Algoritmia para problemas difíciles

Práctica 2:
Usando SAT Solvers

Álvaro Seral Gracia - 819425
Dorian Boleslaw Wozniak - 817570

Índice

Introducción..... 3
Detalles de diseño e implementación..... 4
Análisis de prestaciones de la implementación..... 6
Comparación con otros programas de resolución de cuadrados latinos..... 11
Distribución del trabajo..... 13
Bibliografía..... 14

Introducción

El objetivo de esta práctica es desarrollar un programa que resuelva cuadrados latinos convirtiendo el problema a uno de satisfacción booleana (SAT), utilizando una herramienta de resolución de dichos problemas. En nuestro caso, se ha seguido la propuesta del guión de utilizar Minisat.

El programa desarrollado resuelve tableros de cuadrados latinos. Un cuadrado latino es un problema similar al sudoku, con las siguientes reglas:

1. Los tableros son cuadrados de NxN casillas.
2. Los tableros contienen una serie de valores distintos, N en total.
3. Cada fila no puede repetir valores (p.e. no puede haber dos unos en la misma fila).
4. Cada columna no puede repetir valores.

La diferencia entre ambos problemas es que el sudoku adicionalmente divide el tablero en subcuadrados donde tampoco se puede repetir valores.

El programa lee, a partir de un fichero, un tablero. Este tablero puede contener casillas en blanco. Se convierte dicho tablero en una serie de ecuaciones booleanas en forma normal conjuntiva (CNF), donde cada cláusula es una disyunción de variables literales, y el objetivo es que se cumplan todas con una entrada.

El programa admite tableros de cualquier tamaño superior a 2, y comprueba que los tableros de entrada sean resolubles y los de salida sean correctos.

Se han realizado una serie de pruebas con variables de interés para el problema, principalmente el tamaño y la probabilidad de que cada casilla esté en blanco. También se ha comparado con otros programas de resolución de cuadrados latinos. Interesa especialmente observar el rendimiento del programa al ser SAT considerado un problema intratable (NP-completo), observando el crecimiento de tiempo de ejecución y uso de memoria.

Detalles de diseño e implementación

Los cuadrados latinos se pueden leer y escribir desde ficheros. Un cuadrado latino está formado por elementos separados por espacios en blanco, donde cada elemento es o un número entre 1 y N, siendo N el tamaño del tablero, o una casilla en blanco (*). Por ejemplo, el siguiente tablero es válido:

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

Para convertir los tableros en una ecuación booleana, se ha tomado como referencia una solución para generar un CNF para sudokus [1]. Al ser ambos problemas similares, se ha utilizado las reglas descritas salvo la de los subcuadrados que caracteriza al sudoku.

Estas reglas son:

1. Cada casilla del tablero debe contener alguno de los valores válidos.
2. Cada casilla debe contener un único valor (no puede contener dos valores simultáneamente).
3. Cada fila debe contener en cada casilla valores no repetidos.
4. Cada columna debe contener en cada casilla valores no repetidos.
5. El tablero resultante debe contener el tablero inicial, respetando el valor de cada número del tablero inicial.

Hay N^3 variables para un cuadrado de tamaño N, que representan una combinación de fila, columna y valor del cuadrado única. En cuanto a las cláusulas, habrá:

1. N^2 cláusulas para cada casilla, que será la disyunción de todos los valores posibles.
2. $N^2 * N * (N-1) / 2$ cláusulas binarias para las disyunciones de valores que no pueden estar presentes simultáneamente.
3. N^2 cláusulas para que cada fila contenga todos los valores.
4. N^2 cláusulas para que cada columna contenga todos los valores.
5. Una cláusula para cada valor ya conocido en una casilla.

El número total de cláusulas es $3N^2 + N^3 * (N-1) / 2$ + casillas conocidas.

Una vez obtenido el problema, se escribe en formato DIMACS [2]. A continuación, se invoca desde el programa a Minisat, pasando como valores el fichero generado y el nombre del nuevo fichero que contendrá la solución. Dicha entrada se procesa para generar un nuevo tablero que se escribe en un fichero.

Cabe destacar que, aunque Minisat ofrece una API para integrar directamente la funcionalidad de Minisat con el código, se ha optado por una solución más directa y sencilla de implementar invocando el propio binario de Minisat. Por ello, es necesario que el programa se encuentre instalado en el sistema y en la variable PATH.

Análisis de prestaciones de la implementación

Para analizar el rendimiento del programa, tanto en memoria como en tiempo, se ha realizado una serie de pruebas generando tableros de varios tamaños y porcentaje de casillas en blanco. En concreto:

- Se han probado tamaños de tableros desde 2 a 20. Tableros de tamaño mayor de 20 tienden a no terminar de ejecutarse en un tiempo razonable. Cada tablero contiene al menos un 30% de casillas vacías.
- Se han probado tableros de 9x9 donde se ha ido modificando el porcentaje de casillas en blanco desde 10% a 90%.

Minisat ofrece una salida de estadísticas útil para procesar la información de tiempo de ejecución y consumo de memoria. Se han medido los valores obtenidos para generar una serie de gráficas.

Tamaño	Memoria (MB)	Tiempo Minisat (s)	Tiempo total (s)
2	10,560000	0,001727	0,007000
3	10,560000	0,002566	0,007700
4	10,560000	0,003001	0,009000
5	10,560000	0,002648	0,008900
6	10,690000	0,003523	0,010600
7	10,690000	0,003174	0,013700
8	10,700000	0,003662	0,018900
9	10,820000	0,803098	0,026000
10	11,000000	0,004507	0,035800
11	11,120000	0,006136	0,048100
12	11,270000	0,006966	0,064400
13	11,740000	0,009848	0,082800
14	12,060000	0,015318	0,106700
15	12,340000	0,022737	0,151000
16	12,810000	0,042019	0,214200
17	13,300000	0,111744	0,332900
18	13,880000	0,238939	0,486600
19	16,575000	2,902183	3,224200
20	17,725000	1,709872	2,087900

Figura 1: Tabla de resultados para resolver diez tablas diferentes de varios tamaños

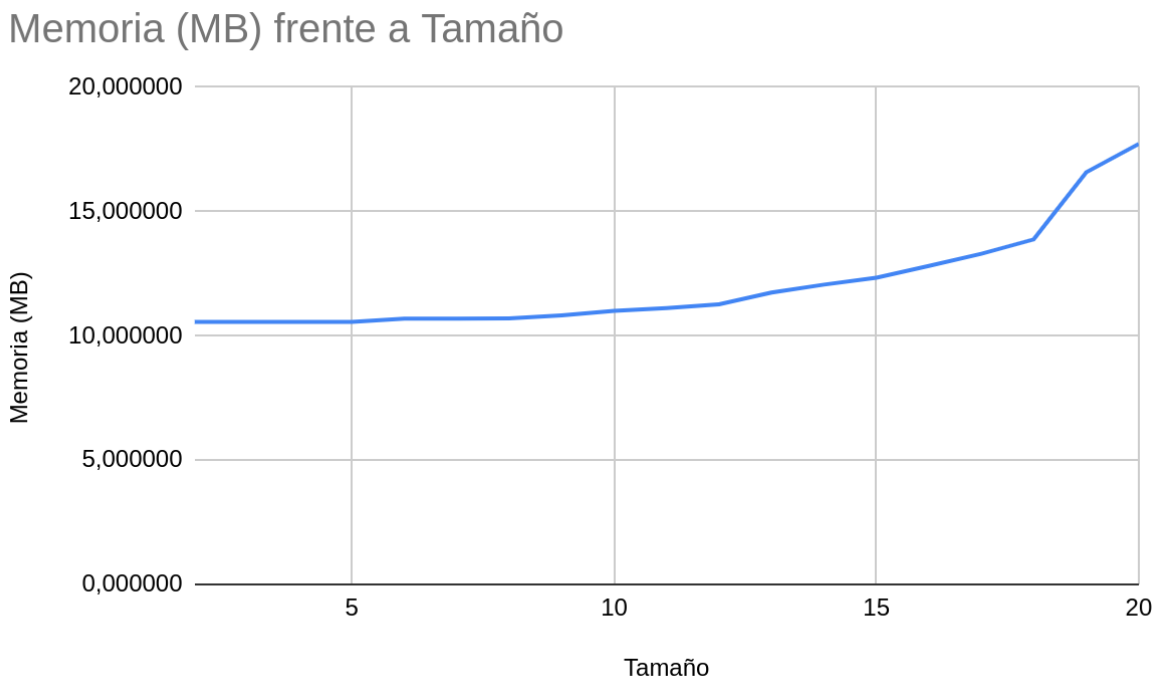

Figura 2: Gráfica de evolución del uso de memoria de Minisat

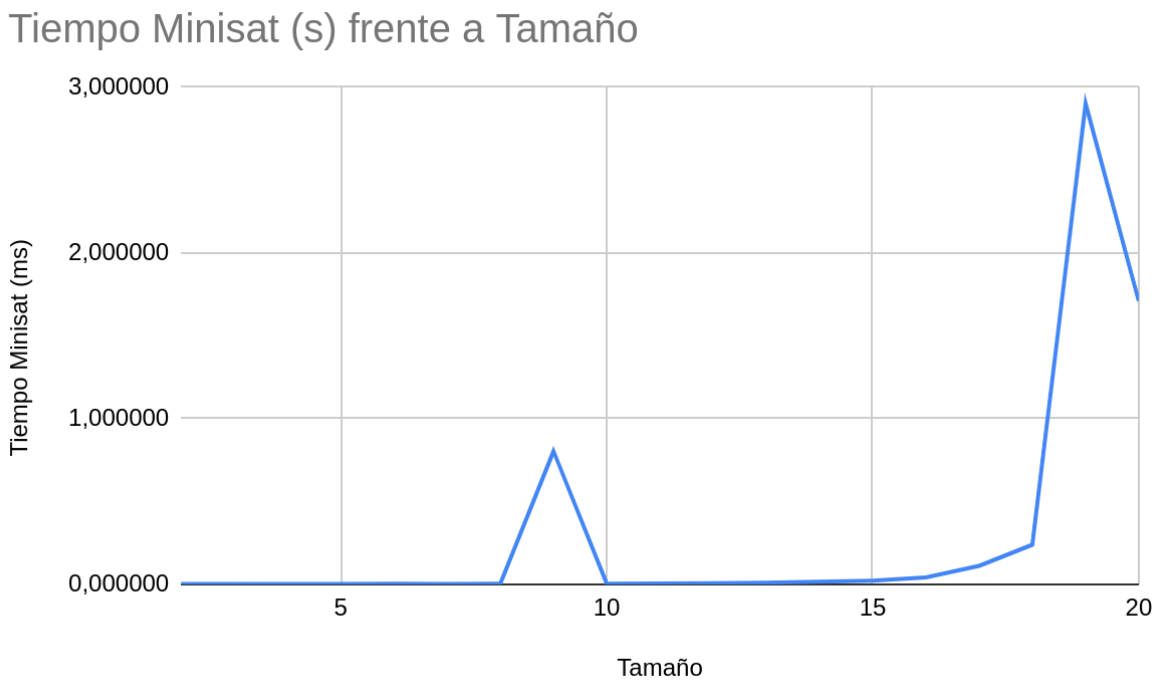

Figura 3: Gráfica de evolución del tiempo de ejecución de Minisat

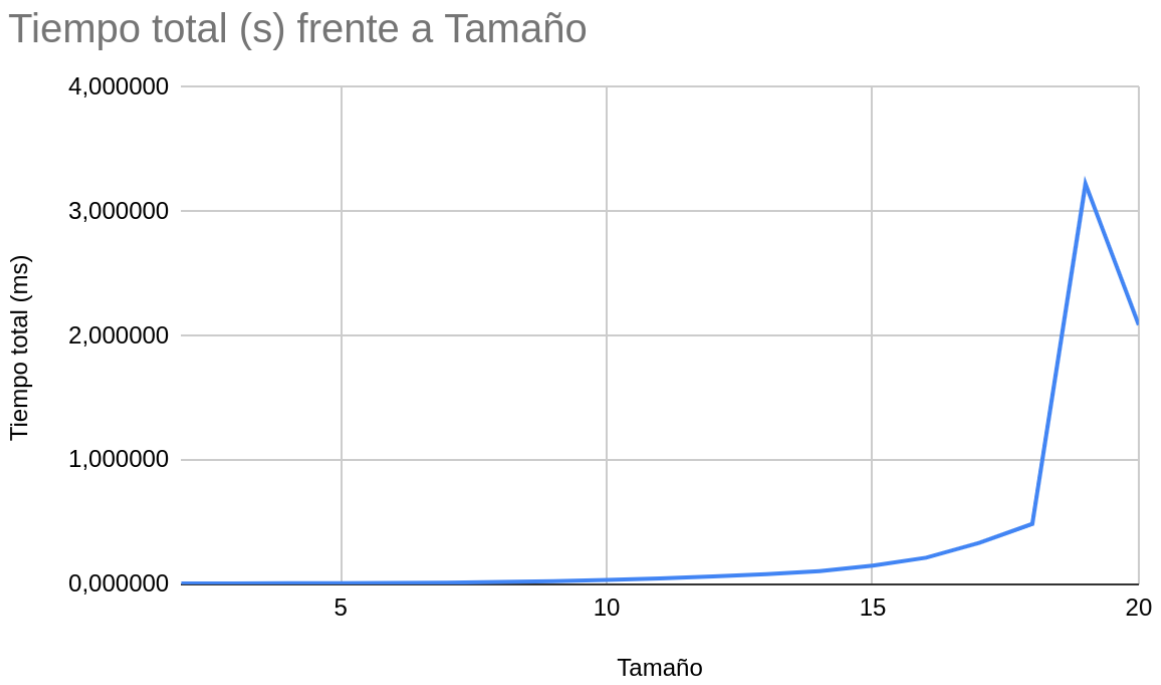

Figura 4: Gráfica de evolución del tiempo de ejecución total del programa

Se puede observar que el uso de memoria de Minisat aumenta de unos 10MB a 17MB, casi el doble para tamaño 20 que para el 2, y se puede apreciar que al final comienza a aumentar la velocidad en la que aumenta dicho valor.

En cuanto al tiempo, se produce un punto de inflexión para cuadrados de tamaño 18 donde el tiempo de ejecución aumenta rápidamente a más de un segundo.

Probabilidad	Memoria (MB)	Tiempo Minisat (s)	Tiempo total (s)
0,1	10,820000	0,003172	0,022700
0,2	10,820000	0,002551	0,022300
0,3	10,820000	0,003572	0,022700
0,4	10,820000	0,004684	0,023500
0,5	10,820000	0,006414	0,025800
0,6	10,820000	0,035387	0,056200
0,7	10,820000	0,209280	0,233300
0,8	10,919000	0,838635	0,863200
0,9	11,959000	1,496231	1,522500

Figura 5: Tabla de resultados para resolver diez tablas diferentes con un número

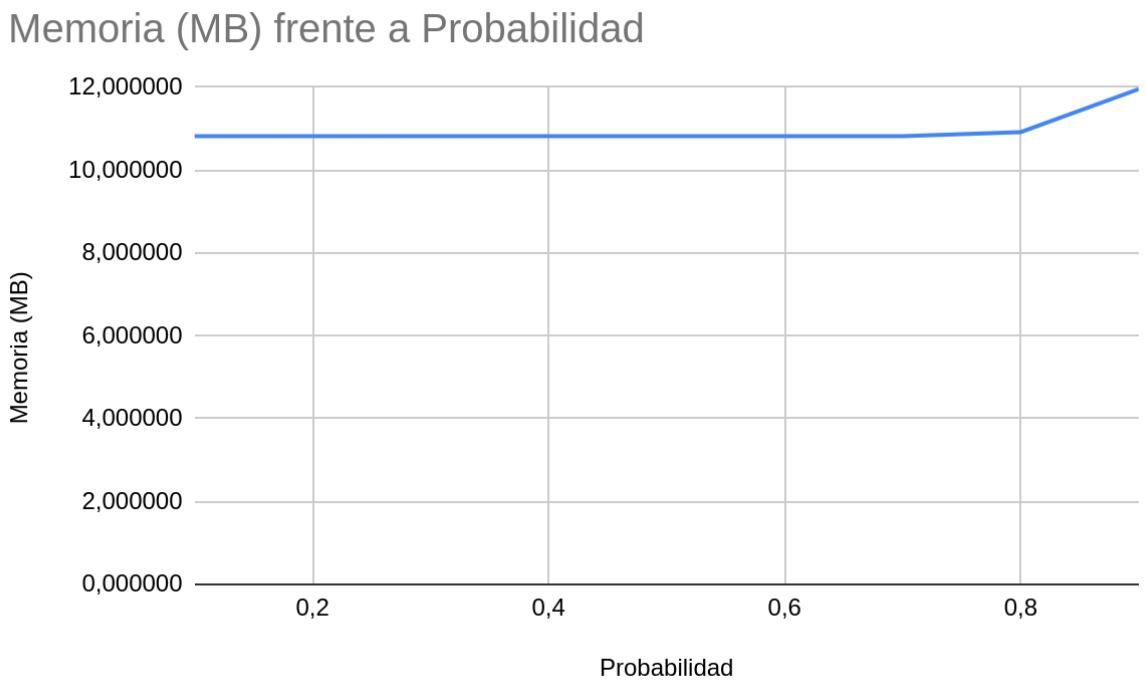

Figura 6: Gráfica de evolución del uso de memoria de Minisat

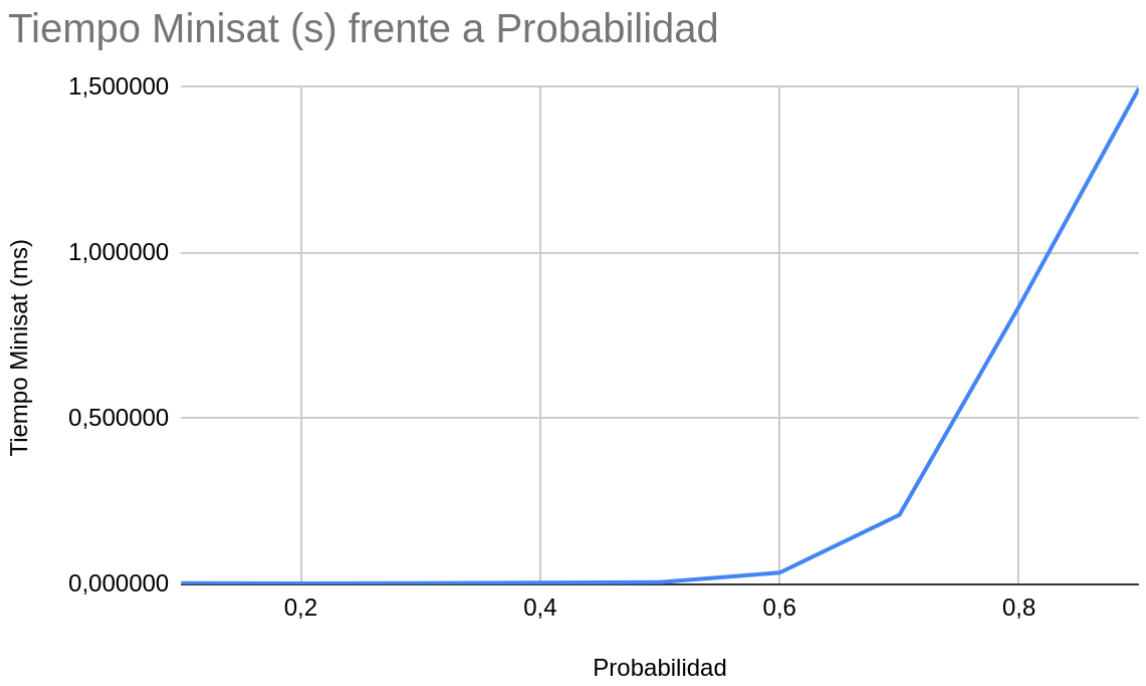

Figura 7: Gráfica de evolución del tiempo de ejecución de Minisat

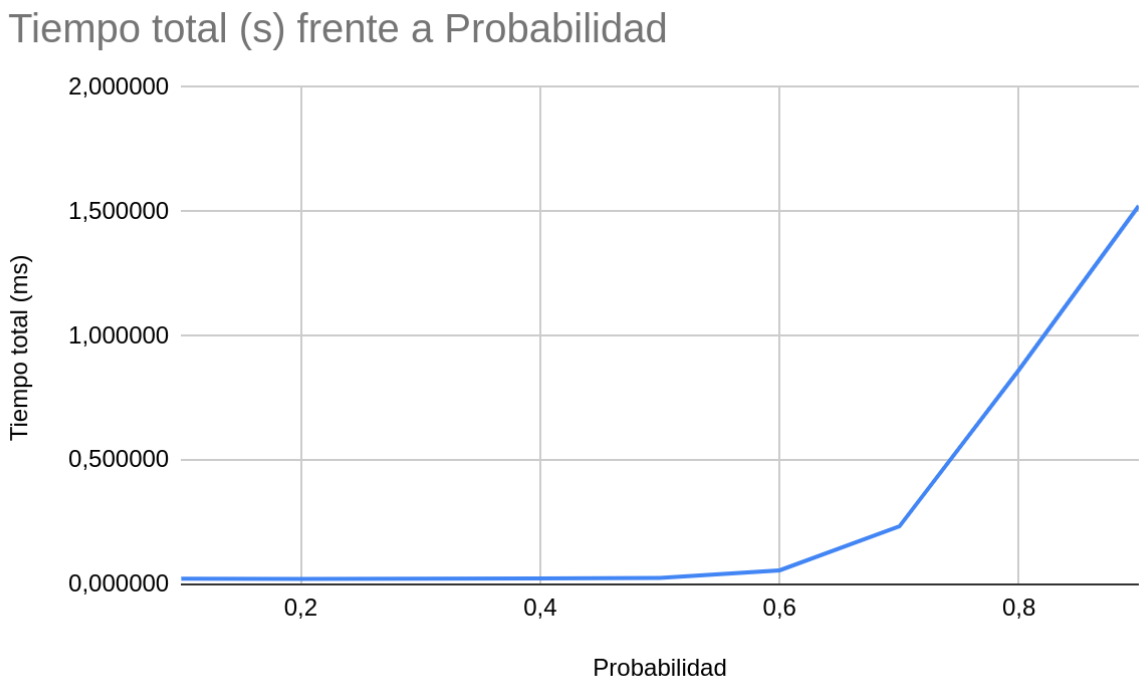

Figura 8: Gráfica de evolución del tiempo de ejecución total del programa

Se puede ver de forma más clara el aumento del tiempo según el número de casillas a resolver. A partir del 70% de casillas en blanco, es mucho más difícil encontrar rápidamente un cuadrado latino, tomando segundo y medio de ejecución para un 90% de casillas en blanco.

Se puede observar como la dificultad del problema aumenta rápidamente a partir de cierto punto. Hay que tener en cuenta que SAT es un problema que se considera intratable (es decir, que su dificultad aumenta en tiempo no polinómico), y además depende de varios factores, como el tamaño y cuantas pistas contiene el tablero de entrada.

Comparación con otros programas de resolución de cuadrados latinos

Para probar nuestra implementación, se ha encontrado una implementación ajena en Internet. Se ha decidido utilizar la siguiente implementación [3] por su similitud del formato de entrada y su facilidad de uso. Está escrito en Python 2 e incluye documentación.

Además, esta implementación resuelve el problema de forma alternativa: en vez de con SAT, utiliza un algoritmo de búsqueda en profundidad (DFS) con algunas optimizaciones como el forward-checking y consistencia de arcos [4].

Adicionalmente, el autor incluye sus propias conclusiones sobre el rendimiento del programa, realiza las siguientes pruebas:

- Por cada tablero de tamaño N = 10 a 100, con K=N agujeros (p.e. un cuadrado de 10x10 con 10 casillas en blanco), obtiene la media temporal de 10 iteraciones.
- Para un tablero 10x10, realiza 50 iteraciones del algoritmo aumentando de 1 a 100 el número de casillas en blanco.

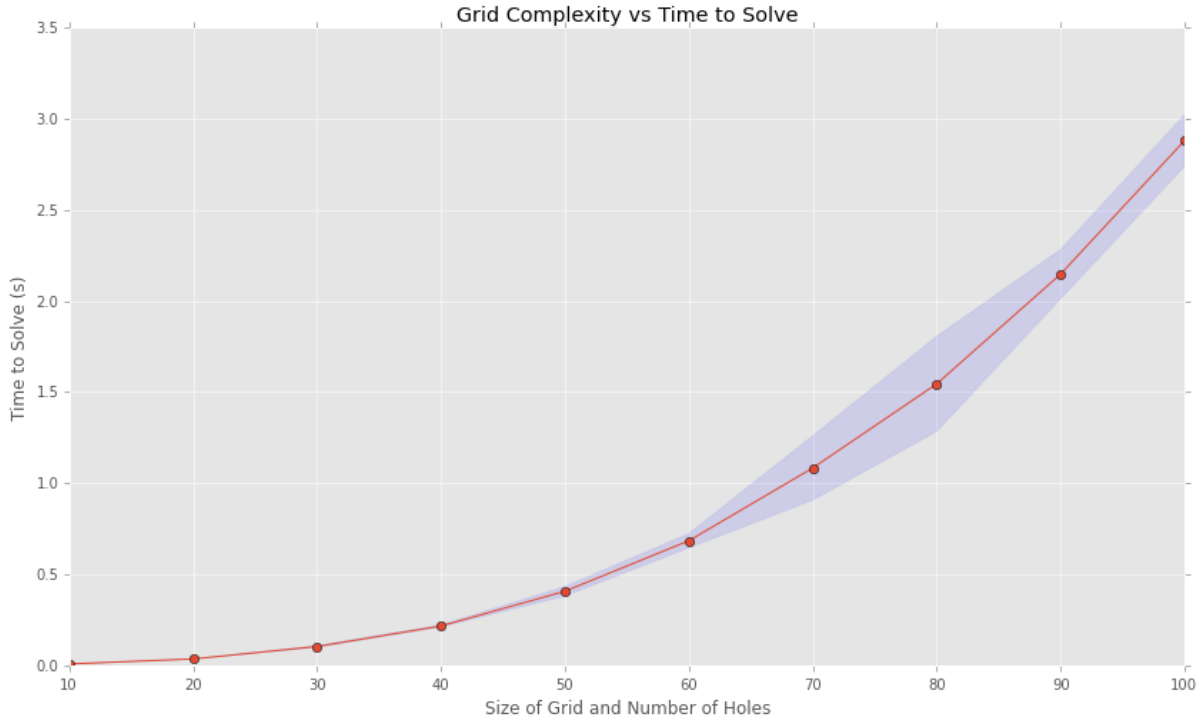

Figura 9: Gráfica de evolución del tiempo de ejecución de la implementación según la variación de tamaño de tablero. Créditos a Nicholas Rutherford

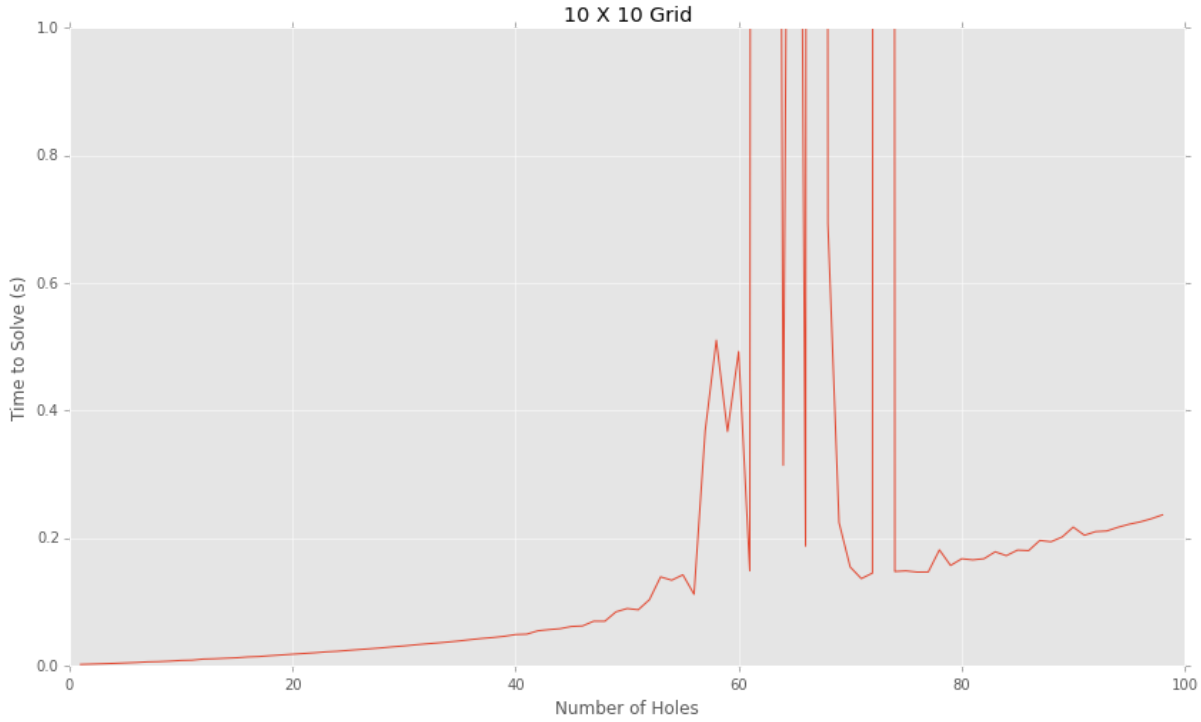

Figura 10: Gráfica de evolución del tiempo de ejecución de la implementación según el número de casillas en blanco en un tablero 10x10. Créditos a Nicholas Rutherford

Aunque no son las mismas pruebas (se ha logrado ejecutar el programa, pero nuestro conjunto de datos no es completamente compatible con el programa y requeriría algunas modificaciones, p.e. indexar valores desde 0 a N-1), son suficientemente parecidas para realizar una comparación de ambos programas.

- En primer lugar, cabe destacar que DFS es un algoritmo que se ejecuta en tiempo en el peor caso $O(b^d)$ (d=profundidad b=factor ramificación). Esto puede dar una ventaja sobre SAT para problemas suficientemente grandes. En nuestro caso, Minisat tiene dificultades para resolver problemas con N>30 siguiendo nuestras pruebas.
- Se puede también observar que el crecimiento del tiempo de ejecución es también mucho más razonable. Incluso siendo DFS un algoritmo lento, tiene cierta ventaja para problemas grandes.
- Finalmente, en problemas de cuadrados con 50% a 80% ambas implementaciones comienzan a tener problemas para encontrar soluciones en tiempo razonable. Aquí nuestra implementación tiene cierta ventaja, con las soluciones de esta implementación tomando hasta 3 segundos por iteración en este rango de valores, más que en nuestro caso.

Distribución del trabajo

	Álvaro	Dorian
Implementación del programa	6h	8h
Realización de pruebas y análisis	7h	12h
Documentación	4h	3h
Total	17h	23h

Bibliografía

[1] https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/solving.html

[2] https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html

[3] https://github.com/nicholasRutherford/LatinSquareSolver

[4] https://latinsquaresolver.readthedocs.io/en/latest/

[5] https://latinsquaresolver.readthedocs.io/en/latest/extraInfo.html#how-well-it-works