MEMORIA - PRÁCTICA 2
Usando SAT Solvers

Fecha de entrega:
19 - Noviembre - 2023

Autores y NIP:
Pilar Fierro Casanova (819658)
Óscar Brizuela García (820773)

Profesores:
Elvira Mayordomo
Ubaldo Ramón

Grado:
Cuarto curso de Ingeniería Informática

Asignatura:
Algoritmia para problemas difíciles

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza
1542

RESUMEN

Este documento recoge el trabajo realizado para llevar a cabo la práctica 2 de la asignatura de Algoritmia para Problemas Difíciles, donde se cuenta con una descripción de los aspectos más significativos del código fuente junto con una explicación de las pruebas realizadas y los resultados obtenidos.

Se han empleado dos enfoques distintos en la implementación para abordar el problema de los "Cuadrados Latinos": uno de ellos implica la reducción desde Latin Square a SATval (y, por ende, utilizando SAT como herramienta) para poder resolver dicho Latin Square. El segundo enfoque se trata de una implementación propia mediante la técnica del *backtracking* que resuelve dicho problema.

Para la realización de las pruebas se han empleado diferentes ficheros de texto con varios casos relevantes, siendo analizado el coste temporal de la resolución del Latin Square con ambos métodos.

Por último, una conclusión personal de lo aprendido en esta práctica junto con la organización de las tareas entre los integrantes del equipo y la webgrafía donde se recogen las fuentes externas empleadas.

ÍNDICE

1. CÓDIGO FUENTE ENTREGADO

En el código fuente entregado se encuentran dos ficheros con las respectivas implementaciones comentadas anteriormente: en *main.cc* se encuentra la implementación con la reducción y en *backtracking.cpp* se encuentra la implementación que emplea *backtracking*. A continuación se encuentran las distintas implementaciones descritas con más detalle.

1.1. RESOLUCIÓN DE LATIN SQUARE MEDIANTE SAT

El problema del Latin Square consiste en colocar *n* elementos del 1 al *n* en una matriz *nxn* con la condición de que no se repita ningún número en la misma fila y tampoco se repita en la misma columna.

Para realizar la reducción de SAT que permite resolver el problema del Latin Square se han realizado las transformaciones necesarias para añadir las cláusulas CNF del SAT de manera que cumplan las restricciones del Latin Square. Por ello, la entrada al algoritmo que resuelve SAT es cambiada de una matriz de cadenas de caracteres leída de un fichero de entrada a un circuito CNF con diferentes cláusulas que es aceptado como entrada para SAT. Esta matriz cuenta con la siguiente estructura: cada celda está caracterizada por $x_{ijk,}$ siendo i la fila, j la columna y k el valor numérico asignado a dicha celda.

Por un lado se cuenta con las cláusulas de existencia: $(x_{ij1} \vee x_{ij2} \vee \dots \vee x_{ijn})$ que asegura que al menos uno de los n valores del dominio [1,n] se asigna a x_{ij}, que es una celda concreta que se añade al circuito CNF.

Por otro lado se cuenta con las cláusulas de exclusión mutua, donde se indica que no se pueden asignar dos valores a la misma celda. Se debe asegurar la exclusión mutua de los valores añadiendo a nuestro circuito correspondiente una celda concreta con una serie de cláusulas de la siguiente forma:

$$\neg (x_{ij1} \wedge x_{ij2}) \wedge \neg (x_{ij1} \wedge x_{ij3}) \wedge \neg (x_{ij2} \wedge x_{ij3}) \wedge \dots \wedge \neg (x_{ij1} \wedge x_{ijn}) \wedge \neg (x_{ij2} \wedge x_{ijn}) \wedge \neg (x_{ijn-1} \wedge x_{ijn})$$

Estas cláusulas, al no estar en CNF, deben ser transformadas utilizando las leyes de Morgan de la siguiente forma:

$$\neg (x \wedge y) = (\neg x \vee \neg y)$$

Por tanto, el conjunto de cláusulas anterior quedaría de la siguiente manera:

$$(\neg x_{ij1} \vee \neg x_{ij2}) \wedge (\neg x_{ij1} \vee \neg x_{ij3}) \wedge (\neg x_{ij2} \vee \neg x_{ij3}) \wedge \dots \wedge (\neg x_{ij1} \vee \neg x_{ijn}) \wedge (\neg x_{ij2} \vee \neg x_{ijn}) \wedge (\neg x_{ijn-1} \vee \neg x_{ijn})$$

Hasta el momento, se ha asegurado que en el circuito CNF se han añadido las cláusulas necesarias para añadir un valor a la celda que esté dentro del dominio [1, n] y, además, que esta celda solo contenga un valor, no más. Por tanto se deben añadir las cláusulas necesarias para cumplir con las restricciones propias del problema del Latin Square.

Para asegurar que cada valor sólo aparece una vez en una fila determinada, debemos añadir el siguiente conjunto de cláusulas:

$$(x_{i1k} \vee x_{i2k} \vee \dots \vee x_{ink}) \wedge (\neg x_{i1k} \vee \neg x_{i2k}) \wedge (\neg x_{i1k} \vee \neg x_{i3k}) \wedge (\neg x_{i2k} \vee \neg x_{i3k}) \wedge \dots \wedge (\neg x_{in-1k} \vee \neg x_{ink})$$

De esta manera un valor concreto de *k* no puede aparecer dos veces en una misma fila. De la misma manera hay que asegurarse de que cada valor de *k* sólo aparece una vez en una misma columna. Para ello se cuenta con las siguientes cláusulas que cumplen dicha restricción:
$(x_{1jk} \vee x_{2jk} \vee \ldots \vee x_{njk}) \wedge (\neg x_{1jk} \vee \neg x_{2jk}) \wedge (\neg x_{1jk} \vee \neg x_{3jk}) \wedge (\neg x_{2jk} \vee \neg x_{3jk}) \wedge \ldots \wedge (\neg x_{n-1jk} \vee \neg x_{njk})$

Por último, al estar todas las cláusulas relacionadas entre sí por puertas AND, el circuito CNF completo será unir todas esas puertas lógicas mediante estas mismas puertas para lograr la conjunción de todas ellas.

El resultado de la ejecución de este código, el cual está localizado en el fichero *main.cc*, se muestra por pantalla con los mensajes correspondientes: si se ha podido ejecutar o no y, en caso de haberse podido ejecutar, el Latin Square resuelto.

1.2. RESOLUCIÓN LATIN SQUARE MEDIANTE *BACKTRACKING*

Para la implementación que no emplea SAT, si no que hace uso de la técnica de *backtracking*, se utiliza una matriz de cadenas de carácteres (vector estático de vectores estáticos de elementos de tipo *strin*g) como estructura de datos donde almacenar los elementos del Latin Square a resolver.

Se ha optado por esta matriz ya que sus dimensiones son fácilmente configurables en función del tamaño *n* n del Latin Square.

Además, el hecho de que almacene cadenas de caracteres y no simplemente caracteres o datos de tipo entero *int* se debe a que el cuadrado puede contener números de una cifra o más, además del carácter "*". De esta forma, a pesar de que el tamaño de la matriz es estático una vez está definido *n*, es muy fácil pasar la estructura de datos como argumento para las demás funciones de las que se harán uso en la implementación.

En primer lugar, se lee la información del fichero de entrada correspondiente y se rellena dicha estructura con las cadenas de caracteres leídas mediante la función *fillMatrixFromFile()*. Una vez se tiene la matriz con todos los datos, la función *main* llama a *solveLatinSquare()* con el tamaño de *n* correspondiente, y le indica a dicha función que empiece a intentar completar el Latin Square por el primer elemento de la matriz (aquel cuyo índice es [0, 0]).

La función *SolveLatinSquare()*, cada vez que se encuentra un carácter "*", que representa un número a "adivinar", intenta colocar en la posición de dicho carácter un número correcto, es decir, que no esté ni en la misma fila ni en la misma columna que dicho carácter "*". Esta comprobación se lleva a cabo en la función *isSafe()* que garantiza que, hasta el momento, el número en cuestión se puede colocar en esa posición. El número que primero se intenta colocar es el 1 y, si no se puede, se va probando con el 2, 3… hasta *n*. Si se puede colocar alguno de esos números, se realiza una llamada recursiva a *SolveLatinSquare()* con la siguiente columna, hasta llegar a la columna *n*, momento en el que se realiza también una llamada recursiva con la siguiente fila hasta completar el cuadrado completo.

Sin embargo, si el Latin Square no tiene solución (porque el cuadrado original de entrada tiene dos números iguales en la misma fila o en la misma columna), la función *SolveLatinSquare()* devuelve false, determinando que es imposible resolver el Latin Square.

En caso de haber sido resuelto satisfactoriamente dicho Latin Square, este es escrito en un fichero de salida mediante la función *PrintLatinSquare()*. Si no, el programa simplemente muestra por pantalla un mensaje indicando que no lo puede resolver. En cualquiera de los dos casos se muestra el tiempo de ejecución.

2. PRUEBAS REALIZADAS

Para las pruebas se han realizado con distintos valores de *n*: 3, 9, 15, 25 y 50. Todas ellas han sido ejecutadas en la misma máquina con el objetivo de obtener unos resultados más objetivos a la hora de comparar las mediciones de los tiempos de ejecución.

Los pasos a seguir para realizar las pruebas con SAT (la reducción) y sin ella son diferentes, pero en ambos casos debe someter el fichero que desea probar donde se encuentra un Latin Square vacío (sólo con "*") o incompleto.

- *main.cc*: si se desea ejecutar el que contiene la reducción, se ha de llevar a cabo mediante el *makefile* que compila y ejecuta el *main.cc*. Para facilitar su compilación se encuentra un script *ejecutar.sh* que contiene los comandos correspondientes para su compilación y ejecución. En este caso, las pruebas del código implementado se han realizado mediante la terminal junto con las herramientas de *Visual Studio Code*. En la carpeta *build* se encuentra el ejecutable de este *main.cc: main_practica2.exe*

  Tras ejecutarlo, el programa solicita al usuario el tamaño *n* del Latin Square que desea resolver y el nombre del fichero *.txt* que lo contiene. Tras ello, muestra el mensaje correspondiente por pantalla: si se ha podido resolver o no, junto con el tiempo de ejecución y el propio Latin Square resuelto.

- *backtracking.cpp*: para ejecutar el fichero correspondiente al *solver* que utiliza la estrategia de *backtracking*, se debe compilar y ejecutar de la siguiente manera:
  - Compilación: *g++ backtracking.cpp -o <nombre_ejecutable>*
  - Ejecución: *./<nombre_ejecutable>*

  El programa solicitará por pantalla el tipo de prueba para ser realizada. Estos tipos están numerados del 1 al 8, siendo el 1 el correspondiente a la prueba con cuadrados vacíos y el 8 el correspondiente a los casos imposibles debido a repeticiones de números en una fila. Los números correspondientes a los tipos de pruebas están presentes, entre paréntesis, en los encabezados de cada prueba de esta sección.

  Finalmente, el programa solicitará, también por pantalla, el tamaño del cuadrado latino (*n*) que desea ser resuelto. Debido a los ficheros de entrada que se utilizan para las pruebas, solo se puede utilizar como *n* los valores 3, 9, 15, 25 y 50.

  Se advierte al usuario que, si no quiere malgastar su tiempo en pruebas complejas que requieren un gran tiempo de ejecución, se dirijan a la tabla para ver qué pruebas tienen tiempos de ejecución razonables en caso de que quiera probar alguna. Todos los resultados se presentan en milisegundos.

2.1. Cuadrados vacíos (1)

Se han utilizado como ficheros de entrada cuadrados de *n x n* en el que todos los elementos son el carácter "*".

Tamaño de *n* / *Solver*	*Backtracking*	Utilizando *SAT*
3	0.0000	0.0000
9	0.0000	37.913
15	49453.1	151.048
25	> 1'5 horas	1466.65
50	???	47912.7

2.2. Cuadrados parcialmente rellenos aleatoriamente (2, 3 y 4)

Se han utilizado cuadrados de *n x n* generados aleatoriamente mediante la herramienta:
https://www.dcode.fr/latin-square

Cabe apuntar que esta herramienta sólo permite generar Latin Squares aleatorios de 26 x 26 como mucho, por lo que los cuadrados de 50 x 50 se han conseguido con un fichero propio.

Una vez los cuadrados latinos aleatorios han sido generados mediante esta herramienta (que, al ser aleatoria, en cada ejecución muestra un cuadrado distinto para el mismo valor de *n*), se ha sustituido el 25%, el 50% y el 75% de los números aleatorios del cuadrado por el carácter "*" para utilizarlo como entrada de los *solvers*. Esta sustitución se ha realizado mediante la ejecución de un *script* en *Python* que utiliza como entrada el fichero con el Latin Square correspondiente y la probabilidad de reemplazar un número por un elemento a adivinar ("*").

2.2.1. 25% de elementos a adivinar (2)

Tamaño de *n* / *Solver*	*Backtracking*	Utilizando *SAT*
3	0.0000	0.999
9	0.0000	1.903
15	0.996	19.449
25	198.335	164.159
50	> 1'5 horas	> 0'5 horas

2.2.2. 50% de elementos a adivinar (3)

Tamaño de *n* / *Solver*	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	1.993	2.94
15	> 1’5 horas	32.021
25	???	774.338
50	???	> 0’5 horas

2.2.3. 75% de elementos a adivinar (4)

Tamaño de *n* / *Solver*	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	3.986	9.924
15	> 1’5 horas	198.709
25	???	851.661
50	???	> 0’5 horas

2.3. Casos especiales (5 y 6)

Para estos cuadrados se ha decidido que el *solver* intente adivinar el 50% de los números, por lo que aproximadamente uno de cada dos números serán “*” en el fichero de entrada.

2.3.1. Primera fila ordenada de forma ascendente (5)

En este fichero de entrada, la primera fila contiene todos los números ordenados de *1 a N*. La segunda fila contiene los números en el orden *2…N, 1*. La tercera fila contiene los números en orden *3…N, 1, 2*. Y así sucesivamente. Puede pensarse esta configuración como una operación lógica *ROR* de los números hacia la izquierda, de manera que el que estaba en el primer puesto en la fila *i* aparece al final en la fila *i + 1*.

Tamaño de *n* / Solver	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	0.0000	4.496
15	11759.8	24.229
25	> 1’5 horas	300.097
50	???	> 0’5 horas

2.3.2. Primera fila ordenada descendentemente (6)

En este fichero de entrada, la primera fila contiene todos los números ordenados de *N a 1*. La segunda fila contiene los números en el orden *N-1…1, N*. La tercera fila contiene los números en orden *N-2…1, N, N-1*. Y así sucesivamente. Esta configuración, de la misma forma que en el caso anterior, también puede pensarse como una operación lógica *ROR* de los números hacia la izquierda, de manera que el que estaba en el primer puesto en la fila i aparece al final en la fila *i + 1*.

Tamaño de *n* / *Solver*	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	4.982	2.989
15	1.30027 * 10^6 → 22 min aprox	27.486
25	> 1’5 horas	280.901
50	???	> 0’5 horas

2.4. Casos imposibles (7 y 8)

En estos casos, se mide el tiempo que tarda el *solver* correspondiente en determinar que no se puede resolver el Latin Square.

2.4.1. Caso imposible por columna (7)

El fichero de entrada cuenta con una columna en la que existen dos elementos iguales. La columna que tiene estos dos elementos es aquella de índice *n/2*, redondeado por arriba, de forma que se encuentra aproximadamente en la mitad del cuadrado latino. Además, los elementos repetidos de esta columna están en la fila también de índice *n/2* redondeado por arriba y en una fila cualquiera de las últimas del cuadrado latino.

Tamaño de *n* / *Solver*	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	284.564	3.417
15	318007.257 → 5.3 min aprox	26.155
25	> 1’5 horas	171.799
50	???	> 0’5 horas

2.4.2. Caso imposible por fila (8)

El fichero de entrada cuenta con una fila en la que existen dos elementos iguales. La fila que tiene estos dos elementos es aquella de índice *n/2*, redondeado por arriba, de forma que se encuentra aproximadamente en la mitad del cuadrado latino. Además, los elementos repetidos de esta fila están en la columna también de índice *n/2* redondeado por arriba y en una columna cualquiera de las últimas del cuadrado latino.

Tamaño de *n* / S*olver*	***Backtracking***	**Utilizando *SAT***
3	0.0000	0.0000
9	1.033	1.999
15	> 1’5 horas	22.917
25	???	149.242
50	???	> 0’5 horas

3. CONCLUSIONES DE LOS RESULTADOS

Como se puede comprobar, el *solver* que utiliza la estrategia de *backtracking* es muy bueno para Latin Squares pequeños. Sin embargo, debido a la gran cantidad de llamadas recursivas que realiza y las veces que tiene que volver hacia atrás para rehacer la configuración del Latin Square que había puesto en un principio, es muy poco eficiente para tamaños de cuadrado considerables.

En el caso peor (es decir, en el caso en el que está todo el fichero vacío), el coste del algoritmo por *backtracking* es $O(n^n)$. Esto se debe a que el algoritmo puede probar hasta *n* números en cada celda, y hay n^2 celdas en total.

Evidentemente, el caso medio es mejor que el caso peor, pero este depende de la configuración inicial del Latin Square. Al haber realizado varias pruebas, a pesar de que el coste algorítmico varía de una a otra, se puede comprobar cómo crece exponencialmente al aumentar el tamaño de la entrada (*n*).

Por otro lado, el *solver* que utiliza la reducción a *SAT* no es tan eficiente para tamaños de cuadrado pequeños, pero sí lo es, con una grandísima diferencia, para tamaños más grandes. Esto se debe a que el coste algorítmico para modificar la entrada y utilizar SAT es $O(n^4)$, procedente de los grupos de 4 bucles anidados para recorrer todas las filas, columnas, valores del dominio y pares de variables para asegurar la exclusión mutua. Al no haber ningún bloque de código con un número mayor de bucles anidados, se considera que el coste algorítmico es polinómico $O(n^k)$, con $k = 4$.

Esta es la razón por la que, para Latin Squares pequeños (*n* pequeño), la solución por *backtracking* obtiene un mejor rendimiento en cuanto a tiempo. Sin embargo, a medida que *n* crece, el tiempo de computación aumenta incluso más que exponencialmente, puesto que $O(k^n) << O(n^n)$, con *k* constante.

4. ASIGNACIÓN TAREAS

Para llevar a cabo la realización de la práctica entre los integrantes de la pareja, se ha realizado una asignación de tareas inicialmente donde se define el alcance de la práctica identificando los objetivos y requisitos, repartiéndose de manera equitativa.

Primero se han llevado a cabo las dos implementaciones de los Latin Square *solvers*, por un lado el que utiliza la estrategia *backtracking* y por otro el que hace uso de la librería *minisat* que resuelve el problema SAT.

A continuación, se han conformado los distintos ficheros de prueba para ser utilizados como entrada en ambos *solvers*. En la realización de las pruebas, se ha mostrado por pantalla los tiempos de cada una de las ejecuciones con el fichero de entrada correspondiente a ambos *solvers*. A medida que se iban recogiendo las pruebas, se iban recogiendo en las tablas de esta memoria.

Tras la realización de las pruebas, se ha procedido a la documentación del código.

Por último, se ha realizado la documentación de la memoria y se procede a la preparación de los ficheros fuente entregados.

5. WEBGRAFÍA

Se ha hecho uso de Copilot como soporte de ayuda en la elaboración de algunas funciones.

Herramienta para generar Latin Squares aleatorios de *nxn*: https://www.dcode.fr/latin-square