Práctica 3 de Algorítmica
(Búsqueda con retroceso)

Algoritmo
Davis-Putnam-Logemann-Loveland

Curso 2017-2018

Problema de satisfacibilidad booleana

El problema SAT consiste en determinar si para una expresión booleana con variables existe una asignación de valores para dichas variables que haga que la expresión sea cierta.

Un ejemplo de este problema es ver si es posible dar valores cierto/falso a las variables x_1,x_2,x_3,x_4 tales que la fórmula:

$(x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$

se evalúa a cierto (donde $\overline{x_i}$ representa a la variable x_i negada).

Vamos a limitarnos exclusivamente a expresiones lógicas en **forma normal conjuntiva** (también conocida por sus siglas en inglés **CNF**).

Utilizaremos la siguiente nomenclatura:

- Un **literal** es una variable o una variable negada. En algunos contextos a las variables se las denomina *átomos*.
- Una **cláusula** es una disyunción de literales.
- Una **fórmula en formato CNF** es una conjunción de cláusulas.

Este problema ha sido históricamente el primero identificado como perteneciente a la clase de complejidad NP-completo aunque se restrinja el número de literales por cláusula a un máximo de 3.

Resolviendo el problema SAT en fórmulas CNF

Este problema tiene aplicaciones prácticas. Se utiliza, por ejemplo, para verificación de modelos (o model checking), para planificación automática y para diagnóstico en inteligencia artificial.

Existen diversas aproximaciones para resolver el problema de satisfacibilidad.

Por ejemplo, se puede aplicar *fuerza bruta* y probar todas las combinaciones de maneras de asignar valores de verdad a las variables. Obviamente, esto resulta enormemente ineficiente ya que el número de combinaciones es 2^n siendo n el número de variables.

Los algoritmos utilizados en la práctica para resolver este problema son variantes del algoritmo DPLL que vamos a estudiar en esta práctica.

El algoritmo DPLL, abreviatura de Davis-Putnam-Logemann-Loveland data de los años 60 y utiliza la estrategia algorítmica de búsqueda con retroceso, vuelta a trás o backtracking.

Forma normal conjuntiva en formato DIMACS

Vamos a probar los algoritmos de satisfacibilidad utilizando instancias guardadas en un formato conocido como “DIMACS CNF file format”. Puedes encontrar varias descripciones de este formato en la red (pincha en el siguiente enlace, por ejemplo). Un fichero en este formato tiene el siguiente aspecto:

```
c  simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

- Las líneas que empiezan con la letra `c` son comentarios y se deben ignorar.
- La primera línea que no es un comentario empieza con la letra `p`:

  ```
  p cnf 3 2
  ```

  y contiene: la propia `p`, la cadena `cnf` y después 2 números enteros que indican, respectivamente, el número de variables y el número de cláusulas. En la ecuación del ejemplo hay 3 variables y 2 cláusulas.
- La fórmula sería la conjunción de todas esas cláusulas. El resto del fichero, salvo comentarios, corresponde a una cláusula por línea.

Forma normal conjuntiva en formato DIMACS

Cada cláusula viene representada como un conjunto de números enteros donde los negativos representan una variable negada y donde el valor `0` se reserva para indicar el fin de la cláusula. Por ejemplo, la línea

```
1 -3 0
```

indica que la cláusula es la disyunción de $x1$ y de $\overline{x_3}$ (x_3 negada porque el valor es negativo). El `0` del final sirve únicamente para delimitar la cláusula.

Por tanto, el fichero:

```
c  simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

representa la fórmula:

$(x_1 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_1})$

La función que se os proporciona para cargar una fórmula CNF devuelve una lista de cláusulas, donde cada cláusula es una lista de enteros. Por ejemplo, la fórmula del ejemplo anterior se representaría mediante la siguiente lista python:

```
[[1,-3],[2,3,-1]]
```

Resolviendo el problema SAT en fórmulas CNF

Ejercicio 1 Implementa la función `simplify` que recibe una fórmula, un literal y devuelve la fórmula simplificada al suponer que el literal es cierto. Observaciones:

- Una cláusula es una disyunción de literales. El neutro de la disyunción ("o" lógica) es el valor falso. Por tanto, una cláusula vacía es falsa.
- Una fórmula CNF es una conjunción de cláusulas. El neutro de la conjunción ("y" lógica) es el valor cierto. Por tanto, una fórmula sin cláusulas es cierta.
- Podemos suponer sin pérdida de generalidad que ninguna cláusula tiene simultáneamente un literal y su literal negado. Si esto ocurriera, basta con eliminar dicha cláusula ya que $x \vee \overline{x}$ siempre es cierto.

Cuando suponemos que un literal es cierto, debemos:

- Eliminar de la fórmula las cláusulas donde aparece dicho literal.
- Eliminar el literal negado en las cláusulas donde éste aparece.

Por ejemplo, si en la fórmula: $(x_1 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_1})$ suponemos que x_1 es cierto, eliminamos la clásula $(x_1 \vee \overline{x_3})$ y eliminamos el literal $\overline{x_1}$ de la cláusula $(x_2 \vee x_3 \vee \overline{x_1})$. De esta manera, la fórmula quedará así: $(x_2 \vee x_3)$. Así,

```
simplify([[1,-3],[2,3,-1]],1)
```

debe devolver la lista `[[2,3]]`.

- Si aparece alguna cláusula vacía, la función debe devolver `False`.
- Si la conjunción de cláusulas queda vacía, deberá devolver `True`.

Resolviendo el problema SAT en fórmulas CNF

Una primera aproximación para resolver SAT mediante backtracking sería tal y como refleja el siguiente pseudo-código donde suponemos que `simplify` devuelve la fórmula simplificada excepto en dos casos: `True` cuando la fórmula se vuelve cierta y `False` cuando se vuelve falsa.

```
def backtracking(formula):
  literal = choose_literal(formula)
  for choice in (literal,-literal):
    f = simplify(formula,choice)
    if f is not False:
      if f is True:
        return True
      resul = backtracking(f)
      if resul:
        return True
  return False
```

La función `choose_literal` elige un literal de la fórmula.

Atención

- Utilizar `choose_literal` es una mejora respecto a seguir un orden predeterminado ya que la simplificación con un literal puede eliminar otros literales.
- La correcta elección de literales tiene una influencia muy importante en la eficiencia de este algoritmo.

Observación general:

Es **importante** que tu función `simplify` no modifique la fórmula recibida (recuerda que en Python una función que recibe una lista realmente recibe una referencia a la misma). Tu función ha de crear una lista nueva.
Ten en cuenta que en python las listas son referencias a estructuras mutables y no se consideran valores. Esto queda claro con el siguiente ejemplo:

```
>>> def f(a): a.reverse()
...
>>> a = [1,2,3]
>>> f(a)
>>> a
[3, 2, 1]
>>> b = a
>>> b.append(5)
>>> a
[3, 2, 1, 5]
```

Por tanto, has de tener precaución para que las funciones realizadas no modifiquen la lista de cláusulas que reciben, ya que ésta se puede utilizar más de una vez en el caso de backtracking.

Ejercicio 2 Implementa este algoritmo básico de backtracking modificándola para que *recuperar la secuencia de literales asignados para que la fórmula original se vuelva cierta*. Para ello has de hacer que `backtracking` devuelva `None` si no hay solución o bien una lista con los literales que hacen que la fórmula se vuelva cierta.

Atención

- El código que se proporciona ya contiene la cabecera de dicha función así como un pequeño test que comprueba si el resultado es correcto. No obstante, dicho código hace uso de tu función `simplify`, por lo que su corrección depende también de que tu código `simplify` sea correcto.

Veremos que el algoritmo DPLL añade a este esquema básico de backtracking 2 mejoras significativas:

- **Unit propagation:** Si una cláusula contiene un único literal (una cláusula unitaria), sabemos de antemano que la única forma de que la fórmula tenga solución es asignar a cierto dicho literal. Esto es un proceso determinista que no requiere búsqueda.
- **Pure literal elimination:** Si una variable aparece únicamente negada o únicamente sin negar en todas las clásulas, podemos asignarle el valor cierto a ese literal y así eliminar las cláusulas en que aparece de manera determinista (al igual que la eliminación de cláusulas unitarias, sin necesidad de realizar ningún tipo de búsqueda).

Ejercicio 3

- Implementa una función `unit_propagation` que elimine todas las cláusulas unitarias de manera adecuada (dando valor de verdad a esos literales y simplificando).
- La función deberá devolver 2 valores (una tupla con 2 elementos):
  - El primero es la fórmula simplificada o bien un booleano (si aparece algúna cláusula vacía, la función devolverá el valor `False`, pero si por el contrario se vacía la lista de cláusulas, debe devolver `True`).
  - El segundo es una lista con los literales asignados (i.e. un valor negativo si es un átomo negado).

Un posible pseudo-código del algoritmo:

```
def unit_propagation(clausulas):
  asignados=[]
  mientras existan cláusulas unitarias:
    tomar una cláusula unitaria [L]:
      asignados.append(L)
      eliminar las cláusulas que contienen el literal L
      eliminar del resto de cláusulas el literal -L
      si el cjt de clásulas contiene []:
        devolver False,[]
      si el cjt de clásulas queda vacío:
        devolver True,asignados
  devolver las clásulas,asignados
```

Observaciones:

- Cuando devuelve `False` daría igual la lista de asignados.
- Al tratarse de un proceso determinista en la búsqueda por backtracking, no habría problema en principio en modificar la lista de cláusulas recibida.
- Ten en cuenta que cuando eliminas un literal de una cláusula ésta puede convertirse en unitaria, con lo que habría que procesarla también en el propio algoritmo, y eso incluye tener en cuenta también las cláusulas que ya se han procesado hasta el momento.

Una posible estrategia para dar cuenta de la última observación es crear una lista de las unitarias que aparecen durante una iteración e iterar el algoritmo de nuevo con las cláusulas ya simplificadas mientras esa lista no sea vacía.

Ejercicio 4

- Implementa una función `pure_literal_elimination` que detecte los literales que no aparecen también negados y que los elimine de la fórmula de manera adecuada (dándoles valor de verdad y simplificando la fórmula).
- La función deberá devolver 2 valores (una tupla con 2 elementos):
  - El primero es la fórmula simplificada o bien un booleano (si aparece algúna cláusula vacía, la función devolverá el valor `False`, pero si por el contrario se vacía la lista de cláusulas, debe devolver `True`).
  - El segundo es una lista con los literales asignados.

El pseudo-código del algoritmo para eliminar los literales puros:

```
def pure_literal_elimination(cláusulas):
  positivos = cjt de literales positivos que aparecen en las cláusulas
  negativos = cjt de literales negativos que aparecen en las cláusulas
  puros = los literales que aparecen en un cjt y no negados en el otro
  eliminar las cláusulas que tengan algún literal puro
  si la lista de cláusulas está vacía:
    devolver True,puros
  devolver clausulas,puros
```

Ejercicio 5 Implementa la función `dpll` que básicamente consiste en extender backtracking aplicando unit propagation y luego pure literal elimination dentro del algoritmo recursivo antes de probar a simplificar con una variable o con su negada.