Universidad Nacional Autónoma de México

Facultad de Ciencias

Lógica Computacional

PROBLEMA SAT Y MINISAT EN HASKELL PARA RESOLVER EL PROBLEMA DE COLORACIÓN DE GRÁFICAS

Equipo:
Cabrera Sánchez Ana Dhamar, 319299904
Rosas Franco Diego Angel, 318165330
Sosa Romo Juan Mario, 320051926

Fecha de entrega:
9 de Junio de 2024

Profesor:
M.C.I.C. Manuel Soto Romero

Ayudantes:
José Alejandro Pérez Márquez
Karla Denia Salas Jiménez
Erik Rangel Limón
Dicter Tadeo García Rosas

Resumen

El problema de satisfacibilidad booleana (problema SAT), demostrado por Stephen Cook y Leonid Levin de manera independiente y catalogado como NP-completo, se centra en determinar si existe alguna asignación de verdad para las variables lógicas que satisfaga una fórmula dada. Los solucionadores SAT, se enfocan en resolver el problema de satisfacibilidad utilizando algoritmos heurísticos y técnicas como el backtracking para encontrar de la manera más eficiente que pueden a éstas asignaciones.

En este trabajo hablamos sobre el problema SAT y los algoritmos más comunes que utilizan los solucionadores SAT (DPLL, CDCL y Look-ahead) junto a métodos incompletos que los mejoran (SLS, SP, NeuroSAT). Se mencionan también algunas variaciones de Max-SAT (problemas más complejos basados en el problema SAT) y hablamos sobre 2 aplicaciones prácticas que tiene el problema de satisfacibilidad booleana en el mundo real: la optimización de tratamientos contra el cáncer y el criptoanálisis, demostrando así la utilidad que este problema tiene en áreas como la biomedicina y la seguridad informática.

En la sección final, hablamos sobre el problema de *coloración de gráficas* y presentamos una implementación en Haskell para la k-coloración de gráficas como una instancia del problema SAT usando 2 bibliotecas: *Graficas* (creada por nosotros) y *MiniSat* (disponible en Haskell) se detalla el código en la sección correspondiente.

Preliminares

Stephen Cook y Leonid Levin demostraron de manera independiente el problema de satisfacibilidad booleana (problema SAT) en los años 1971 y 1973 respectivamente. Éste problema, catalogado como NP-completo, determina si existe o no, alguna asignación de verdad de las variables lógicas involucradas en una fórmula que logre satisfacer a ésta, las pruebas en las que trabajaron muestran que cada problema de decisión en la clase de complejidad NP se puede reducir al problema SAT para fórmulas normales conjuntivas.

[AcademiaLab(s.f)]

Los solucionadores SAT se encargan particularmente de resolver el problema de satisfacibilidad booleana de la manera más eficiente que pueden, utilizan algoritmos que exploran las posibles asignaciones de verdad sobre las variables lógicas de una fórmula dada e intentan encontrar los valores que vuelven satisfacible a dicha fórmula. En caso de encontrar una solución, los solucionadores SAT muestran los valores de verdad que se han encontrado para satisfacer a la fórmula y en caso contrario, notifican que ninguna asignación a las variables lógicas puede satisfacer la fórmula proporcionada.

Por naturaleza, la complejidad combinatoria del problema logra que en el peor de los casos, el tiempo de resolución jamás logre ser polinomial y se convierta en exponencial, sin embargo, con el paso de los años y el estudio de estos problemas, se ha optado por usar métodos de búsqueda y técnicas de backtracking para encontrar las soluciones en un menor tiempo al exponencial, los algoritmos usados para esto, nombrados algoritmos heurísticos, no garantizan la eficacia en todos los casos que han de solucionarse, pues la resolución tiende a utilizar reglas prácticas que funcionan en la mayoría de las soluciones pero no exactamente para todas, su enfoque se basa en tomar decisiones rápidas por lo que es común que al final de su ejecución se encuentren soluciones sub-óptimas o que el algoritmo se detenga en una solución que pareciera ser aceptable sin revisar el resto de las soluciones posibles para el problema.

[Johnson et al.(2022)Johnson, Clarke, and Plaisted]

Algoritmos de Búsqueda para el problema SAT

1. **Davis-Putnam-Logemann-Lovelan (DPLL)**
   Es uno de los primeros algoritmos que se desarrollaron para la búsqueda de satisfacibilidad booleana, a la fecha, resulta de los mejores y más completos algoritmos para resolver el problema SAT, llegando a ser la base de otros algoritmos modernos que buscan una solución a la satisfacibilidad de una fórmula dada.

   De manera general, el algoritmo realiza la búsqueda de asignaciones que reducen al problema original; dependiendo de la implementación, puede o no tener varias

optimizaciones pero las optimizaciones esenciales que utiliza son las siguientes:

- **Backtracking Search:** Selecciona una variable y le asigna un valor, después intenta resolver recursivamente el problema para la formula resultante si no se puede continuar con la asignación, regresa y cambia nuevamente el valor de la variable.
- **Unit Propagation:** Si una clausula contiene una sola literal, entonces solo es satisfacible mediante el valor necesario para hacer verdadera la clausula (literal), esto nos reduce muchísimo el espacio de búsqueda.
- **Pure Literal Elimination:** Si una literal aparece en todas las clausulas con el mismo signo entonces podemos asignarle el valor necesario para hacer verdadera a todas las clausulas (si la literal 'A' aparece en todas las cláusulas y su complemento jamás aparece, entonces podemos asignar la literal 'A' de forma que todas las cláusulas sean verdaderas).
  Con el avance del algoritmo, se ha notado que é-s esto no reduce en gran escala a la formula y verificar si una literal es pura resulta muy costoso, por lo que se está dejando de utilizar esta optimización.

[Johnson et al.(2022)Johnson, Clarke, and Plaisted]
[Sancho(2022)]

2. **Conflict-Driven Clause Learning (CDCL)**
   Se le considera una ligera mejora respecto a DPLL (le agrega mas cosas) y es el algoritmo más usado en la actualidad, la idea es la misma pero se le añade un mecanismo de aprendizaje de cláusulas que nos permiten reducir el espacio de búsqueda, adicionalmente, se le puede incluir retrocesos cronológicos y heurísticas de selección de variables.

   Destacamos las siguientes partes del algoritmo:

   - **Conflict Analysis:** Si una asignación nos lleva a una contradicción, entonces se activa el proceso de aprendizaje de clausulas.
   - **Learned Clauses:** Generamos nuevas cláusulas basadas en los conflictos para reducir el espacio de búsqueda.
   - **Backjumping (no cronológico):** Si una cláusula aprendida nos lleva a un conflicto, entonces retrocedemos hasta la variable que causo el conflicto (no necesariamente es la más reciente).
   - **Restarts:** Cada cierto tiempo se reinicia el algoritmo para evitar que se quede estancado en un mínimo local.

   [Biere et al.(2016)Biere, Heule, van Maaren, and Walsh]

3. **Look-ahead**
   Esta ultima técnica también es una mejora con respecto a DPLL, pero toma un camino algo diferente que CDCL, se enfoca en evitar conflictos, es decir intenta tener una buena selección de variables y simplificar la formula para dirigir la búsqueda.
   [Heule et al.(2016)Heule, Kullmann, and Marek]

   Por ello esta estrategia tardara mucho mas en seleccionar una variable, pero se espera que el tiempo de búsqueda sea menor, algunas de las técnicas que se usan en este algoritmo de búsqueda son:

   - **Unit Propagation:** Se sigue usando, pero llega a ser mas profunda.
   - **Impact Analysis:** Se evalúa cuánto impacto tiene una variable en la fórmula, se puede hacer por ejemplo haciendo un caso con true y otro con false.

   [Heule and van Maaren(2009)]

Métodos incompletos para soluciones al problema SAT

Todos estos métodos como su nombre indica, no garantizan encontrar una solución, incluso si existe una, pero generalmente son más rápidos y pueden ser útiles para encontrar soluciones aproximadas o para problemas de gran escala donde los métodos completos son inviables. Algunos métodos incompletos para SAT incluyen:

1. **Stochastic Local Search (SLS)**
   Este método, como en el titulo indica explora de manera más aleatoria, empleando varias heurísticas para guiar la búsqueda. Estos métodos son muy eficientes en la practica y han sido utilizados en la resolución de problemas de gran escala.

   Cuenta con los siguientes principios:

   - **Estado inicial:** Se genera una asignación inicial (potencialmente aleatoria).
   - **Búsqueda local:** Se modifica ligeramente la asignación local para mejorarla.
   - **Movimientos estocásticos:** Se hacen movimientos aleatorios para no quedar atrapados en óptimos locales.
   - **Evaluación de solución:** Utiliza una función de evaluación para determinar la calidad de la solución (suele contar el numero de clausulas satisfechas).

   Algunos ejemplos de SLS populares son: Greedy SAT (GSAT), WalkSAT, Novelty+ y ProbSAT.
   [Hoos(1999)]

2. **Survey Propagation (SP)**
   Probablemente la mas complicada de entender, es un algoritmo que se usa cuando las variables son altamente interdependientes y se basa en la propagación de mensajes entre las variables.

   Tiene los siguientes componentes:

   - **Mensajes:** Se envían mensajes entre las variables y clausulas, estos mensajes tienen la información de la probabilidad de que una variable sea verdadera o falsa.
   - **Reglas de actualización:** Se actualizan los mensajes de acuerdo a reglas predefinidas, como lo son los mensajes recibidos de clausulas vecinas.
   - **Decisiones basadas en surveys:** Se toman decisiones basadas en los mensajes recibidos, de manera que se enfoca en satisfacer la mayor parte de la formula

   [Kroc et al.(2007)Kroc, Sabharwal, and Selman]

3. **NeuroSAT** Otro enfoque realmente complicado, esta vez basado en redes neuronales; se basa en la idea de que las redes neuronales pueden aprender a resolver problemas de satisfacción de restricciones.

   Se forma de los siguientes componentes:

   - **Gráfica:** Se crea una gráfica que representa la formula CNF (se utiliza un gráfico bipartito y las aristas representan relaciones entre variables y clausulas).
   - **Inicialización:** Se hace una representación vectorial de la gráfica y se inicializan los pesos de las aristas.
   - **Aprendizaje:** Se entrena la red neuronal para que aprenda a resolver problemas de satisfacción de restricciones.
   - **Inferencia:** Se hace inferencia en la red neuronal para obtener una asignación de variables que satisfaga la formula.
   - **Convergencia:** Se espera que la red neuronal converja a un método de satisfacción de restricciones.
   - **Resultados:** Se produce una asignación de variables que satisface la formula.

   Este es el método mas reciente y prometedor, pero también el más complicado para su implementación y entendimiento, además, no garantiza que se encuentre una solución y entrenarlo es muy costoso.
   [Cornford and Nabney(1998)]

Max-SAT

1. **Weighted MaxSAT**
   Este es un problema que generaliza al problema SAT, aquí no solo buscamos una asignación de variables que satisfaga la formula buscamos también que la asignación sea óptima en un sentido especificado; se mantiene como un problema NP-Duro y por tanto se utilizan algoritmos diversos para encontrar soluciones de manera eficiente.

2. **Partial MaxSAT**
   Este solamente es una variante del problema SAT, en donde se busca una asignación de variables que satisfaga a todas las clausulas de un tipo deseadas (difíciles) y se maximice la cantidad de clausulas satisfechas del resto.

3. **Weighted Partial MaxSAT**
   Como su nombre lo indica solo es combinar los dos problemas anteriores, en los cuales se busca una asignación de variables que satisfaga el mayor número de cláusulas posibles que además, sea óptima en un sentido dado.

La información de esta sección fue obtenida de:
[Argelich et al.(2015)Argelich, Li, Manya, and Planes].

Aplicaciones

Aplicación 1: Max-SAT con ATPG para optimizar el diseño de terapias contra el cáncer

Para este apartado, consideraremos la aplicación del problema SAT consultada en: [Lin and Khatri(2012)]. La terminología es compleja y el tema abordado es un poco más fuerte de lo que vimos, así que pedimos disculpas por cualquier malinterpretación del articulo.

En **resumen** el artículo explora como aplicar (ATPG) que es una tecnología usada en la automatización de diseño de circuitos electrónicos basada en un "weighted partial Max-SAT"(maximizando el número de clausulas satisfechas sin necesidad de satisfacer todas, con un peso asociado a cada cláusula) para buscar automáticamente los patrones de prueba que nos permiten verificar el funcionamiento lógico de un circuito; para este caso en particular, se busca aplicar dicha tecnología para buscar de manera óptima y automática la combinación más efectiva de tratamientos que puedan combatir el cáncer.

Es interesante notar que aunque no pareciera que existe una relación entre los dos problemas, uno sirve al otro para encontrar la combinación óptima de tratamientos contra

el cáncer, pues esto se puede ver como un problema de satisfacibilidad booleana, donde hay que codificar las restricciones y objetivos, además, hay que maximizar la eficacia de los tratamientos mientras se minimizan los efectos secundarios de los mismos.

Ahora vamos a ver algunos **detalles más precisos** de como que se hizo en este artículo, pero primero, leamos la información previa que nos proporciona el artículo:

En el artículo se menciona que el cáncer y varias enfermedades genéticas pueden ser causadas por una falla en la señalización de células y sus genes; además, sabemos que por la naturaleza de los genes, éstos parecen comportarse similar a las variables lógicas (estando activos o inactivos) mientras que las cadenas de comunicación se pueden representar como funciones lógicas (en el estudio las comparan un poco más con los circuitos). En este caso, se quiere encontrar el momento en el que estas comunicaciones no son efectivas, para ello, se utilizan métodos como los que se emplean en Testings de circuitos digitales para determinar los inputs problemáticos, posteriormente todo se traduce al problema SAT.

Al **final** el artículo nos dice que el algoritmo que utilizaron les dio buenos resultados, al menos a nivel teórico, y lo mejor de todo es que se puede modificar pues es bastante flexible y regresa una solución exacta en menos de 1 segundo.

Aplicación 2: Solucionadores SAT para el criptoanálisis.

La información sobre esta aplicación se puede encontrar en:
[Lafitte et al.(2014)Lafitte, Nakahara Jr, and Van Heule]; este documento habla también con terminología avanzada, especialmente de criptografía, pero vamos a intentar explicar lo mejor posible.

En **resumen** el artículo utiliza un analizador basado en SAT, específicamente uno llamado CryptoSAT para analizar varias funciones de cifrado y hash con la finalidad de encontrar debilidades en los mismos, llega a varios resultados interesantes en cuanto a llaves débiles (e inexistencia de estas) y nos da un método que nos permite probarlas usando SAT.

De manera un poco más **detallada**, los autores utilizan un método de abstracción para automatizar la generación de la formula en CNF que se introduce en el solucionador SAT y así poder probar las debilidades de cifrados como son WIDEA y MESH-64 además de responder una pregunta sobre pre-imágenes en la función de hash MD4.

En **conclusión** éste articulo nos da una idea de como podemos probar la seguridad de algoritmos de cifrado y hash, dándonos específicamente las llaves que encuentra y caracteriza como débiles, o la nula existencia de éstas; además nos proporciona una manera eficiente y automatizada para generar instancias del problema SAT que engloban un gran rango de algoritmos de llave simétricos y funciones de hash.

Implementación

Problema a resolver: Coloración de gráficas

El problema de coloración de gráficas es un problema considerado como NP-Completo que intenta encontrar la cantidad mínima de colores necesarios para colorear los vértices de una gráfica, de tal manera que los vecinos del vértice no compartan su color.

Definida formalmente, «una **gráfica** G es una terna ordenada (V_G, E_G, ψ_G). Llamaremos **vértices** a los elementos de V_G. El conjunto E_G es tal que $V_G \cap E_G = \emptyset$, y sus elementos son llamados **aristas**. La función ψ_G es la **función de incidencia** de G, que asocia a cada arista una pareja no ordenada de vértices (no necesariamente distintos) de G.»

En otras palabras

$$\psi_G : E_G \to \binom{V_G}{1} \cup \binom{V_G}{2}$$

Cuando $e \in E_G$ y $u, v \in V_G$ son tales que $\psi_G(e) = \{u, v\}$, diremos que u y v son los **extremos** de la arista e; también decimos que u es **vecino** de v o que los vértices u y v son **adyacentes**.

Entendiendo esto, definimos entonces la **k-coloración** de gráficas como sigue: «Dada una gráfica G y un entero positivo k, una **k-coloración por vértices**, o simplemente **k-coloración**, de G es una función $c : V \to S$, donde S es un conjunto de cardinalidad k, cuyos elementos serán llamados *colores*. Así, una *k-coloración* de G es una asignación de colores a los vértices de G, usando a lo más k colores. Aunque el conjunto S no tiene restricciones, usualmente se utilizará $\{1, \ldots, k\}$ como conjunto de colores.»

Si la gráfica admite una **k-coloración** decimos que es **k-coloreable** e informalmente, añadimos entre este bloque de definiciones, que la gráfica es **aplanable** si existe una manera de realizar un diagrama de la gráfica en un plano de tal forma que no haya cruces entre sus aristas.

Todas las definiciones y la descripción general del problema fue consultada en [Cruz(2022)]

Teorema de los 4 colores

En el año 1852 el matemático Francis Guthrie notó que todos los condados de Inglaterra podían ser coloreados utilizando únicamente 4 colores, de tal manera que si existían fronteras en común, los condados recibieran colores distintos. Francis preguntó a su hermano Frederick, si creía posible que ocurriera para cualquier mapa, quien a su vez preguntó

a su entonces profesor, Augustus De Morgan, sin saber la respuesta, De Morgan escribió a Sir Rowan Hamilton notificándole que uno de sus estudiantes afirmaba que al dividir una figura, sin importar en cuántas partes fuese, se necesitarían, a lo más, 4 colores para colorear cada parte de la figura sin que el color se repitiera en las regiones vecinas. Esperó la respuesta de Hamilton creyendo que mostraría interés hacia el problema, pero Hamilton no pareció estar interesado; fue entonces cuando De Morgan intentó que otros matemáticos se interesaran en este problema, naciendo así la *Conjetura de los 4 colores*.

La conjetura no sería probada correctamente hasta 1976, cuando Kenneth Appel y Wolfgang Haken publicaron una prueba completa del ahora *Teorema de los 4 colores* que especifica lo siguiente: *Toda gráfica aplanable, es 4-coloreable*.

La prueba fue asistida por computadora y utilizó gráficas donde los vértices representaban las partes fragmentadas de las figuras y las función de incidencia que creaba a las aristas, estaba dada por la propiedad *ser una región vecina* de la parte fragmentada en cuestión.

Al ser una demostración que necesitaba aproximadamente de 1936 configuraciones para reducir el problema, la prueba causó controversia. Aún cuando cada una de las configuraciones se comprobó individualmente con ayuda de solucionadores SAT y otras herramientas computacionales para verificar las propiedades necesarias, seguía sin ser una prueba en papel, por lo que algunos matemáticos se negaron a aceptarla. En 1966 se propuso una prueba más legible con solo 633 configuraciones, pero que en gran medida sigue requiriendo el uso de un ordenador para su comprobación, sin embargo, más científicos han aprobado su validez.

Información sobre el Teorema de los 4 colores consultada en [Seijas(2016)], [Cruz(2022)]

Codificación del problema

Para codificar el problema en lógica proposicional podemos usar variables booleanas y cláusulas que representen las restricciones del problema.

Supongamos que tenemos una gráfica $G = (V, E)$ con v vértices y e aristas, es decir, $|V| = v$ y $|E| = e$, y queremos una k-coloración sobre la gráfica G.

Definimos las variables booleanas $x_{i,j}$ donde $x_{i,j}$ es verdadero si el vértice i tiene el color j. Donde $1 \leq i \leq v$ y $1 \leq j \leq k$.

Asi, tendremos las siguientes restricciones:

1. Cada vértice debe tener al menos un color:

$$\bigvee_{j=1}^{k} x_{i,j} \quad \text{para cada vértice } i$$

Indica que para cada vértice i, al menos una de las variables $x_{i,j}$ debe ser verdadera.

2. Cada vértice debe tener a lo sumo un color:

$\neg x_{i,j} \vee \neg x_{i,j'}$ para cada vértice i y cada par de colores distintos j, j' se debe cumplir que $j \neq j'$

Garantiza que un vértice no tenga dos colores distintos.

3. Vértices adyacentes no pueden tener el mismo color:

$$\neg x_{i,j} \vee \neg x_{i',j} \quad \text{para cada arista } (i, i') \in E \text{ y cada color } j$$

Asegura que si hay una arista entre los vértices i e i', estos no pueden compartir el mismo color.

4. Todos los colores solicitados deben usarse en la coloración:

$$\bigvee_{i=1}^{v} x_{i,j} \quad \text{para cada color } j$$

Confirma que la coloración que se encuentre esté usando el número de colores adecuado, de manera que no se obtenga siempre la coloración mínima de la gráfica.

Biblioteca *Graficas*

Se realizó una implementación en Haskell de una biblioteca para trabajar con gráficas no dirigidas con nodos con valores numéricos enteros. A continuación el código realizado.

```haskell
module Graficas (
    Grafica(..),
    Vertice,
    Arista,
    crearGrafica,
    agregarVertice,
    agregarArista,
    vertices,
    aristas,
    vecinos
) where

import Data.List (nub)

-- Vertice | Representado como un valor entero
type Vertice = Int
```

```haskell
-- Arista | Es una tupla que asocia dos vértices
type Arista = (Vertice, Vertice)

-- Grafica | Definida como un conjunto de vertices y aristas.
-- En nuestro caso estos conjuntos los representamos como listas, y posteriormente nos aseguraremos que no haya elementos repetidos en estas.
data Grafica = Grafica [Vertice] [Arista] deriving Show

-- crearGrafica | Crear una gráfica vacía
-- Inicializa los vertices y aristas como listas vacias
crearGrafica :: Grafica
crearGrafica = Grafica [] []

-- agregarVertice| Agrega un vértice a una gráfica
-- Nos aseguramos de quitar repeticiones en los vertices
agregarVertice :: Vertice -> Grafica -> Grafica
agregarVertice v (Grafica vs es) = Grafica (nub (v:vs)) es

-- agregarArista | Agrega una arista a una gráfica
-- Nos aseguramos de quitar repeticiones en los vertices y aristas
agregarArista :: Arista -> Grafica -> Grafica
agregarArista e@(v1, v2) (Grafica vs es) = Grafica (nub (v1:v2:vs)) (nub (e:es))

-- vertices | Obtener la lista de vértices de una gráfica
vertices :: Grafica -> [Vertice]
vertices (Grafica vs _) = vs

-- aristas | Obtener la lista de aristas de una gráfica
aristas :: Grafica -> [Arista]
aristas (Grafica _ es) = es

-- vecinos | Obtener los vecinos de un vértice
vecinos :: Vertice -> Grafica -> [Vertice]
vecinos v (Grafica _ es) = [u | (u, w) <- es, w == v] ++ [w | (u, w) <- es, u == v]
```

Resolviendo la k-coloración con *MiniSat* y *Graficas*

Haciendo uso de nuestra biblioteca definida y la biblioteca de *MiniSat* definimos una función *kColoracion* que construya la fórmula proposicional para la k-coloración y la pase al solucionador SAT para obtener así el conjunto de coloraciones de la gráfica.

El código realizado fue el siguiente:

```haskell
import Graficas
import SAT.MiniSat (solve, solve_all, Formula(..))
import Data.Map (Map, toList)

```

```haskell
import qualified Data.Map as Map

type Color = Int
type Coloracion = [(Vertice, Color)]

-- Restricción 1: Cada vértice debe tener al menos un color
restriccion1 :: [Vertice] -> Color -> [Formula (Vertice, Color)]
restriccion1 vs k = [Some [Var (i, j) | j <- [1..k]] | i <- vs]

-- Restricción 2: Cada vértice debe tener a lo sumo un color
restriccion2 :: [Vertice] -> Color -> [Formula (Vertice, Color)]
restriccion2 vs k = [(Not (Var (i, j))) :||: (Not (Var (i, j')))  | i <-
   vs, j <- [1..k], j' <- [1..k], j /= j']

-- Restricción 3: Vértices adyacentes no pueden tener el mismo color
restriccion3 :: [Arista] -> Color -> [Formula (Vertice, Color)]
restriccion3 es k = [(Not (Var (i, j))) :||: (Not (Var (i', j)))  | (i, i
   ') <- es, j <- [1..k]]

-- Restricción 4: Se deben utilizar todos los colores.
restriccion4 :: [Vertice] -> Color -> [Formula (Vertice, Color)]
restriccion4 vs k = [Some [Var (i, j) | i <- vs] | j <- [1..k]]

-- Definir la función para obtener solo los elementos cuyo segundo valor
    sea True
filtrarTrue :: Map (Vertice, Color) Bool -> [(Vertice, Color)]
filtrarTrue m = [ (v, c) | ((v, c), True) <- toList m ]

-- Función auxiliar para obtener todas las soluciones
kColoracionAll :: Int -> Grafica -> [Map (Vertice, Int) Bool]
kColoracionAll k g = solve_all formula
  where
    vs = vertices g
    es = aristas g
    -- Construcción de la fórmula con todas las restricciones
    formula = All (restriccion1 vs k ++ restriccion2 vs k ++
   restriccion3 es k ++ restriccion4 vs k)

-- Función principal que construye la fórmula y obtiene todas las k-
   coloraciones
kColoracion :: Int -> Grafica -> [Coloracion]
kColoracion k g = case kColoracionAll k g of
    []      -> []  -- No hay soluciones
    sols    -> map filtrarTrue sols
```

Conclusiones

A lo largo de este proyecto, pudimos observar que la naturalidad combinatoria de crear coloraciones en gráficas, especialmente para mostrar la 4-coloración en gráficas aplanables, puede requerir un tiempo exponencial capaz agotar los recursos computacionales incluso antes de terminar con el proceso de búsqueda si el tamaño de la gráfica que recibe como entrada es lo suficientemente grande, sin embargo, el uso de solucionadores SAT ayuda a reducir el tiempo en el que podamos encontrar una solución, si bien no ofrece garantía de encontrar la solución más óptima o de que el proceso no llegue a su peor caso posible (tienda a exponencial), intenta reducir en gran medida la complejidad del programa.

Al ser el problema de satisfacibilidad un problema catalogado como NP-Completo, sabemos, por el Teorema de Cook, que cualquier otro problema NP puede ser instanciado en él, por lo cual, nuestros argumentos anteriores sobre la manera en que los solucionadores SAT nos ayudan a reducir la complejidad del programa que creamos, sirven no solo para coloración de gráficas, si no también para otras codificaciones que puedan transformarse para resolverse como un problema SAT, por lo cual pensamos en los solucionadores SAT como una de las herramientas más poderosas que pueden enfrentarse a los problemas NP.

Referencias

[AcademiaLab(s.f)] AcademiaLab. Problema booleano de satisfacibilidad. `https://academia-lab.com/enciclopedia/problema-booleano-de-satisfacibilidad/`, s.f. Accedido: 2024-05-31.

[Argelich et al.(2015)Argelich, Li, Manya, and Planes] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. Maxsat evaluation (2016). 2015.

[Biere et al.(2016)Biere, Heule, van Maaren, and Walsh] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. `https://ics.uci.edu/~dechter/courses/ics-275a/winter-2016/readings/SATHandbook-CDCL.pdf`, 2016. Accedido: 2024-06-03.

[Cornford and Nabney(1998)] Dan Cornford and Ian T Nabney. Neurosat: an overview. 1998.

[Cruz(2022)] César Hernández Cruz. Introducción a la teoría de gráficas. Technical report, Universidad Nacional Autónoma de México (UNAM), 2022.

[Heule et al.(2016)Heule, Kullmann, and Marek] Marijn Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT 2016: Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245. Springer, 2016. URL `https://www.cs.utexas.edu/~marijn/publications/p01c05_lah.pdf`. Accedido: 2024-06-03.

[Heule and van Maaren(2009)] Marijn JH Heule and Hans van Maaren. Look-ahead based sat solvers. In *Handbook of satisfiability*, pages 155–184. IOS Press, 2009.

[Hoos(1999)] Holger Hoos. *Stochastic local search-methods, models, applications*. Ios Press, 1999.

[Johnson et al.(2022)Johnson, Clarke, and Plaisted] J. R. Johnson, E. M. Clarke, and D. A. Plaisted. Automated reasoning. *The Stanford Encyclopedia of Philosophy*, 2022. URL `https://plato.stanford.edu/entries/reasoning-automated/`. `https://plato.stanford.edu/archives/spr2022/entries/reasoning-automated/`.

[Kroc et al.(2007)Kroc, Sabharwal, and Selman] Lukas Kroc, Ashish Sabharwal, and Bart Selman. Survey propagation revisited. In *UAI*, volume 7, pages 217–226, 2007.

[Lafitte et al.(2014)Lafitte, Nakahara Jr, and Van Heule] Frédéric Lafitte, Jorge Nakahara Jr, and Dirk Van Heule. Applications of sat solvers in cryptanalysis: finding weak keys and preimages. *Journal on Satisfiability, Boolean Modeling and Computation*, 9 (1):1–25, 2014.

[Lin and Khatri(2012)] Pey-Chang Kent Lin and Sunil P Khatri. Application of max-sat-based atpg to optimal cancer therapy design. *BMC genomics*, 13:1–10, 2012.

[Sancho(2022)] Francisco Sancho. Lógica informática, tema 03. `https://www.cs.us.es/~fsancho/ficheros/LI2022/tema-03.pdf`, 2022. Accedido: 2024-06-03.

[Seijas(2016)] Sergio Pena Seijas. El problema de coloración de grafos. Technical report, Universidad de Santiago de Compostela, 2016.