Lógica Computacional, 2025-2
Unidad 2: Lógica Proposicional

Solucionadores SAT

Manuel Soto Romero

17 de marzo de 2025
Facultad de Ciencias UNAM

En esta nota exploraremos cómo modelar problemas computacionales en términos del problema de satisfacibilidad proposicional SAT, una técnica ampliamente utilizada en lógica computacional, verificación de software y hardwadre, inteligencia artificial y optimización. Como primer paso, analizaremos el problema de decidir si un conjunto de fórmulas es satisfacible. Luego, ilustraremos esta idea con un problema concreto: la selección de un día para un reunión, donde modelaremos restricciones de disponibilidad de las personas participantes y demostraremos cómo convertir el problema en una instancia de SAT.

Más adelante, generalizaremos esta metodología al problema de coloración de gráficas, un problema clásico en teoría de gráficas con aplicaciones en planificación, asignación de recursos y optimización. Veremos cómo expresar las restricciones de coloración en lógica proposicional, cómo traducirlas a una formulación que pueda ser resuelta con un solucionador SAT, y cómo obtener soluciones utilizando herramientas computacionales como MiniSAT en Haskell. Finalmente discutiremos por qué este enfoque es útil y cómo se relaciona con la teoría de la complejidad, en particular con la NP-completitud de SAT.

1. Introducción

Determinar si un conjunto de fórmulas Γ es satisfacible, es decir, si existe un estado de las variables que haga verdaderas todas las fórmulas en Γ, es un problema de gran importancia en lógica computacional y en muchas áreas de las ciencias de la computación. Su importancia radica en que aparece en contextos como la verificación de software y hardware, donde se necesita garantizar que un sistema no tenga errores lógicos; en inteligencia artificial, donde se usa para resolver problemas de planificiación y toma de decisiones; y en optimización, donde muchas restricciones pueden expresarse en términos de lógica proposicional.

Definición 1. (Problema SAT

*Dada una fórmula de la lógica proposicional φ, el **problema de satisfacibilidad proposicional (SAT)** consiste en determinar si existe un estado de las variables de φ de manera que φ sea verdadera.*

A continuación, exploraremos ejemplos concretos de cómo se pueden moderlar problemas reales en términos de SAT.

Ejemplo 1

Un grupo de personas desea reunirse un único día dentro de la semana laboral, de lunes a viernes. La elección del día debe respetar la disponibilidad de cada persona.

Codificación del Problema

Cada persona indica su disponibilidad para ciertos días de la semana. Denotamos los días con p_i, donde $0 \leq i \leq 4$ representa:

- p_0 lunes
- p_1 martes
- p_2 miércoles
- p_3 jueves
- p_4 viernes

Las restricciones se codifican mediante las siguientes cláusulas:

1. **Días disponibles**
$$(p_i \vee p_j \vee \dots)$$
Indica que hay disponibilidad en los días $i, j, k, \dots$

2. **Días no disponibles**
$$(\neg p_i \vee \neg p_j \vee \dots)$$
Indica que no hay disponibilidad en los días $i, j, k, \dots$

3. **Elección de un único día** (sólo uno de los p_i debe ser verdadero)
$$p_i \rightarrow (\neg p_j \wedge \neg p_k \wedge \dots)$$
lo cual se traduce en FNC como:
$$(\neg p_i \vee \neg p_j) \wedge (\neg p_i \vee \neg p_k) \wedge \dots$$
es decir, no pueden ser verdaderos dos días a la vez.

Ejemplo de Instancia del Problema

Adriana, Brenda, Camila y Daniela son mejores amigas y desean reunirse un día entre semana, considerando sus restricciones de disponibilidad:

- Adriana puede reunirse únicamente el lunes o miércoles.
- Brenda no puede reunirse el miércoles.
- Camila no puede reunirse el viernes.
- Daniela puede reunirse únicamente el jueves o viernes.

Las restricciones que representan esta instancia son:

$$r_1 = (p_0 \vee p_2) \wedge \neg p_2 \wedge \neg p_4 \wedge (p_3 \vee p_4)$$

$$r_2 = (\neg p_0 \vee p_1) \wedge (\neg p_0 \vee p_2) \wedge (\neg p_0 \vee p_3) \wedge (\neg p_0 \vee p_4) \\ \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_1 \vee p_4) \\ \wedge (\neg p_2 \vee p_3) \wedge (\neg p_2 \vee p_4) \wedge (\neg p_3 \vee p_4)$$

De esta forma el conjunto Γ se construye mediante la obtención de las clausulas formadas por la conjunción de las restricciones $r_1 \wedge r_2$. A partir de aquí es posibe verificar que aplicando nuestro algoritmo de resolución binaria mediante saturación se puede llegar a la cláusula vacía $\square$, indicando **insatisfacibilidad**, por lo tanto no existe un día de la semana en que se puedan reunir las amigas.

Modificación de las Restricciones

Si modificamos las restricciones de la siguiente manera:

- ⋆ Adriana: lunes o miércoles
- ⋆ Brenda: martes o miércoles
- ⋆ Camila: martes o jueves
- ⋆ Daniela: jueves o viernes

La restricción r_1 cambia por:

$$r_1' = (p_0 \vee p_2) \wedge (p_1 \vee p_2) \wedge (p_1 \vee p_3) \wedge (p_3 \vee p_4)$$

De esta forma el conjunto Γ se construye ahora mediante la obtención de las clausulas formadas por la conjunción de las restricciones $r_1' \wedge r_2$. A partir de aquí es posibe verificar que aplicando nuestro algoritmo de resolución binaria mediante saturación no se puede llegar a la cláusula vacía, indicando **satisfacibilidad**, por lo tanto las amigas pueden reunirse un día de la semana.

Ejercicio 1

1. Verifica que la primera instancia del problema anterior en efecto es insatisfacible.
2. Verifica que la segunda instancia del problema anterior en efecto es satisfacible y responder a las preguntas: ¿en qué día de la semana pueden reunirse? ¿esta solución es única?

SAT como Problema NP-Completo

El algoritmo resolución binaria con saturación que hemos estudiado en esta y otras notas tiene una complejidad exponencial en el peor caso porque cada paso de resolución puede generar nuevas cláusulas, y el número total de cláusulas posibles en una fórmula con n variables puede crecer hasta $\mathcal{O}(2^n)$. Dado que la resolución se aplica repetitivamente hasta que no se puedan generar más resolventes, el conjunto de cláusulas puede expandirse de manera combinatoria, lo que resulta en un aumento exponecial del espacio de búsqueda. Aunque en algunos casos específicos la explosión combinatoria se controla, en general, el número de resolventes crece demasiado rápido para permitir una solución eficiente en el peor de los casos.

Hasta la fecha, no se conoce un algoritmo determinista que resuelva el problema SAT en tiempo polinomial en el peor caso sin recurrir a enfoques aproximados o restricciones sobre la estructura de las fórmulas. Motivo por el cuál es considerado como un problema NP-Completo.

Clasificación de Problemas Computacionales: Una Instrodución Intuitiva

Cuando estudiamos problemas en computación, nos interesa saber qué tan difícil es resolverlos y qué tipo de algoritmos existen para solucionarlos. Para entender esto, los problemas se agrupan en lo que se conoce como clases de complejidad y que nos ayudan a clasificarlos según el tiempo que requieren en función del tamaño de entrada.

La clase P (Problemas Polinomiales)

La clase P contiene todos los problemas que se pueden resolver de manera eficiente. esto significa que existe un algoritmo que puede encontrar la solución en tiempo polinomial, es decir, en un número de pasos que crece a lo más como una potencia del tamaño de la entrada.

Ejemplo 2

Imagina que tienes una lista de números ordenada y necesitas saber si un número específico está en la lista. El algoritmo de búsqueda binaria resuelve este problema en tiempo logarístico, que es incluso más rápido que polinomial. Otro ejemplo es ordenar una lista de números, que se puede hacer en tiempo $\mathcal{O}(n\ log\ n)$, lo cual sigue siendo eficiente.

Si el tamaño de la entrada es n, un algoritmo en P tiene un tiempo de ejecución como:

$$\mathcal{O}(n) \quad \mathcal{O}(n^2) \quad \mathcal{O}(n^3) \quad \mathcal{O}(n^{10}) \quad \dots$$

En general, cualquier problema en P es considerado fácil de resolver en términos computacionales.

La clase NP (Problemas Verificables en Tiempo Polinomial)

Un problema pertenece a la clase NP si, dada una posible solución, podemos verificar rápidamente si es correcta en tiempo polinomial.

Ejemplo 3

Imagina que tienes una caja fuerte con muchas combinaciones posibles, y tu tarea es encontrar la correcta. Probar una clave es fácil y rápido (sólo giras el dial y ves si abre), pero encontrarla desde cero puede ser muy difícil porque hay demasiadas combinaciones.

Si el tamaño de la entrada es n, un problema en NP no necesariamente se resuelve en tiempo polinomial, pero si alguien nos da una solución, podemos verficar en tiempo polinomial si es correcta.

$$\mathcal{O}(n) \quad \mathcal{O}(n^2) \quad \mathcal{O}(n^3) \quad \mathcal{O}(n^{10}) \quad \dots$$

Observación 1

Todos los problemas en P también están en NP, porque si podemos encontrar una solución en tiempo polinomial, también podemos verificarla en tiempo polinomial. Sin embargo, no sabemos si todos los problemas en NP están en P, lo que lleva a la gran pregunta de la computación:

$$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$$

Si alguien demostrara que P = NP, significaría que todos los problemas para los que podemos verificar soluciones rápidamente también pueden resolverse rápidamente.

La clase NP-completo (Los problemas más difíciles de NP)

Un problema es NP-completo si cumple con dos condiciones:

1. Está en NP, es decir, podemos verificar una solución en tiempo polinomial.
2. Es al menos tan difícil como cualquier otro problema en NP, lo que significa que cualquier otro problema en NP puede transformarse en el él en tiempo polinomial (reducción polinomial).

Ejemplo 4

Piensa en los rompecabezas más difíciles que puedas imaginar. Resolverlos desde cero es complicado, pero si alguien te da una posible solución, puedes comprobar rápidamente si es correcta. Ahora, imagina que cualquier otro problema difícil en NP puede convertirse en uno de estos rompecabezas, de modo que si encontramos una forma de resolver uno, entonces podríamos resolver todos los problemas en NP rápidamente.

El primer problema demostrado como NP-completo fue precisamente SAT, en el famoso Teorema de Cook[1] (1971).

Teorema de Cook

Si un problema pertenece a la clase NP, entonces puede reducirse en tiempo polinomial a una instancia del problema SAT.

Desde entonces, muchos otros problemas se han demostrado como NP-completos, como:

- ⋆ 3-SAT (una versión de SAT donde cada cláusula tiene exactamente 3 literales).
- ⋆ Problema del Viajero (encontrar la ruta más corta que pase por todas las ciudades una sola vez y regrese al punto de inicio).
- ⋆ Coloración de Gráficas (determinar si podemos colorear una gráfica con un número fijo de colores sin que vértices adyacentes tengan el mismo color).

Si algún día alguien descubre un algoritmo eficiente para resolver cualquier problema NP-completo, automáticamente resolvería todos los problemas NP en tiempo polinomial.

[1] La demostración de este teorema queda fuera de los alcances de este curso.

2. ¿Qué es un Solucionador SAT

Un solucionador SAT[2] es un programa diseñado para resolver el problema SAT.

En su base, usan la técnica de resolución binaria con saturación, que garantiza la completitud del método. Sin embargo, debido a la explosión combinatoria que esto implica, en la práctica incorporan métodos heurísticos[3] avanzados para acelerar la búsqueda de soluciones. Estas heurísticas permiten reducir significativamente el número de resolventes generados y enfocarse en las partes más prometedoras del espacio de búsqueda.

A pesar de estos avances, desde un punto de vista formal, la existencia de heurísticas no cambia el hecho de que SAT sigue siendo un problema NP-completo. Esto significa que, en el peor caso, el tiempo requerido por cualquier solucionador SAT conocido sigue siendo exponencial.

Breve Historia

1. **1960 - Primer algoritmo SAT**: Davis y Putnam desarrollan el primer enfoque basado en resolución binaria.

2. **1962 - Algoritmo DPLL**: Se introduce la técnica de retroceso[4] y propaganación de unidades.

3. **1971 - SAT es declarado como el primer problema NP-completo**: Según el Teorema de Cook, cualquier problema en NP puede reducirse a SAT.

4. **1992 - Búsqueda Local**: Se introduce un nuevo enfoque basado en heurísticas para mejorar la búsqueda de soluciones.

5. **1996 - Aprendizaje de cláusulas basado en conflictos (CDCL)**: Se mejora el razonamiento dentro del solucionador para evitar exploraciones redundantes.

6. **Desde 2000 - Competencias y avances constantes**: La Competencia Internacional de SAT, organizadas desde 1992, ha impulsado mejorar en solucionadores SAT cada año, permitiendo resolver instancias cada vez más grandes y complejas.

Aplicaciones de los Solucionadores SAT

Los solucionadores SAT han transformado múltiples áreas de la computación, incluyendo:

Verificación de hardware y software

- ⋆ Intel los usa para verificar chips antes de su fabricación.
- ⋆ Microsoft los emplea en verificación de código.
- ⋆ Se aplican en software crítico para autos, aviones y sistemas embebidos.

Inteligencia Artificial y Planificación

[2] *SAT-solver*

[3] Técnica que ayuda a encontrar soluciones de manera más rápida al enfocarse en las partes más prometedoras del problema, en lugar de explorar todas las posibilidades. No garantiza siempre la mejor solución, pero en la práctica mejora el rendimiento y reduce el tiempo de cómputo.

[4] *backtracking*

- ⋆ Son usados en pilotos automáticos y sistemas de calendarización.
- ⋆ En problemas de aprendizaje automático[5], algunos modelos pueden reformularse como problemas de satisfiacibilidad.

Resolución de otros problemas NP-completos

- ⋆ Problemas de coloración de gráficas, *cliques* y tripletas pitagóricas pueden transformarse en instancias de SAT.

Solucionadores SAT en la Era de la IA

Gracias al auge de la Inteligencia Artificial, los solucionadores SAT han segurgido como herramientas clave en la solución de problemas complejos. Modelos modernos de IA requieren resolver grandes cantidades de restricciones lógicas y optimización discreta, tareas en las que los solucionadores SAT son altamente eficientes.

El desarrollo de técnicas híbridas que combinan aprendizaje automático con solucionadores SAT ha permitido abordar problemas antes considerados intratables. Este resurgimiento ha impulsado nuevas líneas de investigación y aplicaciones prácticas en ámbitos como la optimización de redes neutornales, planificación automática y la criptografía.

3. MiniSAT

MiniSAT es un solucionador SAT minimalista y eficiente, desarrollador por Niklas Eén y Niklas Sörensson en 2003. Su diseño compacto y modular lo convierte en una excelente herramienta para quienes desean aprender sobre solucionadores SAT, ya que ofrece un equilibrio entre simplicidad y rendimiento, permitiendo explorar los principios fundamentales de los solucionados SAT modernos sin la complejidad de herramientas más avanzadas. Entre sus principales características se encuentran:

- ⋆ Utiliza la técnica de aprendizaje de cláusulas basado en conflictos (CDCL), que mejora significativamente el rendimiento al evitar exploraciones redundantes del espacio de búsqueda.
- ⋆ Implementa heurísticas avanzadas para seleccionar qué variables asignar primero.
- ⋆ Usa la propagación de unidades para deducir valores automáticamente y retroceso no cronológico para retroceder de manera más eficiente cuando se detectan conflictos.
- ⋆ Es la base para otros solucionadores SAT más avanzados.
- ⋆ A pesar de su simplicidad, ha sido altamente competitivo en la Competencia Internacional de SAT, logrando resolver instancias de gran tamaño con miles de variables y cláusulas.

Está disponible en múltiples lenguajes de programación, lo que facilita su integración. Fue originalmente desarrollado en C++, sin embargo, para los fines de este curso haremos uso de una versión incluida en Haskell conocida como SAT.MiniSat.

Constructores disponibles

Los siguientes constructores están disponibles para representar fórmulas en la biblioteca:

[5] *Machine Learning*

Constructor	Descripción
`Var v`	Define una variable.
`Yes`	Representa la constante verdadera.
`No`	Representa la constante falsa.
`Not (Formula v)`	Negación de una fórmula.
`(Formula v) :&&: (Formula v)`	Conjunción de dos fórmulas.
`(Formula v) :||: (Formula v)`	Disyunción de dos fórmulas.
`(Formula v) :->: (Formula v)`	Implicación entre dos fórmulas.
`(Formula v) :<->: (Formula v)`	Equivalencia entre dos fórmulas.
`All [Formula v]`	Verifica que todas las fórmulas de la lista sean verdaderas.
`Some [Formula v]`	Verifica que al menos una de las fórmulas de la lista sea verdadera.
`None [Formula v]`	Verifica que ninguna de las fórmulas de la lista sea verdadera.
`AtMostOne [Formula v]`	Verifica que a lo más una de las fórmulas de la lista sea verdadera.

Nota 1: La precedencia de los conectivos lógicos está determinada por su orden de aparición.

Nota 2: Las fórmulas están parametrizadas por una variable de tipo que indica cómo se representan las variables (pueden ser enteros, cadenas, caracteres, etc.). En este documento usaremos cadenas para representar las variables.

Obtención de soluciones

Para interactuar con el solucionador SAT, la biblioteca ofrece tres funciones principales:

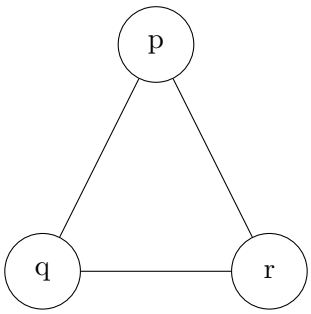
Función	Descripción
`satisfiable :: Formula v -> Bool`	Determina si una fórmula proposicional es satisfacible.
`solve :: Formula v -> Maybe (Map v Bool)`	Obtiene un modelo para la fórmula si existe.
`solve_all :: Formula v -> [Map v Bool]`	Obtiene todos los modelos posibles de la fórmula, si existen.

4. Coloración de Gráficas

El problema de coloración de gráficas consiste en asignar colores a los vértices de una gráfica de manera que dos vértices adyacentes no compartan el mismo color, usando la menor cantidad de colores posible. Este problema tiene aplicaciones en programación de horarios, asignación de recursos y optimización de redes. Una de las ventajas de este problema es que puede reducirse fácilmente a SAT: cada vértice se representa con una variable proposicional por color, y las restricciones de la gráfica (como la adyacencia) se expresan con cláusulas lógicas.

Gráfica Concreta

Primero, veamos un ejemplo concreto en el que modelamos y resolvemos la coloración de la siguiente gráfica simple con tres vértices y tres colores.



La idea es encontrar una forma de asignar colores a los vértices de la gráfica de manera que se cumplan ciertas restricciones.

La gráfica que queremos colorear está definida como:

- ⋆ Vértices: $\{p, q, r\}$
- ⋆ Aristas: $\{(p, q), (q, r), (r, p)\}$
- ⋆ Colores posibles: $\{0, 1, 2\}$ (representados simbólicamente)

Restricciones del Problema

Para garantizar una coloración válida, debemos cumplir dos restricciones fundamentales:

Restricción 1 (r_1): Cada vértice debe tener al menos un color

Cada vértice debe estar asignado a uno o más colores. Esto se traduce en la siguiente condición lógica para cada vértice:

- ⋆ p debe tener al menos uno de los colores 0, 1 o 2.
- ⋆ q debe tener al menos uno de los colores 0, 1 o 2.
- ⋆ r debe tener al menos uno de los colores 0, 1 o 2.

Representación lógica:

$$(p_0 \vee p_1 \vee p_2) \wedge (q_0 \vee q_1 \vee q_2) \wedge (r_0 \vee r_1 \vee r_2)$$

Restricción 2 (r_2): Vértices adyacentes no pueden tener el mismo color

Si dos vértices están conectados por una arista, no pueden compartir el mismo color. Dado que la gráfica es un triángulo, cada vértice está conectado con los otros dos.

Esto se traduce en la siguiente condición para cada color:

- ⋆ Si p tiene el color i, entonces q y r no pueden tener el mismo color i.
- ⋆ Si q tiene el color i, entonces p y r no pueden tener el mismo color i.
- ⋆ Si r tiene el color i, entonces p y q no pueden tener el mismo color i.

Representación lógica:

$$(p_0 \to (\neg q_0 \land \neg r_0)) \land (p_1 \to (\neg q_1 \land \neg r_1)) \land (p_2 \to (\neg q_2 \land \neg r_2))$$
$$(q_0 \to (\neg p_0 \land \neg r_0)) \land (q_1 \to (\neg p_1 \land \neg r_1)) \land (q_2 \to (\neg p_2 \land \neg r_2))$$
$$(r_0 \to (\neg p_0 \land \neg q_0)) \land (r_1 \to (\neg p_1 \land \neg q_1)) \land (r_2 \to (\neg p_2 \land \neg q_2))$$

Esto garantiza que si un vértice tiene un cierto color, los vértices adyacentes no pueden tener el mismo.

Mapeo de las Restricciones a Haskell

El código en Haskell traduce estas restricciones a expresiones que pueden ser procesadas por MiniSAT.

Restricción 1

```
r1 = (Var "p0" :||: Var "p1" :||: Var "p2") :&&:
     (Var "q0" :||: Var "q1" :||: Var "q2") :&&:
     (Var "r0" :||: Var "r1" :||: Var "r2")
```

Restricción 2

```
r2 = (Var "p0" :->: (Not (Var "q0") :&&: Not ( Var "r0"))) :&&:
     (Var "p1" :->: (Not (Var "q1") :&&: Not ( Var "r1"))) :&&:
     (Var "p2" :->: (Not (Var "q2") :&&: Not ( Var "r2"))) :&&:
     (Var "q0" :->: (Not (Var "p0") :&&: Not ( Var "r0"))) :&&:
     (Var "q1" :->: (Not (Var "p1") :&&: Not ( Var "r1"))) :&&:
     (Var "q2" :->: (Not (Var "p2") :&&: Not ( Var "r2"))) :&&:
     (Var "r0" :->: (Not (Var "p0") :&&: Not ( Var "q0"))) :&&:
     (Var "r1" :->: (Not (Var "p1") :&&: Not ( Var "q1"))) :&&:
     (Var "r2" :->: (Not (Var "p2") :&&: Not ( Var "q2")))
```

Fórmula final

La fórmula final combina todas las restricciones en una sola expresión lógica.

```
f = r1 :&&: r2
```

Resolviendo el Problema

MiniSAT nos permite verificar si existe una asignación válida de colores para los vértices que cumpla con todas las restricciones.

Las siguientes funciones pueden usarse para encontrar soluciones:

```
-- Devuelve una asignacion valida de colores (si existe).
solve f

-- Devuelve todas las posibles coloraciones de la grafica.
solve_all f
```

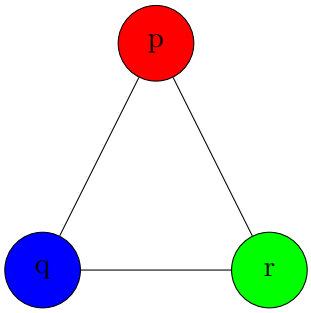
Por ejemplo, si usamos `solve f` una posible salida sería:

```
Just (fromList [("p0",False),("p1",True),("p2",False),
                ("q0",False),("q1",False),("q2",True),
                ("r0",True),("r1",False),("r2",False)])
```

Si asumimos que:

- ⋆ Verde = 0
- ⋆ Rojo = 1
- ⋆ Azul = 2

La solución nos dice que p debe colorearse de rojo, q de azul y r de verde, obteniendo la siguiente solución:



Generalizando el Problema

En la solución anterior con Haskell, presentamos una instancia específica del problema de coloración de gráficas para ejemplificar cómo moderlarlo con un solucionador SAT. Sin embargo, este problema, se puede generalizar para cualquier gráfica y número de colores, lo que nos permite abordar gráficas de cualquier tamaño con restricciones similares. Veamos cómo se estructura y modela esta una posible solución.

Representación del Problema

En este modelo:

- ⋆ Un vértice se representa con un número entero.
- ⋆ Una gráfica es una lista de tuplas, donde cada tupla contiene un vértice y la lista de sus vecinos.

Por ejemplo, la gráfica de nuestro anterior modelo se define como:

```
g1 = [(0,[1,2]),
      (1,[0,2]),
      (2,[0,1])]
```

Traducción del Problema a SAT

Restricción 1: Cada vértice debe tener al menos un color

Cada vértice debe ser asignado a uno o más colores. En lógica proposicional, esto se expresa como:

$$\bigwedge_{v \in V} \bigvee_{c=0}^{k-1} x_{v,c}$$

En Haskell, esta restricción se implementa con la función:

```
r1 :: Grafica -> Int -> Formula String
r1 g k = Data.List.foldr (:&&:) Yes [combina1 v [0..(k-1)] | (v, _) <- g]
```

- ⋆ Se recorre cada vértice v de la gráfica.
- ⋆ Se genera una disyunción con los colores posibles $0, 1, 2, \ldots, k-1$
- ⋆ `combina1` crea la disyunción de colores para cada vértice.

```
combina1 :: Vertice -> [Int] -> Formula String
combina1 v colores = Data.List.foldr (:||:) No [Var (show v ++ show x) |
                     x <- colores]
```

Ejemplo de salida para $k = 3$:

```
(Var "00" :||: Var "01" :||: Var "02") :&&:
(Var "10" :||: Var "11" :||: Var "12") :&&:
(Var "20" :||: Var "21" :||: Var "22")
```

Donde `"00"` representa que el vértice 0 tiene el color 0, `"01"` que tiene el color 1, etc.

Restricción 2: Vértices adyacentes no pueden compartir color

Si dos vértices están conectados, no pueden tener el mismo color:

$$\bigwedge_{(v,w) \in A} \bigwedge_{c=0}^{k-1} (x_{v,c} \to \neg x_{w,c})$$

Esto se implementa en Haskell con:

```
r2 :: Grafica -> Int -> Formula String
r2 g k = Data.List.foldr (:&&:) Yes [combina2 v vs [0..(k-1)] | (v, vs) <- g]
```

`combina2` asegura que un vértices y sus vecinos no ocmpartan el mismo color:

```
combina2 :: Vertice -> [Vertice] -> [Int] -> Formula String
combina2 v vs colores =
    Data.List.foldr (:&&:) Yes [(Var (show v ++ show c)) :->: combina3 vs c
                                | c <- colores]
```

`combina3` niega que los vecinos tengan el mismo color:

```
combina3 :: [Vertice] -> Int -> Formula String
combina3 vs n =
    Data.List.foldr (:&&:) Yes [Not (Var (show v ++ show n))
                                | v <- vs]
```

Ejemplo para $k = 3$:

```
(Var "00" :->: (Not (Var "10") :&&: Not (Var "20"))) :&&:
(Var "01" :->: (Not (Var "11") :&&: Not (Var "21"))) :&&:
(Var "02" :->: (Not (Var "12") :&&: Not (Var "22")))
```

Esto impide que los vecinos compartan el mismo color.

Resolviendo el Problema con SAT

La función principal `kColoracion` resuelve el problema utilizando `solve_all`, que busca todas las soluciones posibles:

```
kColoracion :: Grafica -> Int -> [[(Vertice, Int)]]
kColoracion g k = Data.List.map interpreta (solve_all ((r1 g k) :&&: (r2 g k)))
```

- ⋆ Se generan las restricciones `r1` y `r2`.
- ⋆ Se usa `solve_all` para obtener todas las asignaciones de colores.
- ⋆ Se convierte la salida del solucionador SAT en una lista de vares `(vertice,color)` con la función `interpreta`:

```
interpreta :: M.Map String Bool -> [(Vertice, Int)]
interpreta vars =
    [ (read [v !! 0], read [v !! 1])
    | (v, True) <- M.toList vars
    , length v == 2
    ]
```

Ejemplo de salida para la gráfica `g1`:

```
[[(0,1), (1,2), (2,0)]]
```

Significa que:

- ⋆ Vértice 0 tiene color 1.
- ⋆ Vértice 1 tiene color 2.
- ⋆ Vértice 2 tiene color 0.

Lo que cumple con las restricciones del problema.

5. Conclusión

A lo largo de esta nota, hemos explorado cómo los problemas computacionales pueden modelarse y resolverse utilizando la lógica proposicional y, en particular, los solucionadores SAT. Comenzamos con una introducción a la satisfacibilidad proposicional (SAT) y su importancia en múltiples áreas de la computación, desde la verificación de software y hardware hasta la inteligencia artificial y la optimización. A partir de ejemplos concretos, demostramos cómo estructurar problemas de manera que puedan traducirse a instancias de SAT, asegurando que su solución sea verificable mediante un solucionador automático.

El caso del problema de coloración de gráficas nos permitió ilustrar cómo transformar restricciones de asignación de colores en un conjunto de fórmulas lógicas que pueden evaluarse con un solucionador como MiniSAT. Con ello, vimos que problemas aparentemente distintos, como la asignación de horarios o la planificación de redes, pueden abordarse con una misma metodología basada en lógica proposicional. Este enfoque no solo destaca la potencia de los solucionadores SAT, sino que también refuerza la conexión entre la teoría de la complejidad y la computación aplicada, mostrando que muchos problemas fundamentales en la informática pueden reducirse a instancias de SAT.

Referencias

[1] Sinz, C., & Balyo, T. (2019). *Practical SAT solving* [Diapositivas]. Recuperado de *https://lcs.ios.ac.cn/ caisw/Talks/CaiSATtutorial-Indian6.pdf*

[2] Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*. IOS Press.

[3] MiniSat Developers. (n.d.). *MiniSat: A Minimalistic and Highly Efficient SAT Solver*. Recuperado de *http://minisat.se/Main.html*

[4] Hackage. (n.d.). minisat-solver-0.1 - SAT.MiniSat. *Hackage Documentation*. Recuperado de *https://hackage.haskell.org/package/minisat-solver-0.1/docs/SAT-MiniSat.html*