Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 – Diseño de Algoritmos I
Intensivo Junio-Julio 2020

Proyecto
Resolvedor *Eficiente* de Sudoku

¡Eureka! Los *"estudiantes de diseño de algoritmos I"* lograron reducir el antiguo tablero de Sudoku a una instancia de SAT. Sin duda, éste es el evento más importante de la historia reciente en Titán. La algarabía y el júbilo se apoderan de la plaza *Github*, en donde los ahora afamados entusiastas mostrarán sus resultados al público en tan sólo unos minutos. ¡Hoy es un gran día para la humanidad!

Nadie se pierde el espectáculo. Hasta el más escéptico se une a la congregación, una chispa de curiosidad y esperanza en sus corazones. Todos estamos más que ansiosos por redescubrir los secretos olvidados de nuestro pasado y, por primera vez en mucho tiempo, parece que seremos capaces de hacerlo.

Con un estruendo de aplausos y silbidos, los *"profesores"* (título que tienen los líderes de los *"estudiantes de diseño de algoritmos I"*) aparecen orgullosos en la tarima. Esperan unos minutos a que la audiencia se calme, un silencio de suspenso e intriga dibujando cada esquina de esta histórica plaza. Finalmente, rompen el silencio e inician la ceremonia al canto de *"git pull"*, como es costumbre en una ocasión como ésta. ¡La muestra ha dado inicio! Uno a uno empieza la ejecución de los programas. Resolvedores de SAT de toda clase atacan el tablero traducido, buscando aquella esquiva verdad que nos fue arrebatada por el tiempo. ¡Estamos tan cerca de resolver el acertijo!

Poco nos dura la emoción, sin embargo. Con cada minuto que pasa, la tensión aumenta y los ánimos se enervan. ¿Por qué tardan tanto? ¿Qué ocurre?

Después de horas, la audiencia comienza a mermar y los rumores empiezan a fluir. Los profesores, viendo que pierden la confianza de la población en que han resuelto el problema, llaman a todos de vuelta para un anuncio especial. Debo admitir que me siento renuente a volver, pero sé que finalmente me vencerá la curiosidad.

Con la atención de todos prestada, al menos por unos minutos, proceden a recordar a todos sobre la existencia de una antigua leyenda. En épocas ya olvidadas, explica uno de los profesores, existían resolvedores de SAT tan rápidos que hubieran podido resolver problemas como éste en menos de unos segundos. El otro profesor muestra al público una colección de pergaminos con un sello que lee *"papers"*.

¿Qué habrá en esos pergaminos? Los profesores dicen que contienen pistas para recrear los antiguos resolvedores. Nos prometen estudiar los pergaminos, implementar sus secretos y volver a intentar resolver el tablero de Sudoku. Aseguran que podrán lograrlo en una semana. ¿Ustedes qué creen? Ya lograron su objetivo una vez, transformando el tablero de Sudoku en una instancia de SAT. Ahora quieren resolver ese SAT rápidamente, usando las técnicas olvidadas de nuestros ancestros. ¿Podrán lograrlo?

1 Mejoras para SAT

Una implementación inocente para SAT seguramente hará *backtracking* sobre los posibles valores de verdad de las variables.

Notemos que si hay n variables, esto se traduce a 2^n posibilidades. Para valores de n relativamente pequeños, esta estrategia es impráctica. Sin embargo, podemos mejorar mucho nuestra solución usando algunas observaciones claves.

Aquí podrán ver una técnica explicada en detalle, pero no es la única disponible. Los artículos que se encuentran disponibles con este enunciado tienen algunas técnicas adicionales que mejorarán aún más su resolvedor. También pueden investigar más allá de estos artículos e incorporar tantas técnicas como crean sea útil para mejorar sus implementaciones.

1.1 Propagación unitaria

Recordemos que nuestras fórmulas estarán en forma normal conjuntiva. Esto es, serán conjunciones de cláusulas y cada cláusula será disyunciones de literales. Los literales, a su vez, pueden ser una variable o la negación de una variable.

Por ejemplo:

$$\phi = p \wedge (\neg q \vee \neg p) \wedge (q \vee p)$$

tiene tres cláusulas:

1. p
2. $\neg q \vee \neg p$
3. $q \vee p$

Decimos que una cláusula es unitaria, si nada más contiene un literal. En el ejemplo anterior, la primera cláusula es unitaria (sólo tiene el literal p). Estas cláusulas reducen el tamaño de nuestro problema, pues fijan el valor de una variable.

En el ejemplo anterior, es necesario que $p \equiv \mathit{true}$, por lo que podemos reescribir la fórmula:

$$\phi = \mathit{true} \wedge (\neg q \vee \neg \mathit{true}) \wedge (q \vee \mathit{true})$$

Aprovechando que true es neutro de la conjunción y absorbente de la disyunción, obtenemos:

$$\phi = \neg q \vee \neg \mathit{true}$$

Sabiendo que $\neg true \equiv false$ y que $false$ es neutro de la disyunción, obtenemos:

$$\phi = \neg q$$

Notemos que ahora ϕ tiene una sola cláusula y esta cláusula es unitaria. Por necesidad $q \equiv false$ y reescribimos una vez más la fórmula:

$$\phi = \neg false$$

Finalmente, sabemos que $\neg false \equiv true$, por lo que $\phi = true$ y, por ende, ϕ es satisfacible. La asignación de variables es consecuencia directa de las cláusulas unitarias, $p \equiv true$ y $q \equiv false$.

Obtuvimos una asignación de variables que hace a la fórmula satisfacible, sin tener que recurrir a *backtracking*. En el caso general, no será así, pero si reducirá en gran medida el tamaño del problema.

1.1.1 ¿Cuándo hay que propagar las cláusulas unitarias?

Al comenzar la ejecución, todas las cláusulas unitarias deben resolverse antes de comenzar con el *bactracking*. Esto asegura que no realizaremos una búsqueda en vano por valores de variables que no satisfacerán la fórmula.

Una vez comenzada la búsqueda, cada asignación a una variable puede tener como consecuencia la aparición de nuevas cláusulas unitarias. Éstas también deben resolverse antes de continuar con la búsqueda.

Por ejemplo, consideremos la siguiente fórmula:

$$\phi = (\neg p \vee q) \wedge (q \vee \neg r \vee s)$$

Aún no hay cláusulas unitarias. Al comenzar el *backtracking* podríamos decidir intentar hacer $p \equiv true$ y ver si llegamos a una fórmula satisfacible

1.1.2 Estructuras de apoyo para propagación unitaria

El proceso de propagación unitaria debe ser invocado al inicio del proceso y después de cada asignación de variables. Siendo así, debemos tener estructuras de datos que permitan realizar este proceso de forma eficiente.

Una posible implementación es almacenar la fórmula, separando las cláusulas por la cantidad de literales que tienen. Las cláusulas unitarias en una lista, las cláusulas de dos literales en otra lista, las cláusulas de tres literales en otra lista y así en adelante.

Además, cada cláusula tendrá:

- Un contador que indica cuántos literales quedan sin asignar
- Una lista de apuntadores a las variables involucradas en la cláusula. Para cada variable, además tendrá un *flag* que diga si corresponde a un literal negativo (p) o negativo ($\neg p$)

Cada variable, a su vez, tendrá

- Una lista de apuntadores a las cláusulas de las cuales forma parte. Para cada cláusula, además tendrá un *flag* que diga si la ocurrencia de esa variable corresponde a un literal negativo (p) o negativo ($\neg p$)
- Un valor de verdad, que puede ser *true*, *false* o *unassigned*.

Tomemos el mismo ejemplo:

$$\phi = p \wedge (\neg q \vee \neg p) \wedge (q \vee p)$$

Si c_0 es la primera cláusula, c_1 es la segunda cláusula y c_2 es la tercera cláusula, la información de variables es:

- $clausulas(p) = \{(pos, c_0), (neg, c_1), (pos, c_2)\}$
  $verdad(p) = unassigned$
- $clausulas(q) = \{(neg, c_1), (pos, c_2)\}$
  $verdad(q) = unassigned$

Así mismo, la información de cláusulas es:

- $variables(c_0) = \{(pos, p)\}$
  $tam(c_0) = 1$
- $clausulas(c_1) = \{(neg, q), (neg, p)\}$
  $tam(c_1) = 2$
- $clausulas(c_2) = \{(pos, q), (pos, p)\}$
  $tam(c_2) = 2$

La fórmula estará representada por $\phi = \{(1, \{c_0\}), (2, \{c_1, c_2\})\}$. Notemos que las cláusulas están separadas por su tamaño (cantidad de literales que contienen).

1.1.3 Actualización de las estructuras

Cuando se realiza una asignación de verdad, hay que actualizar todas las estructuras de datos afectadas. Esto puede resultar de una propagación unitaria o de una decisión temporal en el *backtracking*.

Al hacer una asignación $p \leftarrow true$, debe ocurrir lo siguiente

- Se cambia el valor de $verdad(p)$ para que sea $true$.
- Se recorren las cláusulas c que dependen de p
  - Si la dependencia con c es positiva, se elimina c de ϕ, pues ya está satisfecha.
  - Si la dependencia con c es negativa, se disminuye en uno $tam(c)$. Nótese que no eliminamos la variable de la cláusula, pues es probable que necesitemos esa información al volver en el *backtracking*.

    Al decrementar el tamaño de la cláusula (que tenía tamaño T) debe pasar al conjunto de cláusulas que tiene tamaño $T - 1$. Si la cláusula resultante es unitaria, se inicia la propagación unitaria de la forma es que ya fue presentado.

Al hacer una asignación $p \leftarrow false$ el tratamiento es análogo.

El programa debe ser capaz de deshacer cualquier acción (inclusive la propagación unitaria) al volver en el *backtracking*. Es por esto que debe mantenerse una **pila de eventos** que mantenga qué acciones se han hecho entre cada decisión tomada por el proceso. Al volver, estas acciones deben poder deshacerse para permitir que se tome otra elección.

1.2 Otras mejoras

Existen muchas mejoras que se pueden aplicar para resolvedores de SAT.

Aprendizaje de implicantes Dada una fórmula ϕ, agrega nuevas cláusulas que son implicadas por las cláusulas existentes en ϕ, que pueden llevar a la búsqueda a dar con una solución de forma más rápida.

Backtracking no–cronológico Muchas decisiones que son tomadas en el *backtracking* son independientes entre sí. Por ejemplo, tomar la decisión $p \leftarrow true$ pudiera ser independiente de $q \leftarrow true$. Si una decisión parece ser prometedora, se pudiera obviar el paso que deshace la decisión, manteniendo las consecuencias y deshaciendo decisiones que fueron tomadas en momentos previos.

Heurística de bifurcación Durante el *backtracking*, debemos escoger cuál variable ha de ser asignada y con qué valor, de forma tal de explorar asignaciones más prometedoras primero.

Otras mejoras son presentadas en detalle en artículos científicos alrededor de resolvedores de SAT. Junto al enunciado tienen seis (6) artículos a su disposición con información útil para mejorar sus resolvedores. Son libres de buscar más allá de estos artículos y agregar todas las mejoras que consideren necesarias.

Su implementación debe tener, como mínimo, la técnica de propagación unitaria para mejorar sus resolvedores. Cualquier otra optimización que haga su código más eficiente, debe ser explicada en detalle en su informe y contará para obtener una mayor puntuación en la evaluación de su entrega.

2 Entrega

Los proyectos deben continuar en el mismo repositorio privado en Github del primero proyecto. Una vez más, los colaboradores serán los miembros del equipo y los profesores del curso.

- Usuario de Github de Ricardo Monascal: rmonascal
- Usuario de Github de Carlos Infante: carlosinfante96

La entrega será el día Viernes, 17 de Julio (semana 6), hasta las 11:59pm VET. A la hora de la entrega, se hará `pull` del repositorio y se corregirá lo que exista ahí hasta ese punto.

2.1 Programas

Su entrega debe incluir los siguientes programas:

- Traductor de instancias de Sudoku a instancias de SAT
- **Resolvedor propio de SAT (mejorado)**
- Traductor de soluciones de SAT a soluciones de Sudoku

Además, debe incluir al menos dos *scripts* que permita orquestrar los programas implementados.

- Tomar una entrada de Sudoku, traducirla a una entrada de SAT, **aplicarle el resolvedor propio de SAT**, traducir la solución a una solución de Sudoku y reportar.
- Tomar una entrada de Sudoku, traducirla a una entrada de SAT, **aplicarle el resolvedor ZCHAFF de SAT**, traducir la solución a una solución de Sudoku y reportar.

Es importante que incluya en sus *scripts* alguna forma de detener el cómputo y reportar que no se pudo hallar una solución, si pasan más de T unidades de tiempo. El valor de T queda de su parte, pero, al realizar experimentos, debe ser igual para ambos scripts (de tal forma que estén en igualdad de condiciones).

Nótese que lo único que cambia para este proyecto es su implementación del resolvedor de SAT.

2.2 Informe

Su proyecto debe incluir un manual de uso y una explicación detallada sobre la implementación: las estructuras de datos y algoritmos utilizados, así como un análisis de recursos para las componentes más importantes (en notación asintótica).

Se recomienda que esta sección del informe se ubique en un archivo `README.md` (escrito en el lenguaje *Markdown*), de tal forma que se muestre correctamente en la página inicial de su repositorio en Github.

2.2.1 Tiempos de ejecución

Debe ejecutar sus soluciones sobre todas las instancias incluidas en el archivo `InstanciasSudoku.txt` y reportar lo siguiente:

- Tiempo de ejecución del resolvedor de sudoku (en milisegundos), usando el resolvedor propio de SAT.
- Tiempo de ejecución del resolvedor de sudoku (en milisegundos), usando el resolvedor ZCHAFF de SAT.

Para esto es necesario crear gráficos comparativos, de tal forma que se facilite el análisis de los resultados presentados.

2.2.2 Instancias resueltas

Debe elaborar un informe tal que, para las instancias que hayan sido resueltas por cualquiera de los dos métodos se incluya una representación legible (en forma de matriz) del tablero de Sudoku que se pasó por entrada y la solución reportada por el algoritmo.

El análisis de tiempo de ejecución y el reporte de instancias resueltas puede ir en un mismo informe. Aunque no es un requerimento, se recomienda crear un programa que genere automáticamente estos informes a partir de la información de ejecución.

2.2.3 Manual de uso

Su documentación debe incluir un manual de uso, incluyendo las dependencias necearias (lenguajes, herramientas y librerías con sus respectivas versiones).

El Manual debe poder llevar a un usuario de su programa a obtener resultados equivalentes a aquellos obtenidos en su informe de tiempos de ejecución.