Proyecto 1 — Sudoku Solver

David Segura 13-11341 & Amin Arriaga 16-10072

28 de Junio, 2020

Abstract

En este proyecto aplicamos todo el contenido estudiado en la materia Diseño de Algoritmos I dictada por el profesor Ricardo Monascal. **Nota especial:** Gran reconocimiento de agradecimiento al estudiante Amin Arriaga por decidirse a cursar la materia (by D10S).

1 Introducción

Dado un número entero positivo N, un tablero de Sudoku de orden N^2 es una matrix de N^2 filas y N^2 columnas, donde la matriz está dividida en N^2 secciones disjuntas, cada una una matriz de tamaño N x N, el proyecto requiere que se encuentre una solución a dicho Sudoku.

Para conseguir una respuesta al problema debemos reducir la obtención de una solución para el Sudoku a una traducción en **SAT** (*Boolean Satisfiability Problem*), donde se transforman las casillas del Sudoku a una expresión booleana con variables y sin cuantificadores, para saber si hay alguna asignación de valores para sus variables que hace que la expresión sea verdadera. De esta manera si se consigue, podemos traducir esa expresión de nuevo a una solución para el Sudoku y dar el objetivo por cumplido.

2 Desarrollo

2.1 Entrada

Cada instancia será representada por una linea de texto con el siguiente formato:

- Un entero N, tal que $1 \leq N \leq 3$
- Un espacio en blanco
- Una cadena T de N^2 números, donde $0 \leq T_i \leq N$
  - Si $T_i = 0$ la casilla está vacía

– Si $T_i \neq 0$ la casilla tiene el dígito T_i

Por ejemplo, la siguiente cadena representa a nuestro ejemplo de tablero de Sudoku de orden 3.

3 000400009020980030093000840000000000020000806716030000204900065060000300000059010

La entrada del programa será un archivo con tantas líneas como instancias se quieran resolver.

2.2 Reducción de Sudoku a SAT

Es posible reducir cualquier instancia de Sudoku a SAT, representando el estado del tablero con variables y cláusulas de una fórmula en CNF (*Forma Normal Conjuntiva*).

Para la implementación de esta solución representaremos las casillas del tablero de Sudoku con el conjunto de variables $V = [x^1_{0,0}, ..., x^d_{i,j}, ..., x^{N^2}_{N^2-1,N^2-1}]$ con $0 \leq i, j < N^2$ y $0 < d \leq N^2$, donde $x^d_{i,j}$ será verdadera si la casilla (i, j) del tablero tiene el número d.

Hallamos una biyección entre $[1, N^6]$ que representan las variables booleanas en SAT y el conjunto V dada por:

$f : V \to [1, N^6]$

$$f(i, j, d) = i \times N^4 + j \times N^2 + d \qquad (1)$$

$f^{-1} : [1, N^6] \to V$

$$f^{-1}(x) = \left(\left\lfloor \frac{x}{N^4} \right\rfloor, \left\lfloor \frac{mod(x, N^4)}{N^2} \right\rfloor, mod(x, N^2) + 1 \right) \qquad (2)$$

Las funciones (1) y (2) las representamos en los programa *sudoku_to_SAT.py* y *SAT_to_sudoku.py* a través de las funciones `F` y `F_inv` respectivamente. Dichos módulos se encargan de ejecutar la reducción de Sudoku a SAT y viceversa respectivamente, usando `F` para crear las cláusulas del problema SAT a partir de la instancia de sudoku y `F_inv` para traducir la solución del problema SAT a un tablero de sudoku.

Estas cláusulas C vienen dadas por la completitud, la unicidad, la validez y las variables instanciadas.

- **Completitud:** Se crean tantas cláusulas de tamaño N^2 como casillas hay en el tablero, por lo tanto hay N^4 cláusulas de completitud, cada una de la siguiente forma:

`[F(i, j, 1, N),..., F(i, j, d, N),..., F(i, j, N`^{2}`, N)]` para $0 \leq i, j < N^2$, e indican que en la casilla (i, j) debe haber algún valor entre 1 y N^2.

- **Unicidad:** En cada casilla no puede haber al mismo tiempo dos números d y d' con $1 \leq d < d' \leq N^2$. Estas cláusuras quedarian de la siguiente forma:
  `[-F(i, j, d, N), -F(i, j, d', N)]` con $0 \leq i, j < N^2$. Por lo tanto habrán $\binom{N^2}{2} \times N^4 = \frac{N^8 - N^6}{2}$ cláusulas de esta tipo.

- **Instancias del sudoku:** Son las casillas ya asignadas del sudoku, los cuales diferencian cada instancia del tablero. La representación de esta cláusula viene dada de esta manera:
  `[F(i, j, sudoku[i][j], N)]` para $0 \leq i, j < N^2$ si `sudoku[i][j] != 0`, por lo tanto habrán a lo sumo N^4 cláusulas de este tipo.

- **Validez:** Estarán divididas en tres tipos: por filas, por columnas y por secciones. Por fila, tomaremos cada par de casillas distintas y cada dígito d sólo puede estar en una de las dos, análogamente sucede con las columnas. En las secciones k, por cada par de casillas distintas pertenecientes a la misma, no puede haber un dígito que ocupe a ambas. Por lo tanto habrán $3N^4\binom{N^2}{2} = \frac{3(N^8 - N^6)}{2}$ cláusulas de este tipo. Las cláusulas para esta parte son:

  - Para las filas: `[-F(i, j, d, N), -F(i, j', d, N)]` para
  $$0 \leq i < N^2,$$
  $$0 \leq j < j' < N^2 \qquad \texttt{y}$$
  $$1 \leq d \leq N^2$$

  - Para las columnas: `[-F(i, j, d, N), -F(i', j, d, N)]` para
  $$0 \leq i < i' < N^2,$$
  $$0 \leq j < N^2 \qquad \texttt{y}$$
  $$1 \leq d \leq N^2$$

  - Para las secciones: `[-F(i, j, d, N), -F(i', j', d, N)]` para
  $$N\left\lfloor \frac{k}{N} \right\rfloor \leq i \leq i' < N\left\lfloor \frac{k}{N} \right\rfloor + N\,,$$
  $$N mod(k, N) \leq j, j' < N mod(k, N) + N\,,$$

$$1 \leq d \leq N^2$$
$$0 \leq k < N^2 \qquad \text{y}$$
$$(i, j) \neq (i', j')$$

.

Al final tendremos que solucionar una problema SAT con $|V| = N^6$ variables y $|C| = |completitud| + |unicidad| + |validez| + |instancia| = N^4 + \frac{N^8 - N^6}{2} + \frac{3(N^8 - N^6)}{2} + I = 2N^8 - 2N^6 + N^4 + I$ cláusulas, donde $0 \leq I = |instancia| \leq N^4$.

La función `sudoku_to_SAT` recibe una matriz que representa la instancia del sudoku y retorna un string con la representación en SAT de la instancia del sudoku:

```
c Ejemplo de salida para sudoku_to_SAT
c
p cnf V C
v1 -v2 0
v3 -v4 0
v2 0
.
.
.
```

La entrada de `sudoku_to_SAT` viene dada por la función `read_sudoku`, la cual recibe la instancia del sudoku en string y devuelve una matriz $N \times N$, donde implementamos la extensión permitiendo que $1 \leq N \leq 6$.

Dentro de la función `sudoku_to_SAT` hacemos uso de las funciones `F` y `plus`:

`F`: es la función (1) antes mencionada en 2.2, usada para obtener las transformaciones correspondientes.

`plus`: recibe la posición de una casilla (i, j), la sección k y el grado N; y retorna la posición de la siguiente casilla dentro de la misma sección k mediante las siguiente formulas:

$$D = N \times mod(k, N)$$
$$i = i + \left\lfloor \frac{j+1}{D+N} \right\rfloor$$
$$j = mod(j+1, D+N) + D \left\lfloor \frac{j+1}{D+N} \right\rfloor$$

Esto funciona de la siguiente manera: Si el valor de la siguiente columna $j+1$ alcanza al desplazamiento por sección D mas el grado del sudoku N, significa que debemos pasar a la siguiente fila por lo que se le aumenta 1. En caso contrario no se le suma nada. Para las columnas, el término de la izquierda se encarga que la columna no supere el desplazamiento más el grado del sudoku, que corresponderia a la última columna de la sección, más 1. Sin embargo, dejando solo este término la columna regresaría a 0 al alcanzar dicho máximo, por lo tanto, el término de la derecha notemos que es el mismo que el de i multiplicado por el desplazamiento, esto significa que al superar la máxima columna, el término de la izquierda es 0, y el de la derecha es 1 por D, correspondiente a la primera columna de la sección.

Tiempo Asintóstico:
Si n es la dimensión de la matriz del sudoku, es decir, $n = N^2$, entonces:

- `sudoku_to_SAT` $\in O(n^4)$, debido a que estamos recorriendo una matriz ($O(N^2 \times N^2)$), comparando para cada par de casillas por fila/columna excepto consigo mismo ($O(N^2)$) para cada digito ($O(N^2)$), quedando $N^2 \times N^2 \times N^2 \times N^2 \in O((N^2)^4)$.
- `read_sudoku` $\in \mathrm{O}(n)$, ya que debe recorrer el string de la instancia del sudoku digito por digito para asignarle su casilla correspondiente en la matriz.
- `F` $\in O(1)$.
- `plus` $\in O(1)$.

Memoria Asintótica:
Si n es la dimensión de la matriz del sudoku, es decir, $n = N^2$, entonces:

- `sudoku_to_SAT` $\in O(n^4)$, pues se almacenan todas las cláusulas del sudoku, y como vimos anteriormente, hay $O(N^8) = O(n^4)$ cláusulas.
- `read_sudoku` $\in O(n)$.
- `F` $\in O(1)$.
- `plus` $\in O(1)$.

2.3 SAT Solver

El Problema de Satisfacibilidad Booleana o SAT consta de intentar hallar una asignación booleanas para las variables, que hagan que la fórmula en lógica proposicional sea satisfecha (evalúe a *True*).

Ya con el sudoku transformado en un problema SAT en formato CNF, implementamos una función llamada `read_SAT` que recibe el problema en el formato mencionado y retorna dos arreglos, uno de vectores que dentro del programa tratamos como V, el cual contiene valores inicializados en 0 para efectos más prácticos al momento de resolver el problema. No confundir el valor 0 de una variable con *False*, ya que nosotros consideramos *False* como -1 y *True* como 1, por lo que 0 se considera no inicializado. La longitud de este arreglo será la cantidad de variables encontradas en el problema SAT donde cada posición en el vector representará a la variable en sí. El otro arreglo que devuelve son las cláusulas que formamos al momento de traducir la instancia del sudoku.

Con estas dos estructuras de datos formadas, pasamos a la función estrella del programa, la llamamos `laura_SAT` 😎, es una función recursiva que se compone de dos funciones importantes: `update_C` y `verify_units`. Hablemos primero de estas dos funciones para entender el funcionamiento de `laura_SAT`:

- `update_C`: esta función recibe 3 parametros, V que corresponde a las variables y su valor lógico, C que son las cláusulas y k que es una variable correspondiente a la posición $k-1$ de V. El objetivo es actualizar las cláusulas con respecto a la variable k siguiendo las siguientes reglas:

  - Si el literal encontrado en la cláusula es *False*, se elimina dicho literal.
  - En caso contrario, si es *True* se elimina toda la cláusula a la que pertenece.

  Si al actualizar las cláusulas alguna de ellas quedó vacía, significa que se hizo una mala asignación de literales y por ende la función retorna *True*, que en el ambiente de `laura_SAT` significa que hubo conflicto, de lo contrario la actualización resulto exisota.

- `verify_units`: Esta función recibe las variables V y las cláusulas C y retorna un valor booleano que indica si hubo algún conflicto. En este método se realiza la verificación de cláusuras unitarias y en caso de haberlas actualiza las variables en consecuencia, colocándoles el mismo signo que posee en la cláusula, de esta manera se satisfacerán las clausulas unitarias, y si encuentra que ya la variable había sido asignada con el signo contrario, esta devuelve conflicto. Después que encuentra todas las cláusulas unitarias, realiza un actualización de cláusulas mediante `update_C`, donde si da conflicto es porque se hizo una mala asignación de variables antes.

- También está la función `search_amin_zero`, que dado un arreglo de variables, buscará la menor posición tal que en el arreglo haya un `0`.

- `laura_SAT`: Basada en el algoritmo de *BFS*, selecciona la primera variable que aun no haya sido asignada (`search_amin_zero`), se le asigna el valor de `-1` (`False`), se actualizan las clásulas (`update_C`) y luego se verifica si quedaron cláusulas unitarias (`verify_units`). En caso de no haber quedado un conflicto (alguna cláusula vacía) se verifica si aún quedan cláusuras, si no es así, significa que conseguimos una asignación de variables exitosa y retornamos la asignación actual (sustituyendo los 0 por -1) junto a `False` (que indica que no hubo conflictos). Si aún quedan cláusulas, se realiza una llamada recursiva a `laura_SAT` con las cláusulas y variables actuales, si esta llamada retorna un arreglo junto a `False`, significa que se consiguió una asignación correcta de variables en una rama posterior y se retorna el mismo arreglo junto con `False`, en cambio si se retorna un arreglo junto a `True`, significa que ninguna rama posterior logró conseguir una solución. Luego, si hubo un conflicto con `update_C`, `verify_unit` o en la llamada recursiva, se le asigna el valor de `1` (`True`) a la variable `k` y se repite el mismo proceso. En caso de que vuelva a dar conflicto, se retorna un arreglo con longitud N^2 inicializados en 0 junto a `True`, indicando que en esta rama no hay solución. Si la primera llamada a `laura_SAT` retorna un arreglo junto a `True`, significa que la fórmula es insatisfacible, en cambio si retorna un arreglo junto a `False`, significa que la asignación booleana representada por el arreglo da un valor de `True` en la fórmula. Se asigna primero -1 y luego 1 a las variables, pues para el sudoku, hay más variables falsas que verdaderas.

- `output`: Recibe el conjunto de las variables y el resultado del problema en SAT y devuelve el string en formato CNF.

Tiempo Asintóstico:

- `read_SAT`: Sea c el número de comentarios que en el formato CNF del problema SAT y ℓ la cantidad de literales del problema SAT, entonces `read_SAT` $\in O(c+\ell)$.

- `update_C`: Se recorre el arreglo de cláusulas para actualizarla, si n es la cantidad de cláusulas, entonces `update_C` $\in O(n)$.

- `verify_units`: Se recorre el arreglo de cláusulas para encontrar las cláusulas unitarias. Tomemos n como el tamaño de dicho arreglo. Existen dos posibles peores casos: si todos las cláusulas son unitarias y contienen variables distintas, se recorreran todas a lo sumo n veces, lo cual es $O(n)$, luego, se llamará n veces a `update_C`, una por cada variable, y como `update_C` $\in O(n)$, entonces `verify_units` será $O(n+n^2) = O(n^2)$. El otro caso es si hay n cláusulas, donde la i-esima tiene $n-i$ literales, con $0 \leq i < n$, siendo así la ultima cláusula unitaria, y la i-ésima cláusula es un supra-conjunto de la $i+1$. Entonces cuando se verifique si hay cláusulas unitarias lo hará en n iteraciones. Cuando haya encontrado la cláusula unitaria aplicará `update_C`, eliminando el literal de todas las cláusulas, dejando a la $n-2$-ésima como unitaria y se repetirá el ciclo. Por lo tanto, endríamos n ciclos en los que se buscan las cláusulas unitarias $O(n)$ y se llama una vez a `update_C` (pues en cada ciclo solo se consigue una cláusula unitaria), en total serian $O(n \times (n+n)) = O(n^2)$ operaciones. Así tenemos cotas iguales para ambos peores casos, concluyendo que `verify_units` $\in O(n^2)$.

- `laura_SAT`: Notemos que si V_0 es el número de variables sin asignar, entonces a lo sumo se realizarán 2^{V_0} llamadas recursivas a `laura_SAT`. Sin embargo, esa es una cota muy pesimista para el caso del sudoku. Calcularemos cual es la complejidad promedio (suponiendo que el tablero es satisfacible):

  Gracias a las cláusulas de instancia, hay variables a las que se les puede asignar un valor desde la primera iteración. Si escogemos un número aleatorio entre 1 y N^4 correspondientes a escoger aleatoriamente cuantas casillas tienen un valor predefinido, entonces en promedio tendriamos que se asignan $\frac{N^4}{2}$ casillas, por lo tanto, gracias a las cláusulas de unicidad, también quedan asignadas $\frac{N^6}{2}$ variables en promedio. En particular, la probabilidad de que luego de asignar la mitad de las variables, aún quede incertidumbre del valor de las casillas, es decir, que hayan varias formas de completar el sudoku a partir de la instancia original son muy bajas. Por lo tanto, probablemente se podría resolver el sudoku con sólo una iteración de `laura_SAT`, luego, toda la complejidad recae en `verify_units`, que como vimos anteriormente es $O(C^2)$, como hay $C = O(N^8)$ cláusulas, entonces la complejidad de texttverify_units y por lo tanto de `laura_SAT` es $O(N^{16})$ en el caso promedio del sudoku satisfacible.

Debido a que en el peor caso, que es cuando el tablero esta vacío, la incertidumbre es máxima, entonces la cantidad de elecciones que tiene que hacer el algoritmo también, por lo tanto, la cantidad de combinaciones que tenga que probar antes de conseguir la solución serán muchas. Luego, como tenemos N^6 variables, y cada variable solo puede tener 2 valores, entonces la cota en este caso es $O(2^{N^6})$.

- `output`, `search_amin_zero` $\in O(V)$ siendo V el número de variables del problema.

Memoria Asintótica:

- `read_SAT` $\in O(n^4)$, correspondientes a las $O(N^8)$ cláusulas que hay que almacenar.
- `update_C` $\in O(1)$.
- `verify_units` $\in O(C)$, siendo C el número de cláusulas, pues en el peor caso, se almacenan todas las cláusulas.
- `laura_SAT` En el peor de los casos, que es cuando se prueban muchas combinaciones de variables, al llegar a una asignación de variables completa, es decir, que todas las variables tienen un valor, y que cada valor fue asignado por elección del algoritmo y no por `verify_units`, significa que tendríamos una recursión de profundidad N^6 correspondientes al número de variables asignadas. Luego, por cada llamada recursiva, se hace una copia de las variables y las cláuslas, las cuales son $O(N^8)$. Por lo tanto, se usa memoria en $O(N^14)$ en el peor caso. En el caso promedio, como solo se realiza una llamada recursiva, entonces se usa memoria $O(N^8)$.
- `output` $\in O(V)$ siendo V el número de variables.
- `search_amin_zero` $\in O(1)$

2.4 SAT a Sudoku

Ya en esta etapa del proyecto el mayor problema está resuelto, solo queda transformar las soluciones obtenidas al formato requerido. Esto lo hacemos mediante dos funciones:

- `F_inv`: esta función ya la definimos en 2.2 como (2), el cual recibirá un valor que corresponde a una variable y retornará una posición en la matriz (fila y columna) y el valor contenida en ella.

- `SAT_to_sudoku`: recibe un arreglo de variables que corresponden a la solución y las convierte en un string en formato de una linea, por ejemplo

3 673815429145279836982634571419563782367428915258791364521987643834156297796342158

Tiempo Asintóstico:

- `F_inv` $\in O(1)$.

- `SAT_to_sudoku` $\in O(V)$ siendo V el número de variables del problema.

Memoria Asintótica: FALTAA

- `F_inv` $\in O(1)$

- `SAT_to_sudoku` $\in O(V)$ siendo V el número de variables del problema.

2.5 ZCHAFF

ZCHAFF es un resolvedor conocido para SAT, con tiempos de ejecución muy competitivos. Usa el mismo formato de entrada aquí expuesto y reporta sus soluciones con el mismo formato de salida. Con este programa compararemos nuestros resultados obtenidos.

2.6 Salida

En `sudoku_solver.py` orquestamos todos los programas que implementamos para un funcionamiento completo. Además requerimos implementar algunas funciones más para poder dar unos resultados mejor elaborados, donde para cada instancia de sudoku resuelta, se le añade el tiempo que tardó nuestra implementación `laura_SAT`. Todo esto se reporta mediante el terminal, donde se generará también un archivo .txt con las instancias de los sudokus resuelta. Al final de la ejecución se mostrará un gráfico en comparando nuestro algoritmo con el ZCHAFF. Las funciones que usamos en nuestro programa de salida son:

- `timer`: usamos la librería *multiprocessing* para llevar acabo el multiprocesamiento a través de hilos, donde ejecutamos la función a procesar en un hilo h_1 y el otro hilo h_2 se encarga de estar activo durante un tiempo limite indicado. Cuando h_1 o h_2 finalicen, se termina la ejecución del otro hilo. Recibe la función a ejecutar en h_1 y sus argumentos, y el tiempo límite a establecer donde tenemos por defectos que son 10 segundos.

- `get_solution`: simplemente guadará en una cola el resultado de ejecutar `laura_SAT`. Recibe las entradas de `laura_SAT` que son las variables y las cláusulas.

- `sudoku_solver`: es la función que se encarga de recibir las instancias del sudoku y el tiempo limite, y retornar los resultados. En esta función es donde manejamos todas las funciones anteriores para encontrar la solución al problema planteado. Tiene como nombre `sudoku_solver` pero debió haberse llamado `dudamel`.

Tiempo Asintóstico:

- `timer` $\in O(min(O(F), t)$ siento t el tiempo limite establecido y F la función a ejecutar.
- `get_solution` $\in O($`laura_SAT`$)$.
- `sudoku_solver` $\in O($`laura_SAT`$)$.

Memoria Asintótica:

- `timer` $\in O(F)$
- `get_solution` $\in O($`laura_SAT`$)$.
- `sudoku_solver` $\in O($`laura_SAT`$)$.

3 Conclusiones