 0 0]
 [0 0 3 6 0 0 0 0 0]
 [0 7 0 0 9 0 2 0 0]
 [0 5 0 0 0 7 0 0 0]
 [0 0 0 0 4 5 7 0 0]
 [0 0 0 1 0 0 0 3 0]
 [0 0 1 0 0 0 0 6 8]
 [0 0 8 5 0 0 0 1 0]
 [0 9 0 0 0 0 4 0 0]]

Tablero completado (válido):
[[8 1 2 7 5 3 6 4 9]
 [9 4 3 6 8 2 1 7 5]
 [6 7 5 4 9 1 2 8 3]
 [1 5 4 2 3 7 8 9 6]
 [3 6 9 8 4 5 7 2 1]
 [2 8 7 1 6 9 5 3 4]
 [5 2 1 9 7 4 3 6 8]
 [4 3 8 5 2 6 9 1 7]
 [7 9 6 3 1 8 4 5 2]]

Este Sudoku es válido :D.

Tablero con conflictos:
[[8 1 2 7 5 3 6 4 9]
 [9 8 3 6 8 2 1 7 5]
 [6 7 5 4 2 1 2 8 3]
 [1 5 4 2 3 7 6 9 6]
 [3 6 9 8 4 5 7 2 1]
 [2 8 7 1 6 9 5 3 4]
 [5 2 1 9 7 4 3 6 8]
 [8 3 8 2 2 6 9 8 7]
 [7 9 6 3 1 8 4 5 2]]

El Sudoku NO es válido :c.
jorgesanchez@MacBook-Air-de-Jorge sudoku %
```

6. Conclusiones

Al tener que el Sudoku es NP-Hard y que pertenece a NP, esto por el algoritmo verificador que se implemento, se tiene que el sudoku es NP-Completo. Que el Sudoku sea NP-Completo implica que encontrar una solución para un Sudoku de tamaño $n \times n$ no puede realizarse en tiempo polinomial y de aquí viene la dificultad intrínseca de completar un Sudoku, ya que de momento no se conoce un algoritmo que determine una solución en tiempo polinomial.

7. Referencias consultadas

- (N.d.). Illinois.edu. Retrieved November 20, 2024, from <https://courses.grainger.illinois.edu/cs573/fa2010/notes/30-nphard.pdf>
- Montoya, J. A. (2006). La dificultad de jugar sudoku. Revista Integración, 24(1), 13-24. Universidad Industrial de Santander, Escuela de Matemáticas
<https://drive.google.com/file/d/1rKwhb9cbD00yVLXXlQIHueDLAqnAgyxp/view>
- Fronteras, M. y. S. (2017, July 21). Cuadrados latinos. Matemáticas y sus fronteras -. <https://www.madrimasd.org/blogs/matematicas/2017/07/21/144049>
- Revista Integración. (n.d.). Redalyc.org. Retrieved November 28, 2024, from <https://www.redalyc.org/pdf/3270/327028430001.pdf>
- Tomé, C. (2015, January 14). Cuadrados latinos, matemáticas y arte abstracto — Cuaderno de Cultura Científica. Cuaderno de Cultura Científica. <https://culturacientifica.com/2015/01/14/cuadrados-latinos-matematicas-y-arte-abstracto/>
- (N.d.). Uam.Mx. Retrieved November 28, 2024, from <http://mat.izt.uam.mx/mcmai/documentos/tesis/Gen.08-0/Cortes-CI-Tesis.pdf>
- Colbourn, C. J. (1984). SQUARES. Core.ac.uk. <https://core.ac.uk/download/pdf/81928286.pdf>