Fundamentos de Inteligencia Artificial

Práctica 2

Sudoku

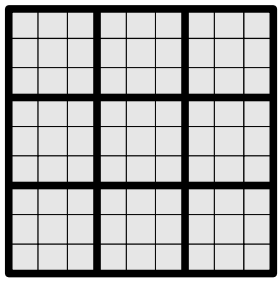
Autor	Cristian Aguilera Martínez
Turno	jueves 11:30 – 13:00
Profesor	Fidel Aznar Gregori
Fecha de entrega	29 de noviembre de 2009

Índice

1. Introducción al juego del Sudoku

1.1. Definición y reglas

El Sudoku es un rompecabezas matemático del que se empezó a hablar en 1986 y se dio a conocer internacionalmente en 2005. Tiene el aspecto de una cuadrícula de crucigrama de 9x9 con sus 81 celdas agrupadas en nueve cuadrados interiores de dimensiones 3x3.



Las reglas del Sudoku son simples:

- No se debe repetir ninguna cifra en una misma fila.
- No se debe repetir ninguna cifra en una misma columna.
- No se debe repetir ninguna cifra en un mismo bloque interior.

Aunque un sudoku está bien planteado si la solución es única, normalmente se ignora este detalle, pudiendo existir múltiples soluciones distintas.

Durante la construcción de un sudoku, existen 5 estados en los que puede encontrarse dicho sudoku:

- *Vacío*: no se ha establecido ningún valor.
- *Incompleto*: tiene algunas casillas con valor asignado, pero no todas.
- *Erróneo*: hay alguna restricción que no se cumple.
- *Sin solución*: es un tablero incompleto en el que, aunque se cumplen todas las restricciones, no podrá completarse.
- *Correcto*: el tablero está completo y cumple todas las restricciones.

1.2. Resolución por ordenador

Para los programadores es relativamente sencillo construir una búsqueda por el método de backtracking o "vuelta atrás". Ésta asignaría, típicamente, un valor (supongamos que 1, o el más cercano a 1 disponible) a la primera celda disponible (supongamos que la superior izquierda) y entonces continuará asignando el siguiente valor disponible (supongamos que 2) a la siguiente celda disponible. Esto continuaría hasta que se descubriera una duplicación, en cuyo caso, el siguiente valor alternativo se colocaría en el primer campo alterado. En el caso de que ningún valor cumpliera la restricción se retrocedería hasta la casilla anterior y se probarían los siguientes números.

Aunque lejos de la eficiencia computacional, este método encontrará la solución si se permite el suficiente tiempo de computación. Un programa más eficiente podría dejar una huella de valores potenciales para las celdas, eliminando valores imposibles hasta que sólo un valor quedase para una celda determinada. Entonces se rellenaría esa celda y se usaría esa información para más eliminaciones y así, sucesivamente, hasta el final. Este método es más parecido al método que una persona utiliza para resolver un sudoku.

2. El Sudoku como problema de satisfacción de restricciones

2.1. Qué es un problema de satisfacción de restricciones

Un problema de satisfacción de restricciones viene definido por los siguientes elementos:

- Un conjunto finito de variables V.
- Un dominio D que contiene los valores que pueden tomar las variables.
- Un conjunto finito de restricciones C que definen una serie de propiedades que debe verificar los valores asignados a las variables .

Para que el problema se considere resuelto, hay que asignar un valor del dominio a cada una de las variables de manera que se verifiquen todas las restricciones

Esta formulación permite una representación simple del problema y el uso de algoritmos de propósito general independientes del problema.

2.2. Formalización del problema

A continuación formalizaremos el juego del Sudoku como un problema de satisfacción de restricciones.

Tal como se ha visto, un sudoku es un tablero de 9x9 casillas –81 en total–. Cada una de estas casillas se corresponde con una variable del problema de satisfacción de restricciones.

$$V = V_{ij} \ , \ \forall i,j \in \{1,2,3,...,9\}$$

Los índices de cada variable se corresponden con el número de fila y el número de columna respectivamente. Por lo tanto, realizando el producto cartesiano de los índices de cada fila y columna, obtenemos:

$$V = \{V_{11}\ ,\ V_{12}\ ,\ V_{13}\ ,\ V_{14}\ ,\ \dots\ ,\ V_{98}\ ,\ V_{99}\}$$

Cada casilla del tablero puede –y debe– contener un valor entre 1 y 9. Esto significa que el dominio del problema tiene una cardinalidad de 9; los 9 primeros números naturales.

$$D = \{1\ ,\ 2\ ,\ 3\ ,\ 4\ ,\ 5\ ,\ 6\ ,\ 7\ ,\ 8\ ,\ 9\}$$

Sin embargo, el tablero puede contener una serie de casillas con un valor predefinido que no puede modificarse. Eso restringe el dominio de esas casillas predefinidas a un único valor, que coincide con el valor predefinido. Al establecer esos valores predefinidos, se considera que cada variable tiene un dominio propio no compartido con el resto de variables. Por lo tanto, existirá un conjunto de 81 dominios –uno por cada variable–, y cada uno de ellos será un subconjunto del dominio D.

$$D' = \{D_{11}\ ,\ D_{12}\ ,\ D_{13}\ ,\ D_{14}\ ,\ \dots\ ,\ D_{98}\ ,\ D_{99}\}\ /\ D_{ij} \subseteq D,\ \forall i,j \in \{1,2,3,...,9\}$$

Tal como se vio en el primer apartado, existen 3 tipos de restricciones en el juego del Sudoku:

- Restricción de fila: dos casillas de la misma fila no pueden tener el mismo valor asignado.

$$\neg \exists k \in D\ /\ V_{ij} = V_{ik}\ \wedge\ j \neq k\ ,\ \forall i,j \in D$$

- Restricción de columna: dos casillas de la misma columna no pueden tener el mismo valor asignado.

$$\neg \exists k \in D \ / \ V_{ij} = V_{kj} \ \wedge \ i \neq k \ , \forall i, j \in D$$

•Restricción de bloque: dos casillas del mismo bloque no pueden tener el mismo valor asignado.

$$\neg \exists k, k' \in D \ / \ V_{ij} = V_{kk'} \ \wedge \ mismo\_bloque(V_{ij}, V_{kk'}) \ , \forall i, j \in D$$

$$mismo\_bloque(V_{ij}, V_{kk'}) \equiv (i \text{ div } 3 = k \text{ div } 3) \wedge (j \text{ div } 3 = k' \text{ div } 3)$$

Se definen en total 1944 restricciones dentro del conjunto de restricciones C en las que únicamente intervienen parejas de variables –relaciones binarias–. Cada una de las 81 variables tiene una restricción con las 8 variables restantes de su fila, otra restricción con las 8 variables restantes de su columna, y otra restricción con las 8 variables restantes de su bloque. Es decir, cada variable tiene un total de 24 restricciones con otras variables. Por lo tanto, 81·24 = 1944 restricciones en el problema.[1] Todas las restricciones binarias que relacionan pares de variables, impiden que esas 2 variables que intervienen en la restricción tengan el mismo valor.

3. Implementación de la solución

3.1. Algoritmo Forward Checking

El algoritmo Forward Checking resuelve problemas de satisfacción de restricciones de forma genérica con restricciones binarias y un dominio finito. Cualquier problema bien formalizado, puede resolverse con este algoritmo.

Forward Checking es un algoritmo de backtracking que reduce el dominio de las variables después de cada asignación de valor a otra variable. Si en algún momento una variable se queda sin ningún elemento en su dominio, hay que dar marcha atrás y restaurar aquellos valores que fueron eliminados durante la reducción del dominio de las variables.

El algoritmo Forward Checking trabaja sobre el conjunto de variables V y sobre el conjunto de restricciones C. Cada variable del conjunto V tiene asignado un dominio de elementos factible con los posibles valores que puede recibir la variable y un dominio de elementos podados que está ini-

1 Aunque cada variable tiene una restricción con el resto de las 8 variables de su bloque, 4 de esas variables ya se han contabilizado junto a las restricciones de fila y de columna, por lo que sería más eficiente establecer sólo 1620 restricciones. Es decir, cada variable establece 20 restricciones y no 24 –81·20 = 1620–.

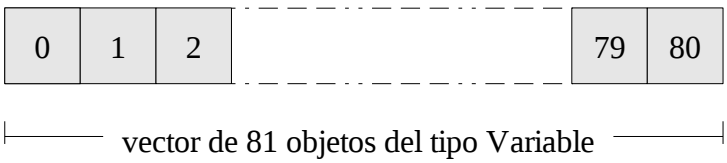
cialmente vacío.

Para organizar las variables se ha construido una clase Variable que proporciona los atributos necesarios a cada variable, que serán utilizados por el algoritmo. Cada variable tiene un valor –que habrá que asignar durante la ejecución–, un conjunto de elementos que se corresponde con su dominio factible y otro conjunto de elementos que se corresponde con los elementos del dominio que han sido temporalmente descartados por el algoritmo. Además, en este segundo conjunto se almacena qué variable fue la causante de este descarte o poda.

Variable
- Integer valor - ArrayList<Integer> factibles - HashMap<Integer, Integer> podados
+ Integer getValor() + void setValor(Integer valor) + Integer getNumFactibles() + Integer getFactible(Integer indice) + boolean anadirFactible(Integer factible) + boolean eliminarFactible(Integer factible) + boolean podarFactible(Integer factible, Integer variable) + boolean restaurarPodado(Integer variable)

El método podarFactible traslada un elemento del dominio factible a los elementos podados. El método restaurarPodado deshace todas las podas que se han llevado a cabo por culpa de la variable que se indique en los parámetros.

Se ha utilizado un vector del tipo Variable de 81 posiciones que representa las 81 variables. Este vector es una variable global que podrá accederse desde cualquier función.



La implementación del algoritmo Forward Checking se basa en 3 funciones que se describirán en los apartados siguientes.

3.1.1. Función FC

Ésta es la función principal del algoritmo y se trata de una función recursiva. Al ejecutarla, se intenta asignar un valor a cada variable del problema de manera que se cumplan las restricciones. Si no fuera posible realizar esta asignación, la función devolverá falso, indicando que no existe ninguna solución.

La primera llamada a esta función siempre comienza con la variable 0. A partir de ahí se irán llamando recursivamente el resto de variables hasta la variable 80, que es la última.

```
función FC(i variable): booleano

para cada a ∈ factibles de Vi hacer
    Vi = a;
    si i es 80 entonces
        retorna CIERTO;
    sino
        si forward(i, a) entonces
            si FC(i + 1) entonces
                retorna CIERTO;
        restaurar(i);
retorna FALSO;
```

El algoritmo intentará asignar el primer valor que aparezca en el dominio factible de la variable que está evaluando. Al asignarlo, eliminará –podará– ese valor de todas las variables con las que tiene restricción y se aplicará la recursividad sobre la siguiente variable. Si la llamada recursiva devolviera falso, significa que hay que restaurar los valores podados y elegir otro valor del dominio siempre y cuando todavía queden valores no explorados en el dominio factible.

3.1.2. Función forward

La función forward se encarga de podar valores desde el dominio de las variables entre las que existe restricción. Recibe dos parámetros: el valor del dominio que se quiere podar y la variable responsable de la poda.

```
función forward(i variable, a valor): booleano

para j=i+1 hasta 80 hacer
    si i y j están relacionadas por una restricción entonces
        podar a en factibles de Vj siendo Vi culpable de la poda
        si factibles de Vj está vacío entonces
            retorna FALSO;
retorna CIERTO ;
```

Todas las variables que sucedan a la variable que va a realizar la poda van a ser examinadas. Si existe relación de restricción entre ambas variables, se eliminará el valor indicado en los parámetros de su dominio factible y, si el conjunto de elementos del dominio factible se queda vacío, abortamos la función devolviendo falso. Sólo se devuelve verdadero si es capaz de podar el valor en todas las variables sin que el dominio quede vacío.

3.1.3. Función restaurar

Esta función se encarga de trasladar los elementos podados al dominio factible. Cuando se devuelve un falso en la función forward o cuando falla la llamada recursiva FC, hay que restaurar los elementos que fueron podados por la variable que está actualmente en ejecución dentro de FC.

Se recorrerán todas las variables y las que tengan elementos podados por la variable se trasladarán a su dominio factible.

```
procedimiento restaurar(i variable)

para j=i+1 hasta 80 hacer
    para todo b ∈ podados de Vj hacer
        si Vi es el responsable de la poda b entonces
            eliminar b de podados de Vj
            añadir b a factibles de Vj
```

3.2. Algoritmo AC3

Este algoritmo pretende eliminar la inconsistencia entre variables y restricciones, acelerando Forward Checking en la mayoría de casos.

Arista
+ Integer primera + Integer segunda
+ Arista(Integer primera, Integer segunda) + ArrayList<Arista> generarAristas(Integer variable, boolean primera): static + boolean equals(Arista arista) + int hashCode()

Se ha construido una clase Arista que relaciona 2 variables. Esta clase sólo almacena 2 valores enteros que pueden tomar valores entre 0 y 80, que hacen referencia a las posiciones de las variables del vector comentado anteriormente.

Se sobreescribe el método equals para decidir si dos objeto Arista son esencialmente el mismo objeto. Es decir, dos aristas son iguales si contienen los mismos valores enteros.

Otro de los métodos es generarAristas. Éste es un método estático que genera las aristas en las que interviene una determinada variable recibida en los parámetros. También recibe otro parámetro booleano que indica en qué parte de la arista debe intervenir: en la primera o en la segunda.

```
función AC3(): booleano

Q = generar todas las aristas iniciales;
mientras Q ≠ Ø hacer
    <Vk, Vm> = extraer la primera arista desde Q;
    para todo b ∈ factibles de Vk hacer
        si b es el único valor de factibles de Vm entonces
            borrar b de factibles de Vk
    si no quedan elementos en los factibles de Vk entonces
        retorna FALSO;
    si se borró algún elemento de los factibles de Vk entonces
        reinsertar todas las aristas <-,Vk> en Q;
retorna CIERTO;
```

Por cada variable, se tienen que generar 24 aristas tal y como se comentó en el apartado 2 –8 aristas para relacionar cada variable con el resto de variables de su fila, otras 8 para las columnas y otras 8 para las variables dentro del mismo bloque–. Por lo tanto, inicialmente se tiene una lista de 1944 aristas –81·24 = 1944–. El bucle se repetirá mientras queden aristas pendientes de examinar.

En cada iteración del bucle se extrae una de las aristas y se comprobará que es consistente. Una arista es consistente si, para cualquier valor del dominio de la primera variable que se le asigne, existe otro valor en el dominio de la segunda variable que se le puede asignar. Cuando aparece una arista inconsistente, hay que eliminar los valores que generan la inconsistencia. Sin embargo, es posible que al eliminarlos el dominio de alguna variable queda vacío, lo que supondría un problema sin solución.

Es decir, con esa arista extraída se comprueba que:

- de todos los elementos del dominio de la primera variable de la arista, no hay ningún valor que, al asignárselo a dicha primera variable, la segunda variable se queda sin dominio factible
- si eso ocurre, se elimina ese valor del dominio factible de la primera variable y se reinsertan las aristas en las que aparezca la primera variable como segunda variable

Por ejemplo, si se está examinando la arista <3,6> y la variable 6 sólo puede tener el valor 8, ese valor 8 hay que eliminarlo del dominio factible de la variable 3 para que FC no lo asigne en ningún caso, ya que no se puede obtener una solución si se asigna; asignarlo implicaría tener que retroceder hasta ese punto para asignarle otro valor diferente. Al eliminar ese valor 8, es necesario reinsertar en la lista todas las aristas en las que interviene la variable 3 en segundo lugar; es decir, todas las aristas de la forma <-, 3>. El dominio de la variable 3 se ha reducido y es posible que ahora exista inconsistencia con variables con las que antes era consistente.

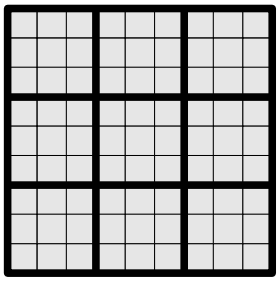
4. Fase de experimentación

4.1. Casos de prueba concretos

En este apartado se van a realizar una serie de pruebas con distintos tableros iniciales para medir el tiempo promedio que tarda en completarse un sudoku ejecutando los algoritmos implementados: Forward Checking y AC3.

El primer tablero evaluado se corresponde con un tablero completamente vacío, sin ningún elemento predefinido:

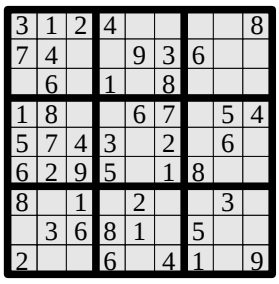
#1	Con AC3	Sin AC3
Llamadas recursivas a FC	193	193
Llamadas a la función forward	279	279
Llamadas a la función restaurar	199	199
Tiempo invertido en AC3 (ms)	5	0
Tiempo invertido en FC (ms)	3	3
Tiempo total (ms)	8	3



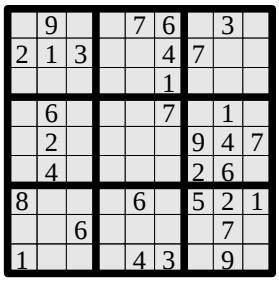
En este caso, AC3 no realiza ninguna modificación sobre las variables del tablero, por lo que el tiempo gastado en AC3 no se aprovecha. Se producen las mismas llamadas recursivas en ambas ejecuciones (con y sin AC3).

Las siguientes pruebas se realizan sobre tableros aleatorios sin ninguna particularidad, para evaluar cómo varían los tiempos y el número de llamadas en función de AC3:

#2	Con AC3	Sin AC3
Llamadas recursivas a FC	81	193
Llamadas a la función forward	80	279
Llamadas a la función restaurar	0	199
Tiempo invertido en AC3 (ms)	16	0
Tiempo invertido en FC (ms)	1	6
Tiempo total (ms)	17	6

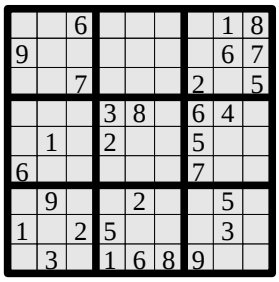


#3	Con AC3	Sin AC3
Llamadas recursivas a FC	98	875
Llamadas a la función forward	101	1527
Llamadas a la función restaurar	21	1447
Tiempo invertido en AC3 (ms)	19	0
Tiempo invertido en FC (ms)	1	10

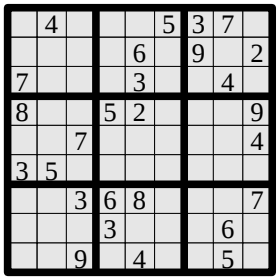


Tiempo total (ms)	20	10

#4	Con AC3	Sin AC3
Llamadas recursivas a FC	860	7088
Llamadas a la función forward	1113	13499
Llamadas a la función restaurar	1033	13419
Tiempo invertido en AC3 (ms)	22	0
Tiempo invertido en FC (ms)	4	76
Tiempo total (ms)	26	76

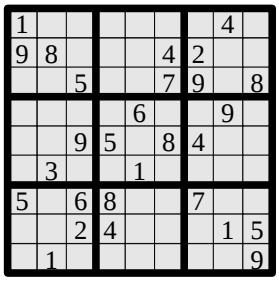


#5	Con AC3	Sin AC3
Llamadas recursivas a FC	2136	6672
Llamadas a la función forward	2742	13419
Llamadas a la función restaurar	2662	13339
Tiempo invertido en AC3 (ms)	20	0
Tiempo invertido en FC (ms)	9	81
Tiempo total (ms)	29	81

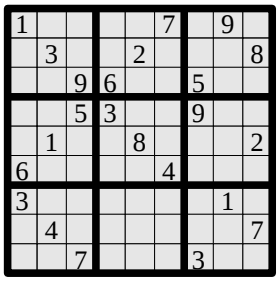


Los 2 siguientes tableros son considerados sudokus de gran dificultad:

#6	Con AC3	Sin AC3
Llamadas recursivas a FC	222	741
Llamadas a la función forward	275	1234
Llamadas a la función restaurar	195	1154
Tiempo invertido en AC3 (ms)	19	0
Tiempo invertido en FC (ms)	1	8
Tiempo total (ms)	20	8

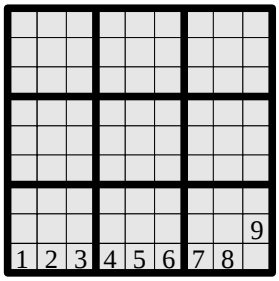


#7	Con AC3	Sin AC3
Llamadas recursivas a FC	7576	14880
Llamadas a la función forward	9694	27063
Llamadas a la función restaurar	9614	26983
Tiempo invertido en AC3 (ms)	22	0
Tiempo invertido en FC (ms)	39	164
Tiempo total (ms)	61	164

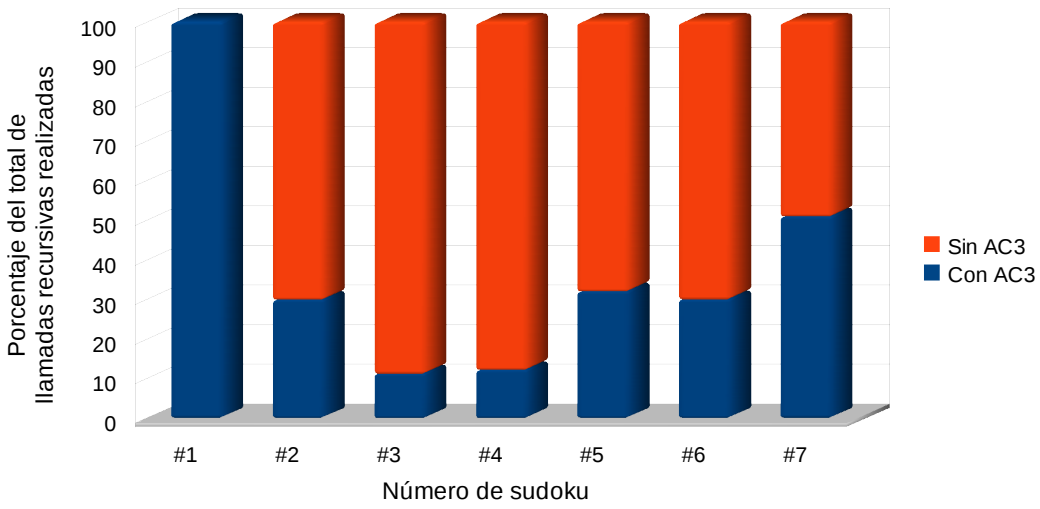


El siguiente tablero no tiene solución:

#8	Con AC3	Sin AC3
Llamadas recursivas a FC	0	-
Llamadas a la función forward	0	-
Llamadas a la función restaurar	0	-
Tiempo invertido en AC3 (ms)	14	0
Tiempo invertido en FC (ms)	0	∞
Tiempo total (ms)	14	∞

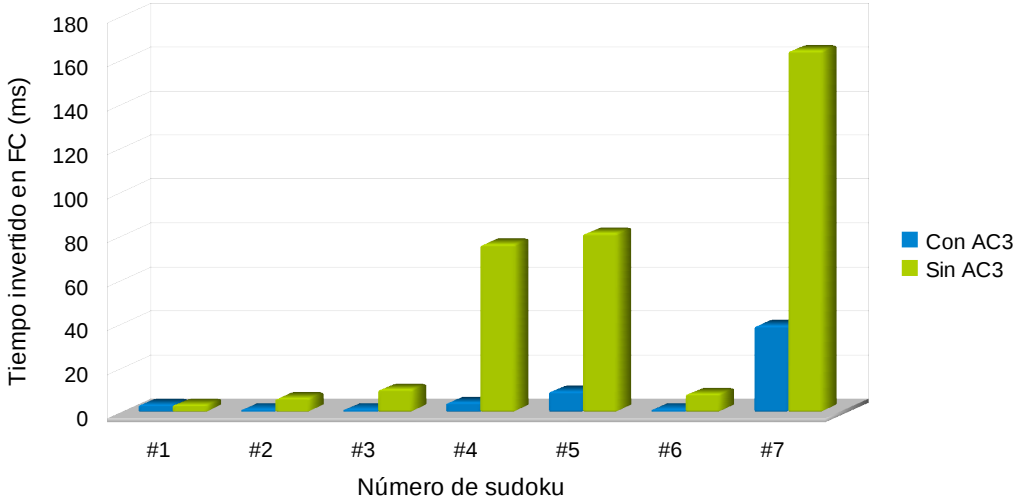


AC3 impide que se ejecute el algoritmo Forward Checking ya que no encontrará ninguna solución. Cuando se ejecuta sin AC3, el algoritmo Forward Checking comprueba todas las posibles combinaciones del tablero hasta que encuentra alguna, descartándolas una tras otra hasta que ya no queden más. Se realiza un backtracking que puede llegar a examinar hasta 81^9 combinaciones distintas si no encuentra alguna solución. Si no existiera una solución, sí que se examinarían todas combinaciones sin llegar a encontrar ninguna. El tiempo necesario para examinar estas combinaciones es tan grande que en la tabla se ha marcado con el símbolo ∞, aunque el tiempo de cálculo es finito. Si comprobáramos los 81^9 tableros distintos suponiendo que se invierte 1 nanosegundo en cada uno de ellos, se tardaría un total de 4 años y medio en evaluarlos todos. Además, 1 nanosegundo por tablero es sólo una estimación aproximada.



El diagrama de barras anterior representa el total de llamadas recursivas que se han realizado –en porcentaje– aplicando AC3 antes que Forward Checking o no aplicándolo. El 100% de las llamadas coincide con el número de llamadas sin AC3. Es decir, se marca qué porcentaje de llamadas recursivas se han ejecutado al aplicar AC3 respecto a la ejecución sin AC3. Por ejemplo, en el gráfico se muestra que el tablero #1 realiza las mismas llamadas aunque ejecute el algoritmo AC3. En el tablero #7 se han ahorrado aproximadamente un 50% de llamadas recursivas.

A continuación se ilustran los tiempos invertidos en la función Forward Checking respecto a ejecutar el algoritmo AC3 o no.



En algunos casos, AC3 evitará que examinemos tableros que no tienen solución, ahorrando recorrer todas las combinaciones posibles. En la mayoría de casos, AC3 sólo disminuye el rango de combinaciones de tableros a comprobar al reducir el dominio de las variables. Esto se refleja en los resultados con una menor cantidad de llamadas recursivas a la función y un menor tiempo de ejecu-

ción.

4.2. Casos de prueba aleatorios

A continuación se van a realizar pruebas del algoritmo con tableros aleatorios. Estos tableros tendrán una cantidad concreta de casillas predefinidas. De esta manera, se examina el algoritmo en distintos tipos de tableros. Se procurará que estos tableros aleatorios siempre tengan solución, por lo que no se entrará en un gran bucle que impida que Forward Checking acabe en un tiempo razonable.

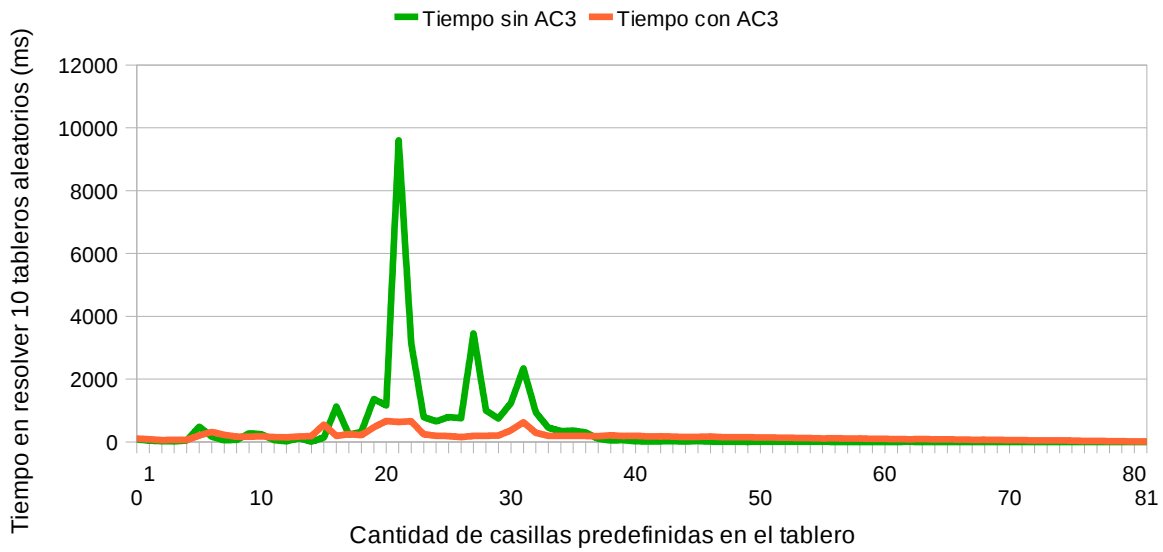
En total, se van a calcular 10 tableros aleatorios para cada cantidad posible de casillas predefinidas –desde 0 hasta 81–. Es decir, 820 tableros aleatorios. En este código se muestra cómo se generan los tableros y se miden los tiempos de ejecución.

```
for (int i = 0; i <= 81; i++)
{
    int cantidadTableros = 10;
    for (int j = 0; j < cantidadTableros; j++)
    {
        // Se genera el tablero con una cantidad “i” de casillas
        // predefinidas.
        Tablero tablero = generarTablero(i);

        // Inicializar el dominio de las variables desde el tablero.
        ...

        // Se mide el tiempo de aplicar FC.
        FC(0);

        // Se mide el tiempo de aplicar FC y AC3.
        AC3();
        FC(0);
     }
}
```

La siguiente tabla muestra los tiempos en milisegundos que tarda el algoritmo en calcular 10

sudokus en cada nivel de casillas predefinidas:

Celdas predefinidas	Tiempo sin AC3	Tiempo con AC3	Celdas predefinidas	Tiempo sin AC3	Tiempo con AC3
0	80	106	41	13	174
1	43	84	42	19	181
2	24	56	43	23	168
3	19	66	44	14	157
4	46	63	45	28	162
5	485	205	46	10	172
6	170	318	47	7	150
7	53	224	48	5	150
8	65	167	49	2	154
9	273	165	50	8	138
10	243	179	51	4	138
11	58	153	52	5	135
12	23	146	53	4	124
13	118	171	54	5	122
14	13	187	55	3	109
15	138	550	56	2	115
16	1124	191	57	2	106
17	232	249	58	2	111
18	317	214	59	2	94
19	1361	465	60	0	95
20	1165	665	61	2	91
21	9600	634	62	3	85
22	3157	660	63	1	91
23	791	242	64	0	84
24	655	195	65	0	81
25	790	186	66	1	74
26	757	154	67	1	69
27	3455	191	68	1	70
28	997	196	69	1	64
29	747	201	70	1	57
30	1237	368	71	0	56
31	2342	626	72	1	48
32	937	291	73	2	47
33	454	193	74	0	48
34	342	188	75	0	42
35	359	188	76	1	32
36	300	188	77	2	30
37	101	183	78	0	25
38	48	209	79	0	23
39	57	187	80	0	13
40	24	195	81	1	14



Se puede ver gráficamente que, cuando no se aplica AC3, es más probable que Forward Checking alcance picos altos de tiempo. En los tableros con 15-35 casillas predefinidas, debido al backtracking que se ve forzado a realizar, es más probable que Forward Checking tome tiempos más largos de ejecución.

A partir de las 40 casillas predefinidas hay muchas variables con el dominio acotado a un único valor, por lo que se requiere menos decisión por parte de Forward Checking. Por otro lado, cuando hay pocas casillas con su valor predefinido –entre 0 y 10– el dominio de las variables está poco acotado, pudiendo asignar prácticamente cualquier valor, evitando también que Forward Checking detecte variables que se han quedado sin elementos factibles en su dominio y tenga que retroceder. En estos casos es menos probable que Forward Checking de marcha atrás para asignar otro valor a la variable. En estos casos no conviene aplicar AC3 antes de Forward Checking ya que el tiempo que tarda en completarse el algoritmo con AC3 es mayor que el tiempo que tarda en completarse Forward Checking sin AC3.

5. Notas para el profesor

A continuación aparecen los puntos relevantes del código que son importantes para la corrección:

Módulo	Localización	
	Parte obligatoria (src/)	Parte opcional (adicional/)
Función ejecutarFC	Jugador.java (línea 26)	Jugador.java (línea 29)
Clase Variable	Jugador.java (línea 218)	Jugador.java (línea 296)
Función FC	Jugador.java (línea 69)	Jugador.java (línea 77)
Función forward	Jugador.java (línea 109)	Jugador.java (línea 117)
Función restaurar	Jugador.java (línea 171)	Jugador.java (línea 179)
Clase Arista	(sólo en la parte adicional)	Jugador.java (línea 495)
Función AC3	(sólo en la parte adicional)	Jugador.java (línea 227)