Facultad de Ciencias - UNAM
Lógica Computacional 2023-2
Práctica 3: Algoritmo DPLL

Javier Enríquez Mendoza
Ramón Arenas Ayala
Óscar Fernando Millán Pimentel
Kevin Axel Prestegui Ramos

13 de Marzo de 2023
Fecha de entrega: 26 de Marzo de 2023

1. Introducción

Como se ha mencionado a lo largo del curso se abordará el tema de la **satisfacibilidad** booleana, utilizando diferentes algoritmos y teoremas para ello.

Para esto tenemos un Teorema que nos ayudará. El Teorema de Cook es de gran importancia, pues además de introducir el concepto de NP-completud establece que el problema de satisfacibilidad booleana es un problema NP-completo y de hecho, el primero en ser demostrado que pertenece a esta clase de complejidad.

Al ser un problema NP-completo el problema de satisfacibilidad booleana, no existe un algoritmo eficiente que dada una instancia de este problema, pueda resolverlo. Sin embargo a lo largo de los años se han desarrollado sistemas capaces de "resolver" instancias del problema SAT de manera eficiente. A estos sistemas se les llama solucionadores SAT.

Muchos de los solucionadores SAT modernos usan como base al algoritmo DPLL, que es un algoritmo basado en la técnica de backtracking. Con el paso del tiempo, el algoritmo ha sido modificado con el fin de hacerlo más eficiente, sin embargo, para propósitos didácticos en este capítulo se revisa una versión simplificada de éste.

2. Objetivo

El objetivo principal de esta práctica es la implementación del algoritmo DPLL (Davis-Putnam-Logemann-Loveland) como solución para el problema de **satisfacibilidad** booleana, además de poder dar solución a un problema reduciéndolo a un problema de satisfacibilidad booleana.

3. Justificación

El problema del SAT es un problema relevante en el estudio de la lógica computacional, el poder resolverlo tambíen es parte importante, así el algoritmo DPLL nos permite alcanzar una solución, para ello es importante estudiarlo y tener una noción de su implementación.

4. Desarrollo de la práctica

4.1. Problema SAT

Para poder entender que quiere resolver el algoritmo DPLL primero debemos explicar el problema de *satisfacibilidad booleana* o *SAT*. De manera general el problema *SAT* busca responder lo siguiente:

Dada una fórmula proposicional φ, ¿existe un estado de las variables I de tal manera que suceda que $I(\varphi) = 1$?

El problema del *SAT* puede servirnos para resolver problemas tan simples como un sudoku o tan complicados como verificación de hardware, de planeación, así como criptoanálisis, además de formas de ser un gran objeto de estudio para la teoría de la *Complejidad*.

4.2. Implementación de algoritmo DPLL (Parte 1)

La implementación del aloritmo *Davis-Putnam-Logemann-Loveland* (DPLL) es muy imporante como objeto de estudio ya que muchos de los solucionadores *SAT* utilizan alguna varación de este aloritmo el cual que sirve para decidir la satisfacibilidad de una fórmula proposicional en forma normal conjuntiva. Aunque existen diversas variantes del algoritmo, todas se basan en el mismo conjunto de reglas.

El procedimiento DPLL es un algoritmo de decisión, es decir, dada una fórmula proposicional éste determina si es o no satisfacible, pero no da el modelo o los modelos que satisfacen a la fórmula. No obstante existe una versión del algoritmo que además de determinar si la fórmula es o no satisfacible, como parte del resultado se obtiene un modelo en caso de que la fórmula sea satisfacible.

El algoritmo se modela como un sistema de transiciones, en donde cada estado $M \models F$ representa la búsqueda de un modelo M para F.

En HASKELL estos estados se puede representar mediante un par, en donde la primer proyección corresponde a una `Interpretacion`, mientras que la segunda proyección corresponde a una `Fórmula`.

Para la implementación vamos a utilizar los modelos en*forma normal conjuntiva*, utilizados en la práctica anterior, los cuales serán modelados con clausilas que a su vez son listas de literales.

```
type Interpretacion = [( String , Bool ) ]
type Estado = ( Interpretacion , [ Clausula ])
```

Por otro lado, las transiciones entre estados, que se modelan como $M \models_? F \quad \triangleright \quad M' \models_? F'$, se representan mediante de la aplicación de reglas. Para ello desarrollaremos las reglas que componen el algoritmo:

Ejercicios

- *Conflicto* (`conflict :: Estado ->Bool`)

  Determina si la claúsula vacía, representada como una lista vacía, forma parte del conjunto de claúsulas. Puesto que no existe modelo que satisfaga a la claúsula vacía, la búsqueda del modelo falla.
  Ejemplo

  ```
  ghci >conflict ([] , [[p,q] , [r,s] , [] , [p,r,t]]) ▷ True
  ```

- *Éxito* (`success :: Estado ->Bool`)

  Determina si la búsqueda del modelo ha sido exitosa, esto sucede cuando el conjunto de cláusulas es vacío.
  Ejemplo

  ```
  ghci >success ([( p , True ) , (q , False ) ] , []) ▷ True
  ```

- *Claúsula unitaria* (`unit :: Estado ->Estado`)

  Si ℓ es una literal que pertenece a una claúsula unitaria, entonces basta con agregar ℓ al modelo y seguir con la búsqueda. Es importante notar que si $\ell^c \in M$ esta regla no puede aplicarse, de lo contrario se necesitaria que tanto ℓ como ℓ^c sean verdaderas, lo que conduce a una contradicción.
  Nota: Para esta regla solo aplica la primer cláusula unitaria encontrada

  Ejemplo

  ```
  ghci >unit ([] , [[p,q] , [r,s] , [l]]) ▷
     ([( l , True ) ] , [[p,q] , [r,s]])
  ```

- *Eliminación* (`elim :: Estado ->Estado`)

  Si ℓ es una literal que pertenece al modelo M y se tiene la claúsula $\ell \vee C$ entonces, dado que ℓ es verdadera, $\ell \vee C$ también lo es, por lo que se elimina la claúsula $\ell \vee C$ del conjunto de cláusulas.
  Nota: Esta regla se aplica para cada una de las literales en la interpretación

  Ejemplo

  ```
  ghci >([( p , True ) ] , [[p,q] , [r,s] , [p,r,t]]) ▷
              ([( p , True ) ] , [[r,s]])
  ```

- *Reducción:* (`red :: Estado ->Estado`)

  Si ℓ es una literal que pertenece al modelo M y se tiene la claúsula $\ell^c \vee C$ entonces, dado que ℓ es verdadera, ℓ^c es falsa, por lo que solo es de interés saber si C es satisfacible.
  Nota: Esta regla se aplica para cada una de las literales en la interpretación

Ejemplo

```
ghci >red ([( p , True ) ] , [[not p,q] , [r,s] , [p,r,t]]) ▷
         ([( p , True ) ] , [[q] , [r,s] , [p,r,t]])
```

- *Separación* (`sep :: Literal ->Estado ->(Estado, Estado)`)

  Dada una literal ℓ se procede a buscar que M, ℓ sea modelo de F, o que M, ℓ^c lo sea.

  Ejemplo

```
                ghci >sep p ([] , [[ p , q ]]) ▷
(([( p , True ) ] , [[p,q]]) , ([( p , False ) ] , [[p,q]]) )
```

  Obsérvese como en esta regla se generan dos caminos a explorar, uno con M, ℓ y otro con M, ℓ^c . De modo que si el primer camino falla en la búsqueda se procede a la búsqueda del segundo camino, si éste también falla entonces no hay modelos.

4.3. Árboles DPLL

Para poder implementar el algoritmo DPLL de manera sencilla, sin tener problemas de tipos debido a la regla de separación, implementaremos unos árboles binarios DPLL, los cuales además de darnos soporte para poder manejar el caso de la regla de separación, nos permitirán analizar de manera clara el *Backtracking* utilizado en el algoritmo. El tipo será el siguiente:

```
data ArbolDPLL = Node Estado ArbolDPLL | Branch Estado ArbolDPLL ArbolDPLL
```

Este tipo de dato es mandatario que se utilice en la implementación del algoritmo junto con las reglas anteriores.

4.4. Implementación de algoritmo DPLL (Parte 2)

Al ser DPLL un algoritmo no determinista, debemos definir una *heurística* para decidir sobre qué literal se aplicará la regla de separación. Para esto definiremos la siguiente función:

```
heuristicsLiteral :: [Clausula] ->Literal
```

Dada una fórmula proposicional nos regresa la literal que más apariciones tiene en la fórmula.

Así, usando los ejercicios definidos previamente, la función principal sería:

```
dpll :: [Clausula] ->Interpretacion
```

Que recibe una fórmula proposicional en forma clausular y devuelve una interpretación que satisfaga a la fórmula obtenida mediante la ejecución del algoritmo DPLL iniciando la ejecución con el estado $\emptyset \models F$. En caso de que la fórmula no sea satisfacible, la función deberá devolver una lista vacía. Es fundamental recordar que el algoritmo trabaja con las cláusulas de una fórmula proposicional en forma normal conjuntiva.

5. Especificaciones de entrega

- **GitHub Classroom:** Para realizar la entrega de la práctica se utilizará la plataforma de GitHub Classroom. `https://classroom.github.com/a/6509fEZB`
- **Equipos:** La práctica puede realizarse en equipos de hasta tres personas. Esto siempre y cuando se respete la política de trabajo colaborativo y podamos ver que ambos miembros del equipo contribuyen en la elaboración de la práctica. En caso contrario, las prácticas serán exclusivamente individuales.
- **Fecha de entrega:** La fecha de entrega será la indicada al principio de este documento. No se recibirá ninguna práctica en fechas posteriores a la fecha indicada, al menos que el profesor indique una prórroga.
- **Flujo de trabajo:** Se recomienda llevar un flujo de trabajo adecuado, en el cual se realicen commits constantemente y no hacer todo al final y en un solo commit. Esto ya que de esta manera podemos comprobar la colaboración de todos los miembros del equipo, dar feedback a lo largo del tiempo de trabajo para mejorar la entrega de las prácticas, facilitar la resolución de dudas, etc.
- **Evaluación:** Para la evaluación la práctica se someterá a un conjunto de pruebas, y además se realizará una revisión del código. Comprobándose que se cumplan las condiciones indicadas para cada ejercicio.
- **Compilado:** Cualquier práctica que al ser descargada no compile, **será evaluada con una calificación de 0**.
- **Datos personales:** Se debe agregar un documento `README.md` en el directorio raíz de sus repositorios, con sus datos personales. En caso de no ser indicados, la calificación no podrá ser asignada.
- **Limpieza y estructura:** en caso de no tener una estructura clara y limpieza dentro de sus repositorios, la calificación se verá afectada negativamente.

6. Sugerencias y Notas

- **Dudas**: Pueden preguntar sus dudas a través de telegram, o correo por el canal que ustedes deseen. Pero les recomendamos preguntar a través del grupo de telegram para que sus demás compañeros también puedan aclarar dudas similares a la suya.
- **Limitaciones**: Pueden utilizar funciones predefinidas siempre y cuando su utilización no derive en que se pierda el objetivo académico del ejercicio que se está realizando. Si se tiene duda acerca del poder utilizar algo, pueden preguntarlo.

Buena suerte a todos! ☺☺☺