Programación Lógica

Laboratorio 2

Germán Faller - 4.520.304-7
Ernesto Gerez - 4.746.122-5
Santiago Ingold - 5.040.942-4

Contenido

Módulos y predicados

Los módulos utilizados fueron *graficos.pl* y *kakuro.pl*. No se implementaron módulos adicionales. Dentro de *kakuro.pl* se definieron los siguientes predicados principales para las distintas partes.

Parte 1

juego(+N,+Tablero)

Genera un tablero a partir de un número. El tablero debe estar definido en el módulo *kakuro.pl*. Se utiliza para invocar al programa con un tablero definido.

kalog(+Tablero)

Invoca al módulo *graficos.pl* para generar la ventana con el tablero indicado y permite interacción con el usuario.

validar_tablero(+Visual, +Tablero)

Validación de las reglas del juego para el jugador humano. Chequea que en un bloque completo se satisfaga la suma y que no hayan elementos repetidos. Utiliza métodos auxiliares para validar línea por línea. La lógica del predicado recorre un tablero primero por filas y luego por columnas y según el tipo de casillero encontrado realiza las siguientes acciones:

- Casillero en blanco (variable libre): Se agrega la variable a una lista (bloque en progreso) y se avanza la recursión al siguiente elemento.
- Casillero negro (n): Si el bloque en progreso no es vacío y está completo (no tiene variables libres) se realiza la validación de la suma acumulada de los números en el bloque. De lo contrario, si tiene variables libres, se verifica que los números no excedan la suma indicada para el bloque.
- Casillero de suma (f,c y p): Los casilleros que indican una suma son discriminados según el sentido de la recorrida. Por ejemplo, cuando se encuentra un casillero de tipo *f* mientras se recorre el tablero por filas, el argumento del functor f indica el valor que debe satisfacer la suma de los casilleros del nuevo bloque (luego de un casillero f debe seguir un bloque de casilleros rellenables asumiendo que el tablero es correcto). De lo contrario, lógica es la misma que si se encontrase un casillero negro. El caso de un casillero de tipo *c* es análogo y el de *p* es similar, tomando el primer o segundo argumento según el sentido de la recorrida.

Nota 1: Si se encuentra un casillero negro,se valida la suma, se reinicia el bloque en progreso, la suma acumulada en quedaria en 0 y se continúa con la recursión.
Nota 2: Si en una recorrida por fila encontramos un casillero c este es tratado como un n

es_casilla_invalida(+Tablero, +F, +C)

Verifica que la casilla a editar sea válida para jugar, que sea una casilla blanca, dando true si es invalida junto con un OR en la invocación y aprovechando el circuito corto, el siguiente método, que sería asignar el valor al tablero no se ejecuta.

pintar_tablero(+Tablero, +Visual)

Llama a los métodos del módulo *graficos.pl* para marcar el tablero ante un evento.

Parte 2

kalog(+Tablero,+Técnica)

Resuelve el tablero dado mediante la técnica especificada.

rangos_sumas(+Tablero, ?Rangos, ?Sumas, +Sentido)

Hace recorridas del tablero capturando los bloques con su respectiva restricción de suma, se usa sentido para indicar si es una recorrida por fila o columna previa transposición del tablero. Si en una fila no hay casilleros blancos, como en la primera, retorna para ese rango una lista vacía y la suma 0,

Técnica std

kalog_rango_std(+Rango, +Suma)

Recordar que los rangos son los conjuntos de variables libres del tablero agrupadas de a bloque, este método va completando con números del 1 al 9 sin repetir y una vez completo valida que ese rango cumpla la suma. Es invocado para todos los rangos identificados con el método rangos_sumas.

Técnica clpfd

kalog_rango(+Rango, +Suma)

Usa las restricciones: all_distinct(Rango), sum(Rango, #=, Suma)
Previamente se le dio como restricción a todas las variables que estuvieran entre 1 y 9.

Parte 3

kalog(+Filas, +Columnas, ?Tablero)

Básicamente la generación de tableros se compone de 3 partes principales: (1) generar un tablero válido, sin bloques mayores a 9 (en un principio mayores a 1, pero se quitó) y que cada bloque tenga su respectivo cabezal. (2) completar con números válidos del 1 al 9 sin que se repitan en el bloque. (3) El cálculo de los cabezales se hace en 2 pasadas, primero por fila y luego por columnas para dejar el tablero listo con su solución incluida. Antes de ser devuelto, el tablero pasa por un proceso de reinicio dejando las casillas blancas vacías.

generar_tablero_valido(+F,+C,?T)

Llama C-1 veces a generar_fila el cual se compone de generar subFilas (bloques) de la siguiente manera, sortea un número entre 1 y min(Tamaño, 9) donde tamaño es el tamaño de la fila menos el tamaño de lo que ya hemos generado (la suma de tamaños de los bloques anteriores), con este número sorteado genera el bloque de ese tamaño con la primera casilla en negro y las demás en blanco, notar que si el valor sorteado es 1 solamente generará una casilla negra y si la suma de casillas generadas alcanza el tamaño de la fila corta la recursión.

Luego, antes de retornar el Tablero genera una fila de todos negros para ubicarla al inicio de las filas generadas.

fill(?T)

Usa métodos estándar, se encarga de llenar con números del 1 al 9 sin repetir teniendo en cuenta los números que ya ubicó en el bloque de la fila recorrida y el bloque vertical correspondiente a la casilla.

calcular_cabezales(+T1,?T2),

Recorre el tablero calculando las sumas de los rangos anteriormente definidos, se basa en funciones auxiliares para tener en cuenta una recorrida por filas o columnas.

Comparación de kalog std vs. clpfd

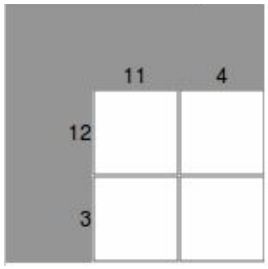
Declaratividad

La estrategia std utiliza técnicas y predicados de Prolog “puro”. Como consecuencia, los predicados son propensos a faltas de eficiencia y el código es más complejo. Por ejemplo, en el caso de la resolución del kakuro mediante la estrategia estándar, el predicado kalog_rango_std debe implementar la lógica de generar números del 1 al 9 y probar que la elección cumpla con las reglas del problema.

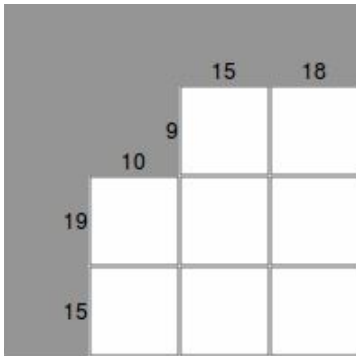
Por otra parte, la estrategia clpfd consigue una resolución más clara, fácil de entender, con menos cantidad de predicados (considerando los predicados de más alto nivel utilizados y despreciando los detalles de implementación de la biblioteca). La programación con restricciones en los enteros permite dar valores a las variables que cumplan con un conjunto de reglas e inferir nuevas reglas incluso antes de ejecutar consultas, por lo que la eficiencia de un predicado que utiliza clpfd tiende a ser mucho mayor a la de un predicado de prolog puro.

Análisis de eficiencia

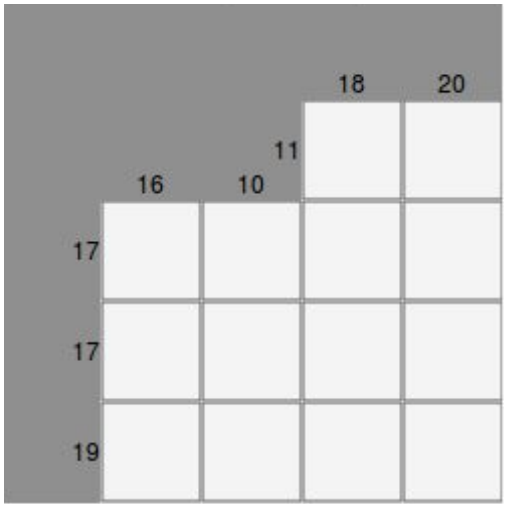
Para ejecutar en análisis empírico de eficiencia, se generaron los siguientes tableros:
3x3



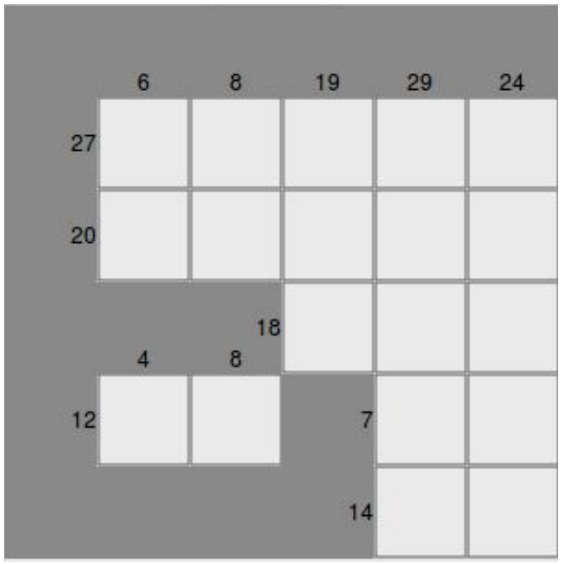
4x4



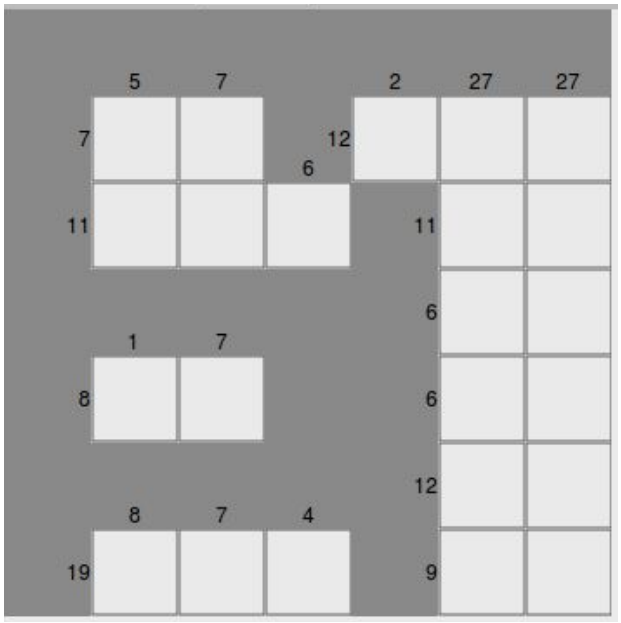
5x5



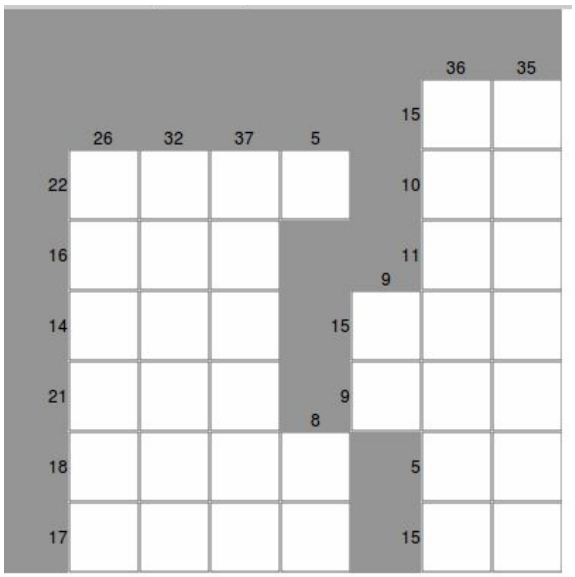
6x6



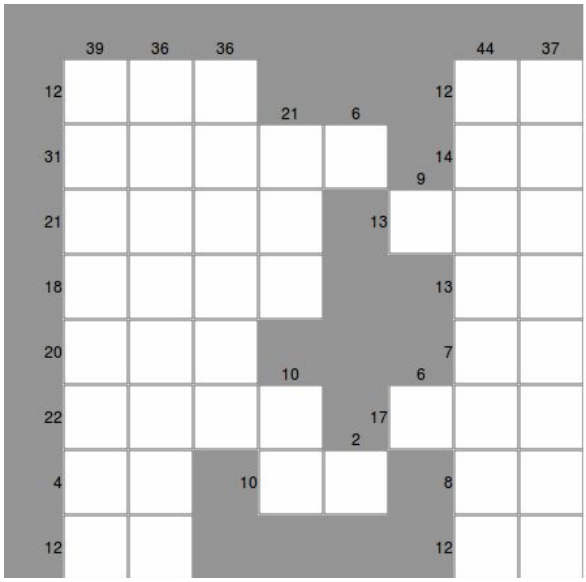
7x7


5
7
2
27
27
7
12
6
11
11
6
1
7
8
6
12
8
7
4
19
9

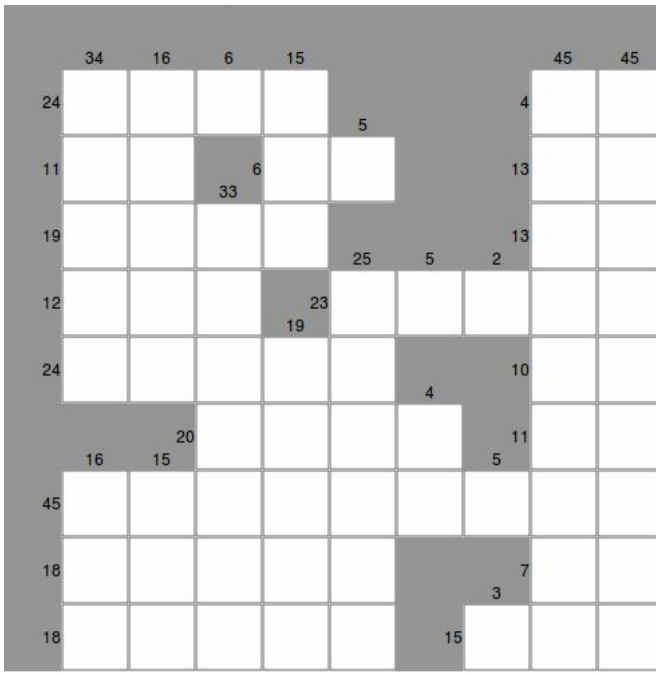
8x8


36
35
15
26
32
37
5
22
10
16
11
9
14
15
21
9
8
18
5
17
15

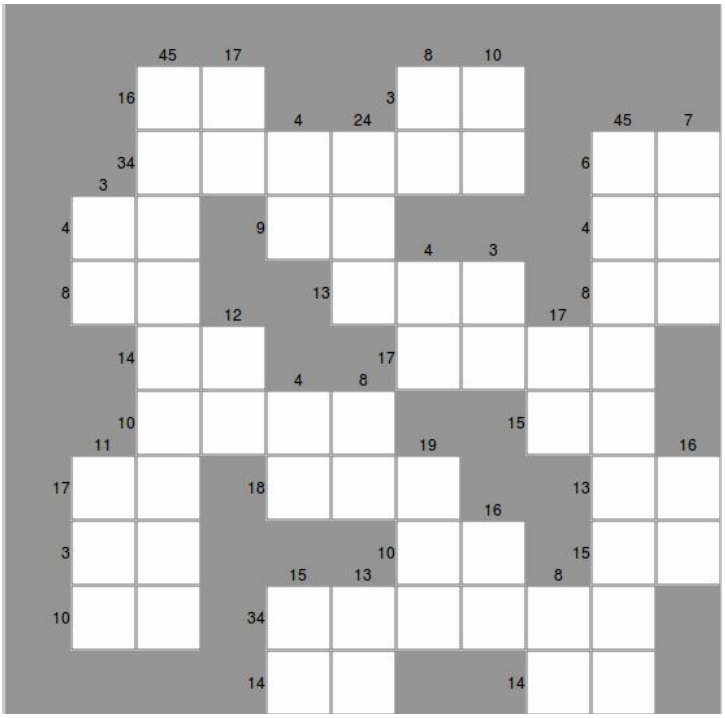
9x9


39 36 36 44 37
12 12
21 6
31 14
9
21 13
18 13
20 7
10 6
22 17
2
4 10 8
12 12

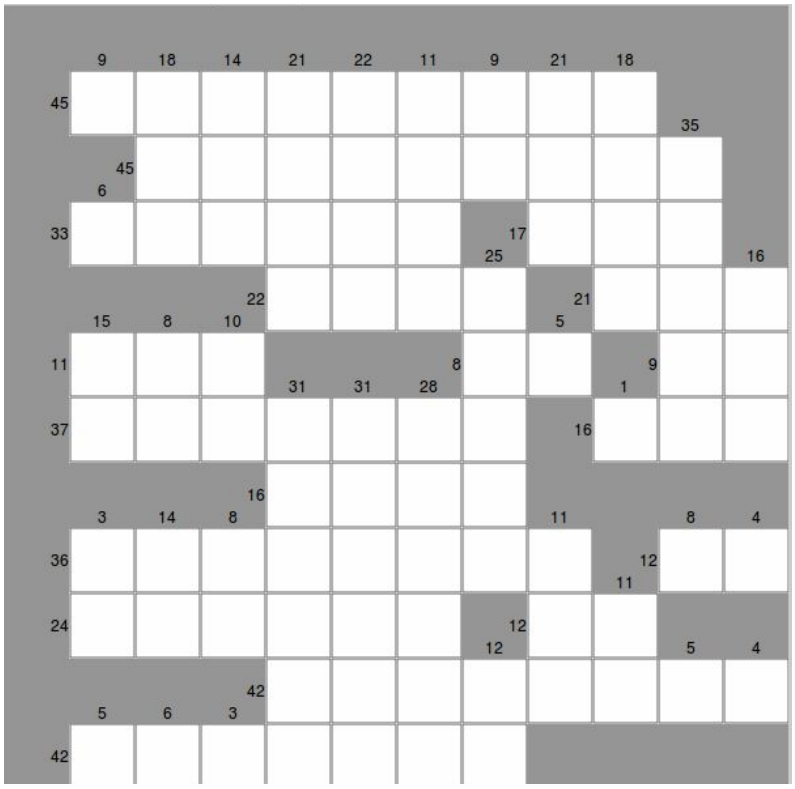
10x10


34 16 6 15 45 45
24 4
5
11 6 13
33
19 13
25 5 2
12 23
19
24 10
4
20 11
16 15 5
45
18 7
3
18 15

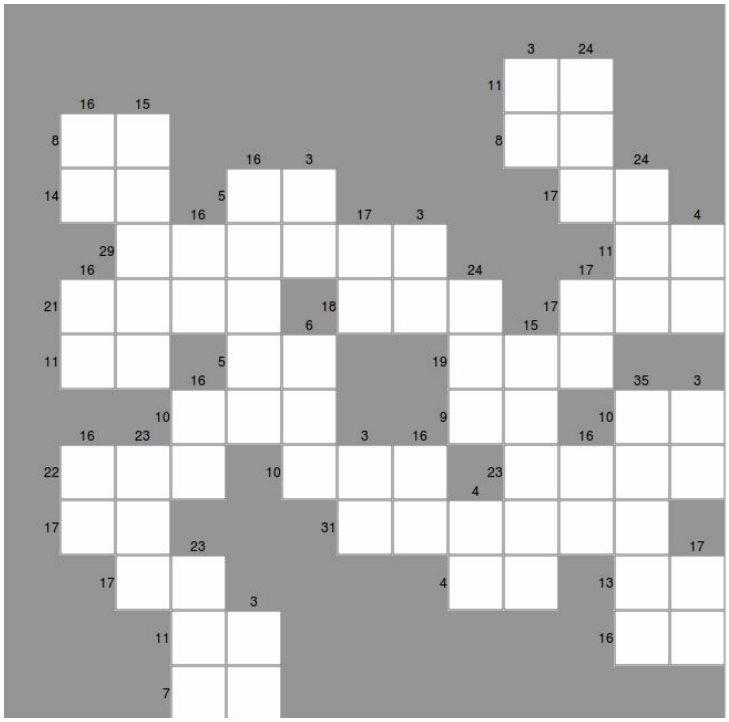
11x11


45 17 8 10
16 3
4 24 45 7
34 6
3
4 9 4
4 3
8 13 8
12 17
14 17
4 8
10 15
11 19 16
17 18 13
16
3 10 15
15 13 8
10 34
14 14

12x12


9 18 14 21 22 11 9 21 18
45 35
45
6
33 17
25 16
22 21
15 8 10 5
11 8 9
31 31 28 1
37 16
16
3 14 8 11 8 4
36 12
11
24 12
12 5 4
42
5 6 3
42

13x13



A continuación se muestran los resultados de la ejecución de los algoritmos std y clpfd para cada tablero:

Tablero	Algoritmo std		Algoritmo clpfd	
	Tiempo	Resultado de time/1	Tiempo	Resultado de time/1
3x3	0,001	% 4,510 inferences, 0.001 CPU in 0.001 seconds (100% CPU, 3490377 Lips)	0,003	% 11,978 inferences, 0.003 CPU in 0.003 seconds (100% CPU, 3466213 Lips)
4x4	0,002	% 8,325 inferences, 0.002 CPU in 0.002 seconds (100% CPU, 3558253 Lips)	0,012	% 43,476 inferences, 0.012 CPU in 0.012 seconds (100% CPU, 3502985 Lips)

5x5	1,1823	% 19,219,120 inferences, 1.819 CPU in 1.823 seconds (100% CPU, 10566099 Lips)	0,022	% 124,599 inferences, 0.022 CPU in 0.022 seconds (100% CPU, 5766904 Lips)
6x6	67,699	% 690,999,039 inferences, 67.700 CPU in 67.699 seconds (100% CPU, 10206787 Lips)	0,028	% 174,640 inferences, 0.028 CPU in 0.028 seconds (100% CPU, 6345097 Lips)
7x7	Se cortó la ejecución luego de 20 minutos	n/c	0,022	% 118,454 inferences, 0.022 CPU in 0.022 seconds (100% CPU, 5325666 Lips)
8x8	n/c	n/c	0,178	% 1,447,250 inferences, 0.178 CPU in 0.178 seconds (100% CPU, 8130679 Lips)
9x9	n/c	n/c	37,175	% 304,942,470 inferences, 37.175 CPU in 37.175 seconds (100% CPU, 8202844 Lips)
10x10	n/c	n/c	1,747	% 13,683,258 inferences, 1.747 CPU in 1.747 seconds (100% CPU, 7833924 Lips)
11x11	n/c	n/c	0,053	% 302,776 inferences,

				0.053 CPU in 0.053 seconds (100% CPU, 5727391 Lips)
12x12	n/c	n/c	0,399	% 3,083,541 inferences, 0.399 CPU in 0.399 seconds (100% CPU, 7729983 Lips)
13x13	n/c	n/c	0,064	% 390,018 inferences, 0.064 CPU in 0.064 seconds (100% CPU, 6063019 Lips)

Podemos ver que el desempeño de la solución utilizando clpfd es claramente superior a partir de los puzzles de 5x5, siendo mejor el desempeño de std en los de menor tamaño. Suponemos que esto se debe al overhead de usar clpfd, que es pequeño y solo se nota la diferencia en la resolución de los problemas más sencillos. También podemos ver que varía el resultado para distintas dificultades del puzzle del mismo tamaño, específicamente en el caso de clpfd vemos que en algunos casos como el de 9x9 demora un tiempo particularmente largo, lo que sería producto de la dificultad del puzzle para el algortimo desarrollado.
Por razones de tiempo no fue posible ejecutar el algoritmo para múltiples tableros de cada tamaño, pero suponemos que de hacerlo no se minimizarían ese tipo de anomalías.

Mejoras Posibles

Por temas de tiempo no se implementaron mejoras que se nos ocurrieron pero pueden ser tenidas en cuenta para un trabajo futuro en este tema.

En la parte 1, se podrian agregar mas chequeos por ejemplo cuando una variable está libre pero teniendo en cuenta las dos sumas (horizontal y vertical) los valores posibles ya están ubicados estaríamos frente a una falla fácilmente detectable para el humano la cual podríamos marcar; o viceversa, teniendo en cuenta los valores ubicados, los valores disponibles no cumplen con alguna de las sumas podremos marcarlo con error.

En la parte 2, debido a la eficiencia de CLPFD no analizamos en profundidad la resolución estándar pero para ser justos con la comparación deberíamos optimizar la resolución estándar, teniendo en cuenta la suma a la hora de completar los valores para no generar tantas soluciones erradas, además de tener en cuenta los valores usados en el bloque vertical. En cuanto a la solución mediante CLPFD es muy eficiente y no generamos tableros lo suficientemente grandes como para identificar una mejora.

En la parte 3, hay una mejora notoria a implementar a la hora de generar los tableros válidos, y es que se generan por fila y se valida por columna e inicialmente para tableros de más de 9 filas, se generan muchas columnas inválidas, regenerando todo el tablero. Además no se uso CLP para calcular los valores, lo cual aceleraría mucho el proceso de generación de tableros.