IIC2213 — Lógica para Ciencia de la Computación

Apuntes Semana 7

1. Resolución vs Fuerza Bruta

Un algoritmo básico para decidir si un conjunto Σ de fórmulas en $L(P)$ es una contradicción (recuerda que en ese caso también decimos que Σ es inconsistente):

1. Generar todas las posibles valuaciones $\tau : P \to \{0, 1\}$,
2. Para cada valuación τ, si $\tau \models \Sigma$, retornar NO.
3. Retornar SI.

Ahora tenemos dos formas distintas de averiguar si Σ es inconsistente. Podemos analizar todas las valuaciones τ para P y verificar que para ninguna valuación se cumple $\tau \models \Sigma$. O podemos usar resolución, dado que Σ es inconsistente si y solo si $\Sigma \vdash \bot$. ¿Cuál método crees que es mejor? ¿Por qué? ¿Cuál es la complejidad de ver si $\Sigma \vdash \bot$?

¿Existirá alguna forma más rápida de resolver este problema? Las próximas semanas van a estar dedicadas a estudiar esto. Por el momento, vamos a ver un poco más de aplicaciones al problema de decidir si Σ es una contradicción o no.

2. SAT

El contrapositivo de decidir inconsistencia recibe el nombre de *satisfacibilidad*, o SAT: consiste en el problema de decidir si un conjunto Σ de fórmulas es satisfacible, es decir, existe por lo menos una valuación τ tal que $\tau \models \Sigma$. De esta forma, Σ es satisfacible si y solo si Σ no es una contradicción.

La popularidad de SAT va más allá de agentes, consecuencia lógica y resolución: tiene que ver con que muchos problemas en computación se pueden reducir a sat. Veamos un ejemplo. Supongamos que tenemos un grafo $G = (V, E)$ y queremos ver si acaso el grafo es 3-coloreable (un grafo es 3-coloreable si podemos asignar uno de tres colores r,a,b a sus nodos, de manera que ninguna arista tiene sus dos nodos del mismo color).

Podemos resolver este problema con SAT! En efecto, suponiendo que $V = \{n_1, \ldots, n_m\}$, definimos el conjunto de proposiciones:

$$P = \{p_1^r, \ldots, p_m^r, p^a, 1, \ldots, p_m^a, p_1^b, \ldots, p_m^b\}$$

con tres proposiciones por nodo (una proposición por cada nodo y color) y la fórmula $\varphi = \psi_{\text{color}} \wedge \psi_{\text{unico}} \wedge \psi_{\text{arista}}$, con las subfórmulas definidas de la siguiente forma:

$$\psi_{\text{color}} = \bigwedge_{i \in [1,m]} p_i^r \vee p_i^a \vee p_i^b$$

$$\psi_{\text{unico}} = \bigwedge_{1 \in [1,m]} (p_i^r \leftrightarrow (\neg p_i^a \wedge \neg p_i^b)) \wedge (p_i^a \leftrightarrow (\neg p_i^r \wedge \neg p_i^b)) \wedge (p_i^b \leftrightarrow (\neg p_i^r \wedge \neg p_i^a))$$

$$\psi_{\text{arista}} = \bigwedge_{(n_i,n_j) \in E} \neg(p_i^r \wedge p_j^r) \wedge \neg(p_i^a \wedge p_j^a) \wedge \neg(p_i^b \wedge p_j^b)$$

Intuitivamente, ψ_{color} asegura que al menos una entre p_i^r, p_i^a y p_i^b es verdad para cada nodo n_i en V, ψ_{unico} asegura que a lo más una entre p_i^r, p_i^a y p_i^b es verdad para cada nodo n_i en V. Es decir, tenemos la siguiente observación:

Observación 1. Sea $\tau : P \to \{0, 1\}$ una valuación que satisface a φ. Entonces de τ podemos deducir una forma de colorear cada nodo en G únicamente por un color: a el nodo n_i le asignamos el color r si y solo si $\tau(p_i^r) = 1$, el color a si y solo si $\tau(p_i^a) = 1$ o el color b si y solo si $\tau(p_i^b) = 1$.

Nota que ψ_{color} asegura que la coloración le asigna un color a cada nodo, y ψ_{unico} que no le asignamos más de uno. Finalmente, ψ_{arista} asegura que para cada arista (n_i, n_j) en E no puede ser que p_i^c y p_j^c sean verdad, para c un color en $\{r, a, b\}$. Deducimos entonces:

Observación 2. Sea $\tau : P \to \{0, 1\}$ una valuación que satisface a φ, y $f : V \to \{r, g, b\}$ la coloración descrita en la **Observación 1**. Entonces para cada arista (n_i, n_j) en G, los nodos n_i y n_j son asignadas colores distintos. Es decir, $f(n_i) \neq f(n_j)$ para $i \in [1, m]$.

Demostramos ahora:

Observación 3. El grafo G es 3-coloreable si y solo si φ es satisfacible.

Demostración. Primero mostramos ($\Leftarrow$): Si φ es satisfacible entonces G es 3-coloreable. Pero esto sale directo de las **Observaciones 1** y **2**: si φ es satisfable entonces hay una valuación τ que la satisface, y de ahí pasamos por las observaciones para concluir que la coloración f muestra que G es 3-coloreable. Ahora mostramos ($\Rightarrow$): Si $G = (V, E)$ es 3-coloreable entonces φ es satisfacible. Sea $f : V :\to \{r, a, b\}$ una coloración que muestre que G es 3-coloreable. De f construimos una valuación τ para φ: para cada $i \in [1, m]$, si $f(n_i) = r$ entonces $\tau(p_i^r) = 1$ y $\tau(p_i^a) = \tau(p_i^b) = 0$. Si $f(n_i) = a$ entonces $\tau(p_i^a) = 1$ y $\tau(p_i^r) = \tau(p_i^b) = 0$. O si $f(n_i) = b$ entonces $\tau(p_i^b) = 1$ y $\tau(p_i^a) = \tau(p_i^r) = 0$. Como f es una función, por construcción se cumple que τ satisface tanto a ψ_{color} como a ψ_{unico}. Además, como f muestra que G es 3-coloreable, $f(n_i) \neq f(n_j)$ para cada arista (n_i, n_j) en E. Nuevamente, esto garantiza que τ satisface a ψ_{arista}.

¿Qué aprendimos de todo esto? Si tengo un grafo, y quiero saber que es 3-colorable, puedo transformar ese grafo a una fórmula, y correr SAT en esa formula. Eso significa que con SAT también puedo resolver 3-colorabilidad! Vamos a ver en el curso muchos otros problemas en los que podemos usar SAT como estrategia para resolverlos. Pero ahora veamos los algoritmos que disponemos.

3. Algoritmos para SAT

Si tenemos una formula φ en $L(P)$, y queremos ver si φ es satisfacible, tenemos las siguientes opciones para resolver ese problema.

- Fuerza bruta: probar todas las valuaciones para P. Si encontramos una que satisface a φ, entonces φ es satisfacible, de lo contrario no lo es.
- Resolución: tomar φ, pasar a CNF, y ver que $\varphi \vdash \bot$. Si efectivamente $\varphi \vdash \bot$ entonces no es satisfacible. Si $\varphi \nvdash \bot$, entonces debe haber al menos una valuación que satisface a φ (notar que este algoritmo no nos da la valuación).
- Resolver SAT con algunas heurísticas (vamos a ver la más clásica ahora).

3.1. Algoritmo DPLL

Una versión simple, pero que aún funciona (mejorada) en la práctica, es el algoritmo de Davis–Putnam–Logemann–Loveland (DPLL). Este algoritmo usa programación dinámica, y lo definimos así. Recuerda que un *literal* es una proposición, o la negación de una proposición. Los literales p y $\neg p$, para la misma proposición p, son llamados literales contradictorios.

Input. Fórmula φ en CNF usando proposiciones $P = \{p_1, \ldots, p_n\}$.
Output. `true` si φ es satisfacible, `false` si no lo es.

El algoritmo está dado por la función DPLL(i, φ), que recibe un $i \in [1, n]$, y una fórmula φ (nuevamente extendemos la lógica proposicional para permitir los valores 0 y 1, representando el falso y el verdadero), y retorna `true` o `false`.

Casos base. (1) Si todas las proposiciones que quedan en la fórmula φ aparecen como p o todas como $\neg p$ (es decir, φ no contiene ninguna copia de dos literales contradictorios), retornamos `true`. (2) Si existe al menos una cláusula de la forma $(0 \vee \cdots \vee 0)$, retornamos `false`.

Caso general. Sea $\varphi_{(i,1)}$ la formula que resulta de reemplazar cada literal p_i en φ por 1 y cada literal $\neg p_i$ en φ por 0.
Sea $\varphi_{(i,0)}$ la formula que resulta de reemplazar cada literal p_i en φ por 0 y cada literal $\neg p_i$ en φ por 1.
Retornar $\big(\text{DPLL}(i+1, \varphi_{(i,1)})$ OR $\text{DPLL}(i+1, \varphi_{(i,0)})\big)$.

4. Ejercicio Propuesto

Muestra que el algoritmo DPLL es correcto y completo: Con input φ, retorna `true` si y solo si φ es satisfacible. Como ayuda, puedes mostrar por inducción alguna propiedad sobre $\text{DPLL}(i+1, \varphi_{(i,1)})$ y $\text{DPLL}(i+1, \varphi_{(i,0)})$.

Este algoritmo está expuesto acá de una forma concisa, pero no da luces sobre implementaciones eficientes. En particular, una optimización inmediata es que las cláusulas que

ya son verdad no deben volver a ser visitadas, y eso podría llevar a eliminar la necesidad de ver algunas otras variables. ¿Puedes pensar en como definir esta versión optimizada del algoritmo?.