

# **PRÁCTICA 1 – SUDOKU**

**Conocimiento y Razonamiento Automatizado**

Blanca Calderón González  
Franck Michael Fierro Chicaiza  
Kevin Orlando Cancio Fernández

# Índice

<b>Índice</b>	<b>2</b>
<b>I - Reparto de tareas (Lista de actividades)</b>	<b>3</b>
Representación gráfica del sudoku	3
Generación de la lista de posibilidades	3
Reglas de simplificación	3
Mejoras añadidas	3
Experiencia con Chat GPT-3	3
Memoria	3
<b>II - Grado de cumplimiento requisitos obligatorios</b>	<b>4</b>
Representación gráfica del Sudoku	4
Búsqueda de las posibilidades	4
Reglas de simplificación	6
<b>III - Mejoras realizadas</b>	<b>10</b>
Implementación de una interfaz	10
Estudio de sobre el rendimiento de las reglas	11
Estudio de sobre el grado de cumplimiento de resolución	14
Búsqueda y estudio de un algoritmo sobre Sudokus predefinido	15
Comparación entre algoritmo desarrollado y predefinido	15
<b>IV - Experiencia con Inteligencia Artificial</b>	<b>15</b>
<b>V - Errores</b>	<b>16</b>
<b>VI - Fuentes utilizadas</b>	<b>16</b>

# I - Reparto de tareas (Lista de actividades)

## Representación gráfica del sudoku

**Descripción** → Código cuya función es mostrar por la consola el sudoku de forma estructurada en filas y columnas de 9 elementos.

**Persona encargada** → Blanca Calderón

## Generación de la lista de posibilidades

**Descripción** → Código que cada vez que encuentra un '.' en el sudoku busca los posibles números que pueden ir en esa posición introduciendolos como una lista de forma que al finalizar su ejecución tengamos un nuevo sudoku con todos los puntos sustituidos por la lista de posibilidades de su posición.

**Persona encargada** → Blanca Calderón

## Reglas de simplificación

**Descripción** → Programación de las cuatro reglas necesarias para simplificar el sudoku y llegar a su solución si es que la tiene. La primera regla consiste

**Persona encargada** → Blanca Calderón (regla 0) y Franck Michael Fierro

## Mejoras añadidas

**Descripción** → Programación de una interfaz gráfica capaz de representar los sudokus, estudio sobre el rendimiento de las reglas en relación con la resolución de los Sudokus de distintas dificultades (Fácil, Normal y Difícil) sobre un banco de pruebas, estudio sobre el grado de cumplimiento de resolución de Sudokus sobre un banco de pruebas, búsqueda y estudio de un algoritmo ya implementado en prolog que resuelve Sudokus, comparación del algoritmo diseñado con el algoritmo predefinido y búsqueda de optimización en nuestro algoritmo.

**Persona encargada** → Kevin Orlando Cancio Fernández

## Experiencia con Chat GPT-3

**Descripción** → Síntesis sobre la experiencia y utilidad de nuevas herramientas de IA como Chat GPT-3 en conjunto de Prolog.

**Persona encargada** → Kevin Orlando Cancio Fernández

## Memoria

**Descripción** → Desarrollo y redacción de la memoria cuyo contenido es el mostrado en el índice.

**Persona encargada** → Blanca Calderón, Kevin Orlando Cancio Fernández y Franck Michael Fierro Chicaiza

## II - Grado de cumplimiento requisitos obligatorios

### Representación gráfica del Sudoku

Este requisito se ha cumplido al completo cumpliendo todo lo pedido en el enunciado.

Para cumplirlo se ha implementado la regla `mostrar_sudoku` el cual recibe uno de los sudokus de prueba previamente definidos. El sudoku está representado como una lista de 81 elementos correspondiendo cada elemento con una casilla del tablero 9x9.

Estos sudoku de prueba están inicializados de forma que algunas posiciones contienen un número fijo y otras están vacías lo cual se representa con el carácter '.', estas posiciones vacías son las que deberá rellenar el programa al solucionar el sudoku como se verá más adelante.

Como se ha mencionado anteriormente se implementa la regla `mostrar_sudoku` para mostrar por consola el sudoku de forma gráfica tal y como pide el enunciado. Para lograr esto, en la regla recibe sudoku a representar y de éste saca la longitud del sudoku sin contar su cabeza. Tras esto evalúa si la longitud + 1 da 0 al hacer módulo 9 ya que si lo es significa que se pasa a la siguiente línea del sudoku (con ésta condición se controla cuando se pasa a la siguiente línea del tablero 9x9 ya que en el programa se representa como una sola lista y no una lista de listas de 9 elementos que corresponden a los elementos de cada fila), si se cumple esta condición se escribe una línea por consola marcando la diferencia de línea gráficamente y se escribe el elemento X (cabeza de la lista) entre dos barras (Ej: |1|), tras esto se hace llamada recursiva pasando el cuerpo de la lista (L) para mostrar el siguiente elemento).

Si no se cumple esta condición significa que seguimos en la misma línea pasando a la siguiente llamada de la regla `mostrar_sudoku` en la que simplemente se escribe elemento separado por las dos barras y se hace llamada recursiva con el resto de la lista (cuerpo lista).

Si `mostrar_sudoku` recibe una lista vacía es decir, se han mostrado ya todos los elementos simplemente imprime una línea por pantalla para indicar fin del tablero.

### Búsqueda de las posibilidades

En el caso de este apartado en el que se pide generar una lista de posibilidades para casilla vacía del sudoku (marcada con un '.') también se ha producido un cumplimiento total siendo su grado de cumplimiento del 100%.

Para lograr cumplir esta funcionalidad se han definido las siguientes reglas:

**buscar\_posibilidades** → recibe el Sudoku (lista de 81 elementos) y el hecho donde guardará la lista generada que contenga a su vez la lista de posibilidades para cada casilla vacía (RP). En esta regla se llama `buscar_posibilidades_aux` dónde se produce realmente la lista de posibilidades.

Cuando se termina la ejecución de `buscar_posibilidades_aux` (se ha analizado todo el sudoku) la cual devuelve la nueva lista de listas de posibilidades se da la vuelta a esta lista ya que se ha recorrido el sudoku al revés.

Al terminar esto ya tendríamos el nuevo sudoku siendo este una lista que contiene listas de posibilidades.

**buscar\_posibilidades\_aux** → esta regla recibe una lista vacía (la primera vez), el sudoku resultante que contenga las posibilidades (P), el sudoku original (L) y la posición por la que se inicia búsqueda (N) que en la primera llamada es 1.

En este predicado primero se obtiene el elemento correspondiente a la posición N del sudoku original (L) y lo guarda en el hecho X, tras esto se comprueba si este elemento corresponde a '.' (casilla vacía) ya que si no corresponde simplemente se pasará al siguiente elemento ( $N1 = N + 1$ ) al no

tener que buscar las posibilidades haciendo llamada recursiva de introduciendo elemento X (actual) en el nuevo sudoku(F) y pasando siguiente posición a revisar(N1). Además, destacar que se hará uso del operador de corte ! ya que solo nos interesa la primera rama que se cumpla.

En el caso de que el elemento sea '.' se buscan las posibilidades obteniendo las coordenadas (fila y columna) en las que se encuentra esa posición a través del predicado obtener\_ejes y llamando a la regla posiblesNumeros el cual recibe el sudoku actual (elementos que faltan por revisar de la lista), la fila y columna en la que se encuentra elemento y los posibles valores (OP variable donde se guarda la lista de los posibles valores de esa posición).

Tras esto se lleva a cabo la llamada recursiva introduciendo la lista de posibilidades en el sudoku a devolver y pasando a revisar la siguiente posición ( $N1 = N + 1$ ).

La llamada a esta regla termina cuando el sudoku está vacío (ya se han revisado todas las posiciones) siendo la posición 82 (última posición del sudoku + 1).

**obtener\_ejes** → Se definen listas de columnas y filas que contienen números del 1 al 9 y luego con regla member se comprueba que hecho J1(fila) y J2(columnas) formen parte de las filas y columnas definidas y que juntos formen la posición que buscamos (N).

**posiblesNumeros**→ Primer coge todos los valores posibles de una posición aunque no sean válidos (números del 1 al 9), llama a la regla obtenerFila con el que se consiguen todos los números distintos que aparecen en la fila actual devolviendo estos valores en el hecho valoresFila, obtenerColumna que hace lo mismo pero para la columna actual guardándolos en valoresColumna y valores posibles según el bloque con obtenerBloque en valoresBloque.

Cuando ya tiene todos estos valores los une almacenandolos finalmente en valoresAux los cuales sacamos de la lista de valoresPosibles (contiene números del 1 al 9), los números guardados en valoresAux son aquellos que no son posibilidades válidas ya que ya aparecen en la fila, columna o cuadrante y es por eso que los eliminamos de la lista de posibilidades quedandonos solo con los que no aparecen (posibilidades válidas).

Ej: Teniendo en cuenta el sudoku de ejemplo 1 mostrado anteriormente, si queremos sacar las posibilidades de la posición 1 valoresFila sería [3,2,7] valoresColumna = [5,2,3], valoresBloque = [3, 5] y valoresPosibles =[1,2,3,4,5,6,7,8,9]

Teniendo en cuenta esto valoresAux sería [3,2,7,5] quitamos estos números de valoresPosibles quedandonos con las posibilidades válidas para esa posición **Posibles = [1,4,6,8,9]** que es lo que se guardaría en el nuevo sudoku.

**obtenerFila** → Este predicado se utiliza para obtener todos los valores que aparecen en una fila los cuales se guardan en valoresFila. Se empieza recorriendo la última posición de la fila disminuyendo en uno la posición para ir avanzando a través de obtenerFilaAux.

**obtenerFilaAux** → Cómo se ha indicado anteriormente se recorre una fila a la inversa usando el hecho index y sacando el valor que se encuentra en la posición que éste indica el cual se guardará en la lista de nuevosValores y se pasará a revisar la siguiente posición ( $N - 1$ ) a través de una llamada recursiva. Esta recursividad termina cuando se llega a la última posición definida como -1.

**obtenerColumna** →En este caso se hace lo mismo que en obtenerFila pero con las columnas, guardando los valores encontrados en la lista de valoresColumna tras ejecutar el predicado obtenerColumnaAux.

**obtenerColumnaAux** → Recorre una columna al revés calculando el índice correspondiente y sacando el valor encontrado en su posición guardando a su vez este valor en lista nuevosValores que

se incorporará en la llamada recursiva a la función junto con N -1 hasta llegar a la posición -1 (última columna) terminando recursividad.

**obtenerBloque** → Este predicado es algo más complejo ya que necesitamos saber cuando empieza la fila y la columna correspondiente a ese bloque para lo cual se lleva a cabo el cálculo correspondiente, también se usa la función findall que encuentra todos los valores y between que saca valores dentro de un rango y lo guarda en una variable (DFila y DColumna).

Con estos valores e inicioFila e inicioColumna se calcula el hecho indexBloque con el cual sacamos el valor de esa posición del bloque. Finalmente se guardan todos estos valores en valoresBloque.

## Reglas de simplificación

### **-Simplificar\_sudoku:**

Este predicado llama a buscar\_posibilidades y buscar\_casos el cual aplica las 4 reglas implementadas hasta resolver el sudoku o hasta que ya no se puedan aplicar ninguna de las reglas, esto se comprueba con un contador inicializado a 1 que cambia a 0 cuando se aplica una de las reglas. Cuando tras evaluar las 4 reglas el contador sigue a 1 significa que no se han podido aplicar ninguna terminando el proceso de simplificación.

**-Regla 0:** Esta regla consiste en que si hay un lugar dónde solo cabe un número (solo una posibilidad) se escribe en esa posición y se elimina de la fila, columna y bloque correspondiente.

Para llevar a cabo esta regla se lleva a cabo predicado regla0 que recibe el sudoku actual y devuelve el actualizado con esta regla llamando a regla0Aux que recorre el sudoku al revés (empieza por la última posición).

**regla0Aux** → El primer predicado controla cuando se llega a la última posición (0) terminando recursividad, el segundo saca el valor de esa posición del sudoku y comprueba si pertenece a la lista de ValoresPosibles haciendo a su vez llamada recursiva para revisar la siguiente posición. El tercero saca también el elemento en esa posición y su longitud ya que si longitud es mayor que 1 (tiene varias posibilidades) se pasará a revisar si la siguiente posición ya que no cumple la regla pero si la longitud es 1 ( es un solo número) llama a los predicados sustituir\_elemento que pone ese valor como final en el sudoku y actualizar\_sudoku el cual elimina ese valor de la fila, columna y bloque correspondiente. Por último se hace una llamada recursiva pasando el sudoku actualizado y posición actual (con operador corte para coger solo primera rama que cumpla condición).

**sustituir\_elemento** → Sustituye en posición indicada por Indice el elemento pasado (NuevoElemento al que ya se le han eliminado posibilidades correspondientes según la regla aplicada) a sustituir.

**actualizar\_sudoku** → Este predicado elimina las apariciones del elemento (según su posición N) de la fila, columna y bloque correspondiente tal y como se ha explicado anteriormente, para ello obtiene ejes J1 e J2 y elemento (X) correspondientes a esa posición y llama predicados actualizarFila, actualizarColumna y actualizarCuadrado que van devolviendo sudoku actualizado eliminado apariciones correspondientes hasta llegar a Resultado que contiene sudoku actualizado con elemento eliminado.

**actualizarFila** → En este predicado se elimina elemento de toda su fila, para ello se recorre una fila a la inversa (empezando en posición 9) hasta llegar a la 0 dónde se acaba la recursividad, al recorrer se calcula el Index correspondiente y se saca el elemento en esa posición (Y) comprobando si el que le hemos mandado (X) pertenece usando member ( si Y es igual al elemento que queremos borrar) y borrándolo si pertenece (guarda sudoku con elemento borrado en Borrada) y llama a su vez a

sustituir\_elemento avanzado también la posición y haciendo llamada recursiva. En el caso de que X no pertenezca a Y avanzamos de posición y hacemos llamada recursiva

**actualizarColumna** → Se hace lo mismo que en recorrerFila pero eliminando elemento de la columna correspondiente.

**actualizarCuadrado** → Lo mismo que los anteriores pero para el cuadrante aunque este es algo más complicado ya que necesitaremos dos contadores (contF y contC) para controlar que no nos salemos del rango del cuadrante (3 filas y 3 columnas), también necesitamos calcular inicioFila e inicioColumna y a partir de ellos sacar la posición del elemento que queremos comprobar. El resto del predicado es idéntico a los predicados anteriores borrando y llamando a sustituir\_elemento hasta llegar al final del cuadrante (también se recorre a la inversa).

**-Regla 1:** Esta regla si hay un número que aparece en una sola de las listas que aparecen en una fila, columna o cuadro, cambiamos la lista por el número y borramos el número del resto de listas de la fila, columna o cuadro.

Para realizar esta regla se hace predicado regla1 que llama a regla1Aux para recorrer sudoku a la inversa (desde posición 81).

**regla1Aux**→ Cómo se ha dicho anteriormente recorre el sudoku a la inversa empezando en posición 81 y terminando en la 0 al recursividad. En este predicado primero se obtiene el elemento (X) de la posición a revisar (N) y se comprueba que pertenezca a la lista de valoresPosibles pasando a revisar siguiente posición (N - 1) y haciendo llamada recursiva.

En el siguiente se obtienen los ejes correspondientes a la posición actual (J1 Y J2) y se saca el elemento X correspondiente y su longitud (Tam), para llevar a cabo este predicado se llama a su vez a regla1Fila, regla1Columna y regla1Cuadrante que buscan si hay un número que aparece en una sola de la fila, columna o cuadrante correspondiente a la posición actual llevando a cabo la sustitución y actualización del sudoku dentro de estos predicados, cuando se devuelve sudoku actualizado (ListaAux3) tras ejecutar los predicados anteriores se pasa a revisar la siguiente posición con la llamada recursiva (le pasamos nuevo sudoku).

**regla1Fila** → En este predicado se aplica la regla 1 en la fila correspondiente empezando por atrás (posición 9) y terminando cuando llega al inicio de la fila (posición 0). Primero se calcula el Index correspondiente a la posición actual a analizar(N) y si es igual que la posición del elemento que queremos saber si solo aparece una vez en la fila, es decir, estamos comparando elemento consigo mismo, pasamos a analizar al siguiente elemento (N - 1) haciendo uso de la recursividad. Si Index es distinto y la posición a revisar es 0 (última posición a revisar) sacamos elemento de la posición que indica Tam de array X y se guarda en Y que será el elemento a sustituir ya que significará que no hemos encontrado otro número igual al que analizamos por lo que se cumple la regla, se pasa el elemento al predicado sustituir\_elemento y tras ésto se llama a actualizar\_sudoku.

Si N no es 0, es decir siguen faltando posiciones por revisar, sacamos el elemento de esa posición y lo guardamos en Sig sacando también el elemento Y del array X y comprobamos si es member Y de Sig disminuyendo el valor de Tam (Tam - 1) para analizar siguiente posición (posibilidad) dentro del array X. Por último, si Y no es member de Sig se pasa a analizar siguiente elemento (N-1) haciendo uso de la llamada recursiva.

**regla1Columna**→ Esta regla tiene el mismo funcionamiento que regla1Fila pero recorriendo los elementos de la columna cambiando el valor de Index.

**regla1Cuadrante** → Es parecido a las dos anteriores pero teniendo en cuenta que los límites de columna y fila es de 3 (límites del cuadrante) por lo que hay que usar dos contadores en vez de 1 (N y M) para recorrer cuadrante. Las operaciones son las mismas que anteriormente pero usando

InicioFila e inicioColumna para sacar posición a revisar (equivale a index) y revisar si  $N > 0$  para seguir recorriendo fila o si  $N$  es 0 para movernos por las columnas y cuando  $N$  y  $M$  son 0 (se termina recorrido sin haber encontrado otro elemento igual al que queremos revisar) terminamos sustituyendo el elemento con `sustituir_elemento` y actualizando el sudoku (`acvtualizar_sudoku`).

**-Regla 2:** Si dos números aparecen solos en dos lugares distintos de una fila, columna o cuadro, los borramos del resto de lugares de la fila, columna o cuadro correspondiente.

Para comprobar esta regla se ha llevado a cabo el predicado `regla2` que recibe el sudoku actual y devuelve uno con la regla aplicada. Para ello hace uso de `regla2Aux`.

**regla2Aux** → Con este predicado se recorre el sudoku al revés (empezando por la última posición) buscando elementos que cumplan las condiciones de la regla 2 para realizar sustitución. Primero saca elemento de la posición a revisar ( $N$ ) revisando si es un número ya que entonces no se aplicará la regla por lo que se pasará a revisar siguiente posición ( $N - 1$ ), si no es un número (varias posibilidades) se saca su longitud y si es mayor que dos (más de dos posibilidades en la lista) se pasa a revisar siguiente elemento ya que no cumple la condición de que sean dos números solos. En el caso de que si sean dos números se obtienen los ejes correspondientes a la posición a revisar y se lleva a cabo la llamada a los predicados `regla2Fila`, `regla2Columna` y `regla2Cuadrante` que revisarán si en la fila, columna o cuadrante correspondiente a la posición del elemento a analizar hay otro elemento que tenga los mismos dos números aplicando la regla.

**regla2Fila**→ Este predicado se usa para encontrar el elemento que sea igual al que le pasamos (tengo dos elementos y estos elementos son iguales al pasado) para ello se recorre la fila inversamente a la inversa calculando el Index correspondiente y comprobando si estamos analizando posición del elemento a revisar, si es un número o si la longitud es mayor que dos ya que no se cumpliría condición de la regla por lo que se pasaría a analizar siguiente elemento ( $N-1$ ). Si ninguna de estas condiciones se cumple significa que elemento actual a analizar es una lista de dos elementos en cuyo caso sacamos sus elementos y comprobamos si pertenecen al elemento original pasado, si estos dos elementos son iguales se cumple la regla llamando a `borrarParejaFila` que elimina del resto de posiciones de la fila los números del elemento.

**regla2Columna**→ En este caso se lleva a cabo lo mismo que en la `regla2Fila` pero haciendo la comprobación en el caso de las columnas cambiando el cálculo de Index.

**regla2Cuadrante** → Similar a los anteriores pero recorre el cuadrante según sus límites (3,3) buscando elemento de tamaño 2 igual al pasado para cumplir la regla, para recorrer el sudoku se usarán dos contadores ( $N$  y  $M$ ) que corresponden al número de fila y columna y se calculará la posición haciendo uso de `InicioFila` e `InicioColumna` como se hacía en el resto de reglas. También se comprueba si estamos revisando misma posición que elemento pasado, si es un número o si su longitud es mayor de 2 ya que si se cumple algunos de estos casos no se cumple la regla por lo que debemos pasar al siguiente elemento.

Si estas condiciones no se dan, es decir; elemento comparado es de tamaño dos, se pasa a comprobar si son iguales sacando los elementos y usando `member`, si los elementos son iguales se llama a `borrarParejaCuadrante` que elimina los elementos del cuadrante que sean iguales.

**borrarParejaFila** → Este predicado se usa para borrar todos los elementos de la lista que contengan uno de las dos posibilidades de la pareja de elementos encontradas en `regla2Fila`. Para ello, recorre la fila correspondiente revisando si la posición que estamos analizando actualmente es igual a una de las dos pasadas o si es un solo número (ya está asignado ese número a la casilla) ya que entonces no eliminaremos pasando a revisar siguiente elemento. Si no se cumplen estas condiciones se llama a predicado `borrarGeneral` en el cual se comparan las posibilidades y si coinciden con las de nuestro elemento encontrado los elimina y llama a `sustituir_elemento`.



**borrarParejaColumna** → Se hace lo mismo que en el predicado correspondiente a la fila pero recorriendo columna actual y llamando a predicado borrarGeneral si se cumplen las condiciones de que el elemento actual no es un número ni uno de las parejas encontradas.

**borrarParejaCuadrante** → Igual que los dos anteriores pero recorriendo el cuadrante teniendo en cuenta sus límites igual que en los predicados de cuadrante del resto de reglas usando InicioFila e InicioColumna y llamando a borrarGeneral cuando no se cumplen las condiciones ya mencionadas (no sea un número ni una de las parejas) pasando a revisar siguiente posición tras realizar el borrado correspondiente.

**borrarGeneral** → Como se ha explicado anteriormente este predicado es general para borrar los elementos que aparezcan en nuestra pareja y trío encontrado y en él se saca el elemento de la posición a borrar y el que contiene los números que debemos borrar. Se comprueba si el elemento a borrar ( de la posición N del elemento sacado X) contiene alguno de estos números de Elem y si lo tiene llama a `sustituir_elemento` pasando a revisar el siguiente número con una llamada a recursiva de borrarGeneral

**-Regla 3:** Si en tres lugares de una fila, columna o cuadro solo aparecen tres números distintos, borramos los números de las restantes listas de la fila, columna o cuadro.

Para cumplir esta regla se lleva a cabo predicado `regla3` que llama a `regla3Aux`.

**regla3Aux** → Recorre sudoku al revés comprobando si encuentra un elemento (X) con longitud 3 ya que si no lo hace (si es un número o tiene otra longitud) pasa a revisar el siguiente elemento sin hacer nada, pero si es de longitud 3 obtiene sus ejes y llama a los predicados `regla3Fila`, `regla3Columna` y `regla3Cuadrante` para comprobar si se cumple la regla (hay otros dos elementos iguales) en la fila, columna o cuadrante correspondiente.

**regla3Fila** → En este predicado se busca si hay dos elementos más en la fila indicada iguales al elemento pasado, para ello se calcula el Index correspondiente y se comprueba si el elemento a revisar no es el mismo que el pasado, si no es un número y si su longitud es diferente a 3 ya que entonces no se cumplirá la regla y habrá que pasar a revisar la siguiente posición. Si no se cumplen estas condiciones (elemento de longitud 3) se pasa a comprobar si es igual al pasado posición por posición y si lo es se guarda su posición en el sudoku en la lista `count1` la cual se usará en predicado `borrarFilaTrio` para evitar borrar los elementos encontrados, si los elementos no coinciden se pasa a revisar la siguiente posición.

**regla3Columna** → Este predicado es igual a `regla3Fila` pero recorriendo columna correspondiente, calculando otro Index y llamando a `borrarTrioColumna` cuando encuentra dos elementos iguales al pasado (se cumple regla).

**regla3Cuadrante** → Similar a las anteriores pero usando dos contadores (para fila y columna) para calcular InicioFila e Inicio Columna como en las reglas previas. Las condiciones a revisar son las mismas que en la fila y columna al igual que las instrucciones pero llamando a `borrarTrioCuadrante` cuando se cumple la regla.

**borrarTrioFila** → En este predicado se recorre fila pasada a la inversa buscando qué elementos del sudoku contienen alguno de los números del elemento cuya posición le pasamos. Además, se compruebe que la posición a revisar no sea ninguna de las tres encontradas que cumplen la regla para evitar que se borren a sí mismas ni que sea un número ya establecido en cuyo caso se pasaría a revisar la siguiente posición sin hacer nada. Si estas revisiones no se cumplen se llama a `borrarGeneral`.

**borrarTrioColumna** → Lo mismo que **borrarTrioFila** pero recorriendo la columna.

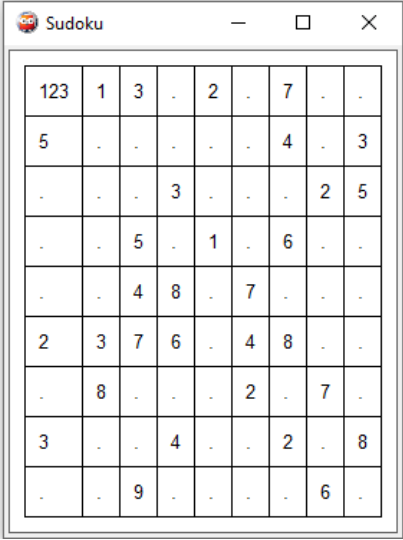
**borrarTrioCuadrante** → Igual que las dos anteriores pero haciendo uso de los dos contadores para no salirse del rango del cuadrante como se ha explicado ya en otras reglas.

## III - Mejoras realizadas

### Implementación de una interfaz

Se ha desarrollado una interfaz visual para mostrar el Sudoku, sus diferentes estados de resolución y su resolución completa. Para ello se han usado las librerías *tabular*, *autowin* y *pce* de Prolog.

Para la representación visual se hace uso del predicado *dibujar\_sudoku*, el cual, recibe como argumento un Sudoku, ya sea resuelto, en su estado inicial o con una lista de candidatos en su interior, este predicado abrirá una ventana de tamaño adaptable que contiene el Sudoku.



123	1	3	.	2	.	7	.	.
5	.	.	.	.	.	4	.	3
.	.	.	3	.	.	.	2	5
.	.	5	.	1	.	6	.	.
.	.	4	8	.	7	.	.	.
2	3	7	6	.	4	8	.	.
.	8	.	.	.	2	.	7	.
3	.	.	4	.	.	2	.	8
.	.	9	.	.	.	.	6	.

Se hace uso de un “auto\_sized\_picture” que es la ventana que contendrá la información del Sudoku, a esta ventana se añade una tabla de la librería *tabular*, que contiene cada número o lista de números de cada celda.

Para poder añadir los números a cada celda hay que recorrer el Sudoku original y crear una lista nueva que sustituya cada número por “append(numero)” y cada 9 números, que incluya un “nex\_row” salto de línea, esto lo lleva a cabo el predicado “recorrer\_sudoku\_visual” y devuelve la lista deseada. La lista resultante hay que formatearla, puesto que está compuesta por cadenas de caracteres (representadas por comillas dobles) y no átomos, como es necesario, para ello se hace uso del predicado “cadenas\_a\_terminos”, que convierte una lista de cadenas en una lista de átomos y devuelve como resultado la lista deseada.

Una vez realizados estos cambios al Sudoku original, ya se puede enviar la lista a la tabla de la ventana para ser representada.

Cabe recalcar también, que es posible que una celda incluya una lista de números (candidatos posibles), para que esto pueda ser representable se hace uso del predicado “lista\_a\_numero”, que transforma una lista de números en un único número (Ej: [1,2,3] -> 123).

## Estudio de sobre el rendimiento de las reglas

Para realizar un estudio sobre el de las reglas para la resolución de los Sudokus se ha creado un banco de sudokus con 300 ejemplos (100 Sudokus fáciles, 100 medios y 100 difíciles).

Fácil = 65 casillas con solución.

Medio = 50 casillas con solución.

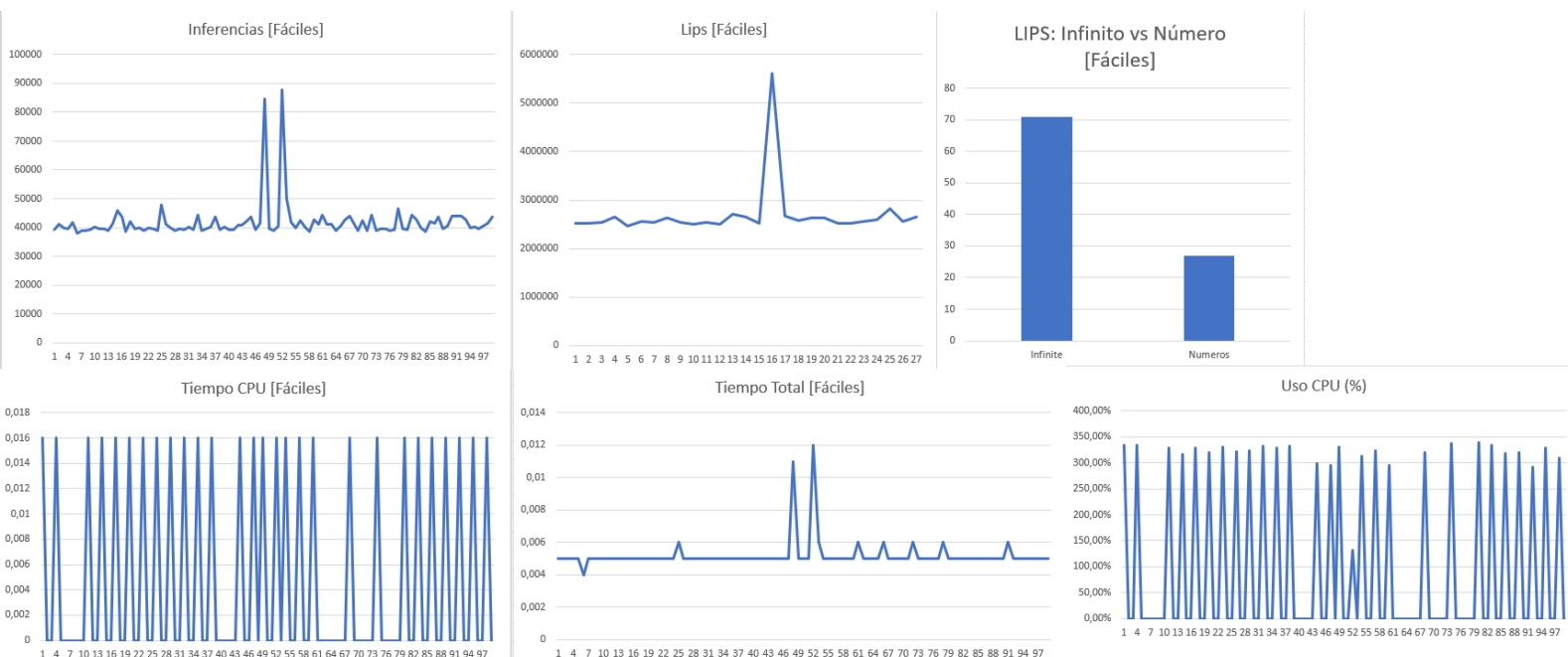
Difícil = 35 casillas con solución.

Se ha usado el predicado “time/1” de Prolog para obtener las estadísticas de la resolución de los Sudokus, a continuación se van a definir los términos que aparecen:

- Inferencias: Número de inferencias realizadas por el predicado durante la ejecución del time/1. Una inferencia es el proceso de aplicar una regla para deducir una conclusión.
- Tiempo CPU: Tiempo específico de CPU utilizado por el predicado.
- Tiempo Total: Tiempo total empleado por el predicado.
- Uso CPU (%): Porcentaje de uso de la CPU. Es posible que se muestre más del 100%, esto se da en ocasiones cuando la CPU donde se ejecuta el test tiene varios núcleos y hace uso de ellos para calcular el predicado.
- Lips: “Logical Inferences per Second”, se refiere a la cantidad de inferencias que Prolog es capaz de realizar en un segundo al realizar el test. Un mayor número indica una mejor eficiencia, el mejor resultado obtenible es “infinito”.

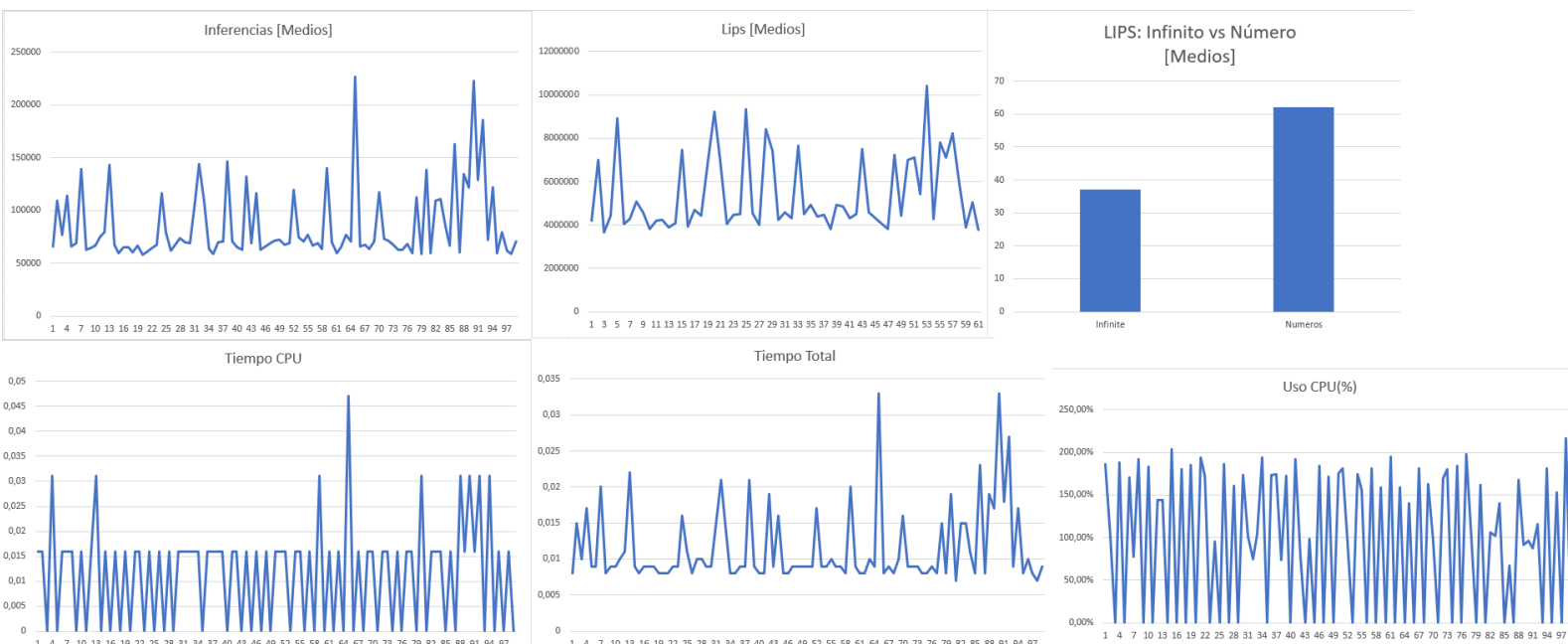
Inferencias	Tiempo CPU	Tiempo Total	Uso CPU (%)	Lips	Inferencias	Tiempo CPU	Tiempo Total	Uso CPU(%)	Lips	Inferencias	Tiempo CPU	Tiempo Total	Uso CPU(%)	Lips
39309	0,016	0,005	334,00%	2515776	65639	0,016	0,008	186,00%	4200896	416459	0,062	0,062	100,00%	6663344
41020	0	0,005	0,00%	Infinite	109347	0,016	0,015	102,00%	6998208	308413	0,047	0,045	104,00%	6579477
39934	0	0,005	0,00%	Infinite	76607	0	0,01	0,00%	Infinite	367331	0,047	0,055	85,00%	7836395
39405	0,016	0,005	334,00%	2521920	114136	0,031	0,017	188,00%	3652352	220704	0,031	0,032	97,00%	7062528
41833	0	0,005	0,00%	Infinite	65700	0	0,009	0,00%	Infinite	376763	0,047	0,056	84,00%	8037611
37862	0	0,004	0,00%	Infinite	69154	0,016	0,009	170,00%	4425856	185017	0,031	0,027	117,00%	5920544
38750	0	0,005	0,00%	Infinite	139152	0,016	0,02	78,00%	8905728	438328	0,062	0,064	98,00%	7013248
38856	0	0,005	0,00%	Infinite	63028	0,016	0,008	192,00%	4033792	177848	0,031	0,025	123,00%	5691136
39082	0	0,005	0,00%	Infinite	64581	0	0,009	0,00%	Infinite	336350	0,047	0,049	95,00%	7175467
40252	0	0,005	0,00%	Infinite	66936	0,016	0,009	183,00%	4283904	363485	0,047	0,055	86,00%	7754347
39600	0,016	0,005	329,00%	2534400	75096	0	0,01	0,00%	Infinite	415437	0,062	0,062	101,00%	6646992
39422	0	0,005	0,00%	Infinite	79560	0,016	0,011	144,00%	5091840	157489	0,031	0,022	143,00%	5039648
38843	0	0,005	0,00%	Infinite	143100	0,031	0,022	144,00%	4579200	500596	0,078	0,075	104,00%	6407629
41317	0,016	0,005	317,00%	2644288	67066	0	0,009	0,00%	Infinite	397480	0,047	0,059	80,00%	8479573
45809	0	0,005	0,00%	Infinite	59409	0,016	0,008	204,00%	3802176	304826	0,047	0,044	105,00%	6502955
43636	0	0,005	0,00%	Infinite	65346	0	0,009	0,00%	Infinite	405945	0,062	0,06	104,00%	6495120
38510	0,016	0,005	329,00%	2464640	65335	0,016	0,009	180,00%	4181440	498590	0,078	0,074	105,00%	6381952
41972	0	0,005	0,00%	Infinite	60689	0	0,009	0,00%	Infinite	415919	0,062	0,061	103,00%	6654704
39387	0	0,005	0,00%	Infinite	66415	0,016	0,008	185,00%	4250560	365221	0,047	0,054	87,00%	7791381

## Estadísticas sobre los sudokus de dificultad fácil:



Como se puede observar el número de inferencias realizadas sobre los sudokus fáciles sigue una media de 41.807, excepto en la ejecución de dos Sudokus puntuales, que tal vez su dificultad se vió incrementada por otros factores e hizo falta realizar más inferencias para poder resolverlo. En relación a los Lips, estos siguen una media de 2.692.762, esto es un buen factor, ya que indica que la dificultad para obtener una solución al predicado fue muy fácil, otro dato a tener en cuenta es que se han obtenido 71 valores “infinito” en relación a los Lips, esto es otro muy buen indicador, que vuelve a demostrar la facilidad de resolución de los Sudokus. Respecto al tiempo de CPU y tiempo total, observamos que los dos tienen una media de 0,005 segundos, si se hace más hincapié en los resultados del tiempo de CPU se observa que muchos de estos resultados son 0 o 0,016 segundos, esto es lo que hace que la media se trunque hasta 0,05 segundos, cuando un resultado de CPU devuelve 0 segundos significa que el tiempo de CPU empleado para la resolución del predicado ha sido ínfimo, en resumen, los tiempos totales para resolver los Sudokus ha sido muy poco, ya que se trata de sudokus fáciles. Finalmente, respecto al porcentaje de uso de CPU, ocurre la misma situación que con el tiempo de CPU, hay resultados en un rango de 0% a 300%, esto es lo que hace que la media se trunque a 89,04% de uso, ese rango de valores se debe a que en ciertas resoluciones de Sudokus (correspondientes a la que constan de un mayor tiempo de CPU) ha resultado más complicado llegar a la resolución de los predicados.

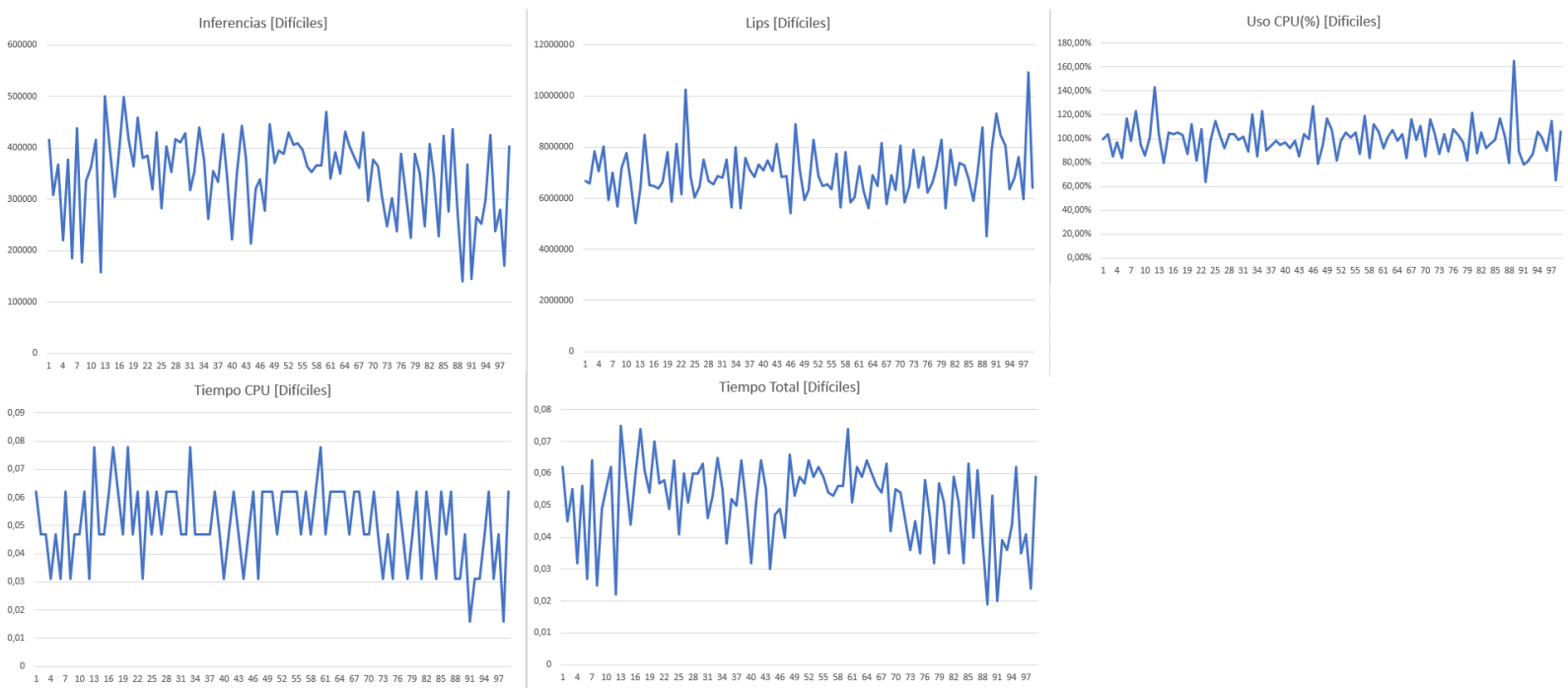
### Estadísticas sobre los sudokus de dificultad media:



Como se puede observar, el número de inferencias medias ha sido de 85.336 inferencias, el doble de inferencias respecto a los Sudokus fáciles, esto demuestra que aproximadamente ha sido el doble de difícil llegar a la resolución de los Sudokus de nivel medio, al igual que con los sudokus fáciles, hay cierto conjunto de Sudokus que requieren un mayor número de inferencias para su resolución, debido otra vez a factores externos que pudieran dificultar su resolución. En relación a los Lips, obtenemos una media de 5.415.930, casi el doble respecto a los Sudokus fáciles, la respuesta a este dato es que su resultado ha sido mayor porque ha disminuido el resultado de “infinites” obtenidos (que es el mejor resultado), por eso la media se ha visto afectada, por ende, los Sudokus fáciles siguen siendo de resolución más fácil. En relación a los tiempos de CPU y totales, otra vez se obtiene una media igual, resultante 0,012 segundos, aproximadamente el doble en contraposición a los Sudokus de dificultad fácil, pero sigue siendo un tiempo inapreciable para llegar a la solución de los predicados. Por último, el porcentaje de CPU utilizado ha tenido una media de 93,93%, otra vez con valores entre

un rango de 0% y 200%, lo que trunca la media, esto es debido a que ciertos Sudokus han sido muy simples de resolver.

Estadísticas sobre los sudokus de dificultad difícil:

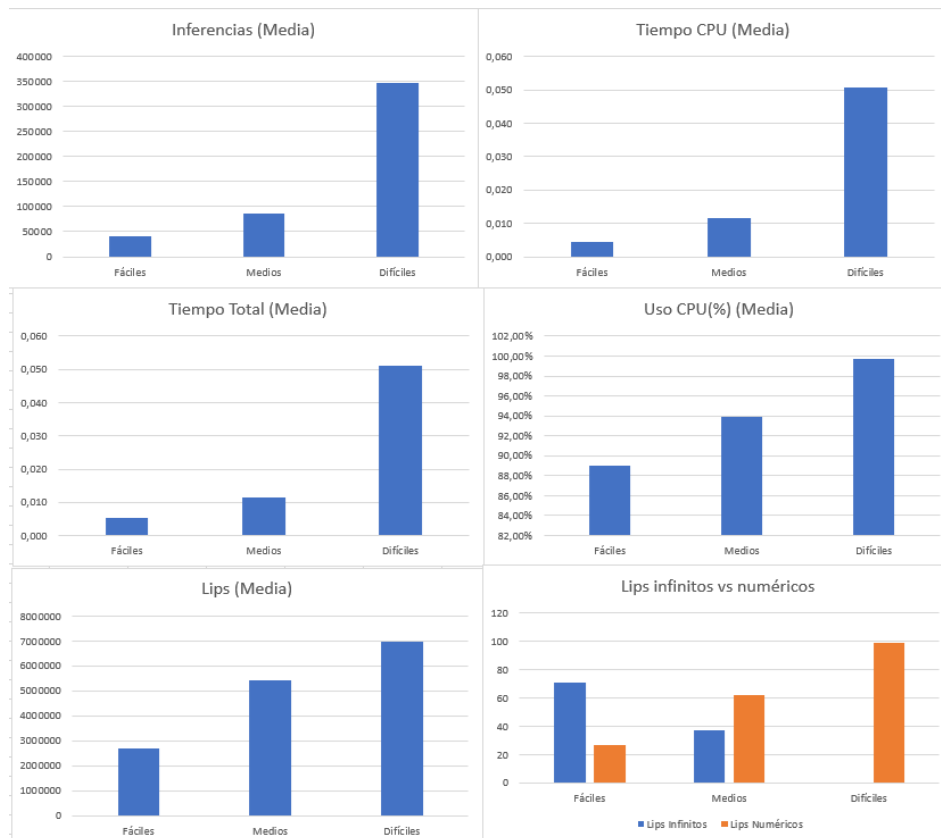


Esta vez como dato más resaltante, no se han obtenido Lips con valor “infinite”, lo que demuestra que la complejidad para resolver los Sudokus de dificultad difícil se ha visto bastante incrementada. La media de inferencias ha subido a 346.916, un 400% más de inferencias respecto a los Sudokus de dificultad media, este resultado demuestra que ha sido necesario aplicar 346.000 predicados hasta llegar a la resolución del Sudoku (con o sin solución), viéndose truncada la eficiencia de nuestro algoritmo. El tiempo de CPU y total siguen una media de 0,051 segundos, este dato resalta que a pesar de que la eficiencia de nuestro algoritmo fue menor, el tiempo de resolución por Sudoku sigue siendo ínfimo y no representa ningún problema para nuestro diseño. La media de Lips que se pueden ejecutar es 6.974.334, otra vez un dato mayor, pero esto se ve mermado por la inexistencia de valores “infinite”. Por último, el porcentaje de CPU utilizado ya no tiene ningún valor de 0%, todos siguen una media de 99,71% para los 100 test, demostrando una vez más que ha sido más difícil solucionar los Sudokus.

Comparación entre dificultades:

Inferencias	Tiempo CPU	Tiempo Total	Uso CPU(%)	Lips	Lips Infinitos	Lips Numéricos	Dificultad
41807	0,005	0,005	89,04%	2692762	71	27	Fáciles
85366	0,012	0,012	93,93%	5415930	37	62	Medios
346916	0,051	0,051	99,71%	6974334	0	99	Difíciles

Como se puede observar, el incremento de dificultad entre los diferentes test reflejan los valores esperados. Relativo a la resolución de Sudokus de gran dificultad, nuestro algoritmo diseñado sigue siendo muy eficiente para resolver los Sudokus y no representan una gran carga de CPU.



## Estudio de sobre el grado de cumplimiento de resolución

Respecto al grado de cumplimiento de resolución en los 300 Sudokus, se han obtenido los siguientes resultados:

- Fáciles: 98% de resolución satisfactoria.
  - 48 y 52 no resueltos.
- Medios: 74% de resolución satisfactoria.
  - 2, 4, 7, 13, 24, 31, 32, 33, 38, 42, 44, 52, 59, 65, 70, 78, 80, 82, 83, 86, 88, 89, 90, 91, 92 y 94 no resueltos.
- Difíciles: 2% de resolución satisfactoria.
  - 89 y 91 resueltos.

En relación al cumplimiento de los sudokus fáciles, solamente la lógica implantada no ha sido capaz de resolver 2 instancias, estas tienen relación al pico de inferencias mostrados en las estadísticas recogidas, por lo tanto se puede predecir que sudokus van a ser resueltos en base a las inferencias. Respecto a los Sudokus medios, la lógica ha sido capaz de resolver un 74% de los casos, esta vez también guarda referencia la resolución satisfactoria del Sudoku con el número de inferencias realizadas de media, según la gráfica de las estadísticas anteriores, los picos de inferencias se corresponden con los Sudokus no resueltos.

Respecto a los sudokus difíciles, sólo se han dado 2 ocasiones donde han sido resueltos, dando un 2% de resolución satisfactoria, la lógica implantada no ha sido capaz de resolver los Sudokus de dificultad difícil.

Analizando los Sudokus no resueltos, estos presentan un gran grado de desorden, es decir, las listas de candidatos que tiene cada elemento del sudoku es demasiado grande para sólo ser resuelta con una regla nueva, habría que implantar nuevas reglas correspondientes a "X-Wing", "Y-Wing", "Pez Espada", "Trios/Parejas apuntadoras" y "Trios/Parejas ocultas". Para poder resolver los Sudokus que no han sido resueltos, habría que desarrollar una lógica nueva aún mayor que la implantada.



## Búsqueda y estudio de un algoritmo sobre Sudokus predefinido

SWI-Prolog cuenta en su página web con un código que resuelve sudokus, este solo ocupa 15 líneas: <https://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku>

```
1 sudoku(Rows) :-
2     length(Rows, 9), maplist(same_length(Rows), Rows),
3     append(Rows, Vs), Vs ins 1..9,
4     maplist(all_distinct, Rows),
5     transpose(Rows, Columns),
6     maplist(all_distinct, Columns),
7     Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
8     blocks(As, Bs, Cs),
9     blocks(Ds, Es, Fs),
10    blocks(Gs, Hs, Is).
11
12 blocks([], [], []).
13 blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
14     all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
15     blocks(Ns1, Ns2, Ns3).
```

Este es capaz de resolver el banco de 300 sudokus realizado con el mismo porcentaje de resolución satisfactoria que nuestro algoritmo de 772 líneas de código. Basa su algoritmo en una librería de programación lógica sobre dominios finitos y el predicado “all\_distinct”, que detecta si todas las variables pueden tomar los distintos valores dado un dominio.

## Comparación entre algoritmo desarrollado y predefinido

Comparando los dos algoritmos, nuestro algoritmo desarrollado ocupa un 5196% más de espacio en comparación del algoritmo de ejemplo de Prolog(772 líneas en comparación a 15), pero en un análisis más profundo, sobre un mismo sudoku del banco, el algoritmo de Prolog realiza un 2000% más de inferencias que nuestro algoritmo, además de tomar un 1200% más de tiempo de CPU. Como ambos algoritmos son capaces de resolver los 300 Sudokus con el mismo grado de solución, determinamos que en términos de rendimiento nuestro algoritmo es mejor que el desarrollado por Prolog, a pesar de ocupar más líneas de código.

Proyecto CRA (Sudoku\_facil43):

*% 40,854 inferences, 0.000 CPU in 0.005 seconds (0% CPU, Infinite Lips)*

Prolog (Sudoku\_facil43):

*% 700,212 inferences, 0.062 CPU in 0.079 seconds (79% CPU, 11203392 Lips)*

## IV - Experiencia con Inteligencia Artificial

Este apartado contiene una síntesis sobre la experiencia obtenida en relación al uso de IA (Inteligencia Artificial) como Chat GPT-3 y Prolog.

Chat GPT-3 es una herramienta muy potente en la actualidad, que ayuda a una gran cantidad de personas en diversos ámbitos, ya sea programación, matemática, lógica... para lograr una sesión más eficiente de trabajo y servir de soporte.

En relación a Chat GPT-3 y Prolog, la experiencia obtenida con la IA ha sido de gran ayuda, pero hay que establecer una definición sobre, ¿cuál ha sido la ayuda recibida?.

La ayuda obtenida gracias a la IA no ha sido en grado de cumplimentación, es decir, introducir un predicado deseado y esperar que la IA te lo programe correctamente no es eficiente ni útil, esto se debe a que Chat GPT-3 no puede compilar código ni comprobar resultados de ejecución por

seguridad. Al intentar pedir a la IA que te escriba un predicado específico que sirva para el trabajo, este está lleno de errores y no suele ser adaptable al código original del proyecto.

Por el contrario, la IA ha sido de gran soporte para realizar una mejor búsqueda y más rápida sobre los bancos de datos de internet y ofrecer una solución aproximada, que luego el alumno/programador ha de discriminar y adaptar a su trabajo. Aquí reside la mayor complejidad y dificultad de usar Chat GPT-3, que es el saber discriminar correctamente la información que te aporta la IA. Como se ha comentado anteriormente, los predicados específicos que se pide completar a la IA normalmente son erróneos, pero a pesar de que estos puedan ser erróneos, estos pueden servir de gran ayuda para orientar y ofrecer ideas de diferentes alternativas al programador. El gran apoyo de la IA sirve en relación a preguntas simples y sencillas, que facilitan al programador no tener que buscar guías o manuales por internet para encontrar la respuesta, un ejemplo de pregunta sencilla es: ¿Cómo puedo invertir una lista?, la respuesta es "Usando el método reverse()", esta respuesta es obtenida en menos de un segundo y optimiza la calidad de trabajo del programador.

Si Chat GPT-3 es una herramienta tan potente, ¿por qué no ha sido de tanta ayuda con Prolog?, esto se debe a que Prolog es un lenguaje minoritario en comparación a Python/Java/C++, y por ende, no cuenta con tantas referencias en internet de las que pueda hacer uso la IA, de hecho, la mayoría de librerías de Prolog están descontinuadas, no ofrecen soporte y no existe un manual en internet sobre su uso, por lo tanto se dificulta mucho el uso de Prolog por parte del programador.

En resumen, Chat GPT-3 es de gran ayuda para Prolog con preguntas sencillas, pero no es óptimo ni útil para programar predicados complejos ya que no cuenta con muchas referencias. Se recomienda su uso para mejorar la eficiencia de trabajo en relación a preguntas sencillas, pero hay que tener en cuenta que la IA no es capaz de resolver problemas complejos en Prolog. Intentar resolver problemas complejos con la IA y Prolog solo conlleva una pérdida de tiempo.

Cabe destacar que no se ha hecho uso de Chat GPT-3 para el desarrollo de las reglas.

## V - Errores

No hay ningún error en la ejecución y se cumplen todos los requisitos especificados en el enunciado.

## VI - Fuentes utilizadas

<https://www.swi-prolog.org/download/xpce/doc/userguide/userguide.pdf>

<http://oefa.blogspot.com/2008/04/programacion-logica-swi-prolog.html>

<https://sudoku.com/es/reglas-del-sudoku>

<https://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku>