Proyecto 2 — Sudoku Solver Mejorado

David Segura 13-11341 & Amin Arriaga 16-10072

17 de Julio, 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

Esta implementación es una continuación del anterior proyecto. Esta etapa consiste en la mejoría del tiempo de ejecución en el resolvedor de **SAT** mediante estrategias que explicaremos posteriormente.

2 Mejoras de SAT

2.1 Idea presentada en el Proyecto

La propagación unitaria propuesta por los profesores del curso en el enunciado del proyecto, es un concepto que ya estábamos aplicando en nuestra implementación para la primera parte. Sin embargo, se desarrollaron dos nuevas estructuras para la optimización de la ejecución: `Variable`, `Closure` y `CNF`.

- `Variable`: Se implementó esta clase para tener una estructura de datos que almacene los datos correspondiente a una variable, los cuales son: las clausulas en las que la variable aparece, el signo que tiene asignado y un *backup* donde se guardan el signo del literal en distintos estados de ejecución.

  Los métodos que esta clase tiene son:

  - `__init__`: Inicializa la instancia con `sign = 0`, `closures = []` y `backup = {}`
  - `assign`: Se encarga de asignarle el signo al literal.
  - `add_closure`: Agrega una clausula a la información del literal.

- `save`: Guarda una copia del signo actual.

  `restaure`: Método que restablecerá una copia guardada mediante la llave indicada, eliminando dicha información del backup para no acumularla. Se utilizó un diccionario y no una pila de eventos pues, por alguna razón, con la pila no se almacenaban/restauraban los datos en el orden correcto, mientras con la llave, podemos estar seguros de que el estado a restaurar será el correcto.

Todos los métodos tienen complejidad en tiempo y memoria de $O(1)$.

- `Closure:` Se implementó esta clase para tener una estructura de datos que almacene los datos correspondiente a una clausula, los cuales son: los literales que pertenecen a la misma, el tamaño de la clausula, una variable que indica si fue satisfecha (si ya tomó valor de *True* mediante alguno de sus literales), y un *backup* donde se guardan toda esta información ya mencionada en distintos estados de ejecución.

  Los métodos que esta clase tiene son:

  - `__init__`: Inicializa la instancia con `literales = []`, `N = 0`, `satisfied = False` y `backup = {}`.
  - `add`: Agrega un literal a la clausula.
  - `delete`: Se encarga de eliminar los literales indicados pertenecientes a la clausula. Devuelve un booleano que indica si la clausula quedó vacía o no.
  - `save`: Guarda una copia de toda la información almacenada en la clausula.

    `restaure`: Restablecerá una copia guardada mediante la llave indicada, eliminando dicha información del backup para no acumularla. Se uso un diccionario y no una pila de eventos por las mismas razones indicadas en la descripción de la clase `Variable`.

  Tiempo Asintóstico:

  - `__init__`, `add`, `save`, `restaure` $\in O(1)$.
  - `delete` $\in O(d)$, siendo d la cantidad de elementos a eliminar.

  Memoria Asintótica:

  - `add`, `delete`, `restaure` $\in O(1)$.
  - `save` $\in O(l)$, siendo l el número original de literales en la cláusula.

- `CNF:` Se implementó esta clase para almacenar todas las cláusulas del problema y el número de cláusulas que quedan sin satisfacer. El principal objetivo de esta clase es no tener que verificar cada cláusula para decidir si el problema ya fue resuelto o no. Al igual que las clases anteriores, contiene un backup, pero almacena únicamente el número de cláusulas que tenía en ese momento.

  Los métodos que esta clase tiene son:

- __init__: Inicializa la instancia con `closures = closures`, `N = 0` y `backup = {}`, siendo *closures* una lista de lista de cláusulas, donde si una cláusula se encuentra en la i-ésima lista, significa que tiene $i + 1$ literales. A esta lista de lista de clásusulas la denominaremos `C` el resto del informe.
- `save`: Guarda una copia del número de cláusulas en ese momento.

  `restaure`: Restablecerá una copia guardada mediante la llave indicada, eliminando dicha información del backup para no acumularla. Se uso un diccionario y no una pila de eventos por las mismas razones indicadas en la descripción de la clase `Variable`.

Todos los métodos tienen complejidad en tiempo y memoria de $O(1)$. Podría parecer que `__init__` $\in O(|C|)$, sin embargo, solo se guarda una referencia a dicho arreglo, no una copia.

Cabe destacar que en nuestra implementación se usará la misma llave (key) para el backup de las variables, de `C` y de la instancia de `CNF` en un estado determinado del problema.

2.2 SAT Solver

Las funciones usadas por nuestro SAT solver original fueron modificadas para mejorar la eficiencia en la ejecución `laura_SAT`:

- `update_C`: A diferencia de antes donde teniamos que buscar en los literales de todas las cláusulas alguna instancia de la variable especificada, ahora simplemente revisamos las cláusulas a las que la variable apunta, por lo tanto nos ahorramos dicha búsqueda. Al principio se verifica si la cláusula es satisfecha, en caso de ser asi se pasa a la siguiente cláusulas. Esta verificación es necesaria pues los apuntadores de las variables hacia las distintas cláusulas donde aparecen nunca son modificados, incluso si la cláusula es satisfecha. Esto nos ayuda a no tener que realizar un backup de todas las referencias a las cláusulas pues estas son invariantes.

  Luego, se revisan los literales de cada cláusula hasta que no quede ninguno o hasta que alguno de `True`. Si ocurre el primer caso, todos los literales falsos serán eliminados de la cláusula y se actualizará su posición en `C`. Si la cláusula queda vacía, significa que dio `False`, y por lo tanto se retorna `True` indicando un conflicto. Mientras que para el segundo caso, se elimina la referencia a la cláusula en `C` y se coloca como satisfecha. Si ninguna cláusula da conflicto, se retorna `False`.

- `verify_units`: Gracias a `C`, la estructura de esta función es mucho más simple, pues simplemente tenemos que verificar `C[0]` para saber si hay cláusulas unitarias, en vez de buscar entre todas las cláusulas. Así, mientras `|C[0]|>0`, significa que aún quedan cláusulas unitarias, se toma una, se coloca como satisfecha y se le asigna True a su literal. Luego, se llama a la función `update_C` para actualizar las cláusulas afectadas.

- `search_amin_zero`: Funciona exactamente igual que antes, sólo que ahora funciona con arreglos de instancia de la clase `Variable` y no de enteros.
- Se crea el procedimiento `restaure` que, dada una llave, restaura el estado de las variables, de `C` y de la instancia de `CNF` al momento en que se les realizó un backup usando dicha llave.
- `laura_SAT`: Ahora la verificación de cláusulas unitarias se realiza justo al principio de la función. En caso de que no de conflicto y el problema aún no esté resuelto, se elegirá una llave tal que no esté siendo usada en el backup de la primera variable, recordemos que todas las instancias a las distintas clases comparten las mismas llaves. Luego, se guardará el estado actual de las variables y las cláusulas. El resto de la función es practicamente igual, con la diferencia de que se restauran los valores de las variables y las cláusulas cuando ocurre un conflicto con la asignación `True` de la variable por el bactracking. No se restaura el estado con la asignación `False` pues en caso de dar conflicto, la función retorna un valor; si la función es la llamada originalmente significa que no hay solución, así que no vale la pena restaurar el estado, y si es una llamada recursiva, el estado será restaurado en la llamada anterior.

Tiempo Asintóstico:

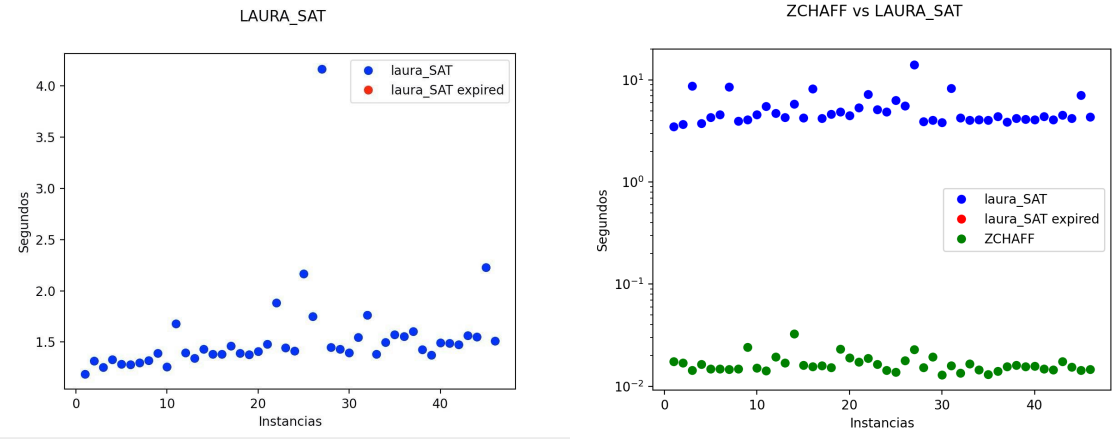
- `update_C` $\in O(|C| * l_{max})$, siendo $|C|, l_{max}$ el número de cláusulas y el máximo entre el número de literales de las cláusulas respectivamente.
- `verify_units` $\in O(|V| * O(\texttt{update\_C})) = O(|V| * |C| * l_{max})$. Esto se debe a que `verify_units` llama a `update_C` por cada cláusula unitaria, sin embargo, no puede encontrar más de $|V|$ cláusulas unitaria sin obtener un conflicto, pues significa que se realizaron más de $|V|$ asignaciones a las $|V|$ variables.
- `search_amin_zero` $\in O(|V|)$ siendo $|V|$ el número de variables.
- `restaure` $\in O(|V| + |C|)$.
- `laura_SAT`: Siguiendo la misma lógica al calcular la complejidad de `laura_SAT` en el proyecto 1, una cota superior es obviamente $O(2^{|V|})$, sin embargo, para el caso promedio del sudoku, toda la complejidad recaerá en `verify_units`, que en ese caso sería $O(|V| * |C| * l_{max})$.

Memoria Asintótica:

- `update_C` $\in O(l_{max})$.
- `verify_units` $\in O(1)$.
- `search_amin_zero` $\in O(1)$.
- `restaure` $\in O(1)$.
- `laura_SAT`: El peor caso ocurre cuando todas las variables son asignadas por decisión del backtraking y no de `verify_units`, pues se haría un backup del estado actual en cada asignación. Luego, en cada asignación se usa memoria $O(|V| + |C| * l_{max})$. Por lo tanto, como hay $|V|$ variables, entonces la memoria asintótica es $O(|V|^2 + |V| * |C| * l_{max})$.

3 Conclusiones

Para concluir, realizamos la comparativa de nuestra implementacioón actual de `laura_SAT` con la anterior, el cual dio una mejora bastante considerable. Los resultados que obtuvimos en este proyecto y en el anterior fueron los siguientes:



Donde el peor tiempo fue de poco más de 4 segundos y el promedio 1.5 segundos, a diferencia del proyecto anterior en el que el peor caso fue de alrededor de 11 segundos y el promedio fue de 4 segundos, lo que supone una ejecución 3 veces más rápida aproximadamente. Esto se debe a que se evitaron muchas búsquedas entre las cláusulas, ya sea para verificar si todas están vacías, buscar las unitarias o las instancias de las variables, y todo gracias a los apuntadores que hay entre cláusulas y variables.