UNIVERSIDAD DE SANTIAGO DE CHILE

Departamento de Matemática y Ciencia de la Computación

Tarea 1 - SAT Lineal

Miguel Olivares Morales
miguel.olivares@usach.cl

Benjamín Riveros Landeros
benjamin.riveros.l@usach.cl

Lógica Computacional - 22625
Licenciatura en Ciencia de la Computación

Semestre Otoño 2025

1 Introducción

El problema de determinar si las variables de una fórmula booleana pueden ser reemplazadas con valores **T** o **F** de tal forma que la fórmula de como resultado **T** se denomina problema de satisfacibilidad booleana o SAT. Si al evaluar la fórmula esta da como resultado **T**, entonces se dice que es satisfactoria.

2 Procedimiento

Las fórmulas que serán analizadas primero tendrán que ser codificadas según la siguiente gramatica:

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi)$$

Para esto usamos el siguiente esquema de traducción:

$T(p) = p$

$T(\neg\phi) = \neg T(\phi)$

$T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$

$T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$

$T(\phi_1 \to \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2))$

Esto quiere decir que se analizarán fórmulas compuestas por proposiciones atómicas, negaciones de otras fórmulas y conjunciones de dos fórmulas.

Luego de codificar se tiene que transformar a su notación postfix o también llamada *notación polaca inversa* con la cual facilitará la creación de un *parse tree* para asignar valores **T** o **F** a cada nodo. Al tener el parse tree correspondiente a la fórmula que se evalúa asignamos **T** al nodo que encabeza el árbol. Esto implica asumir que la fórmula completa es verdadera y a partir de ello se puede extender esta asignación hacia los nodos hijos del árbol aplicando reglas semánticas de los conectores lógicos.

Si el nodo principal es una conjunción $\phi \wedge \psi$ entonces ϕ y ψ deben ser verdaderas. Por el contrario, si el nodo es una negación $\neg\phi$ quiere decir que la subfórmula ϕ es falsa. Este procedimiento se aplica recursivamente hasta llegar a los nodos hoja, los cuales corresponden a átomos proposicionales.

De esta forma se obtiene una asignación de valores de verdad que satisface la fórmula. En caso que las asignaciones conduzcan a una contradicción (por ejemplo, se tiene $p \equiv \mathbf{T}$ y $\neg p \equiv \mathbf{T}$) se descarta el camino recorrido o incluso puede significar que la fórmula es *insatisfacible*.

Adicionalmente para una mayor eficiencia en espacio y tiempo, detectar y reutilizar átomos proposicionales podemos construir en cambio un DAG (Directed Acyclic Graph).

2.1 Ejemplo

Dada la siguiente fórmula:

$$((p \to q) \wedge (\neg r \vee p))$$

El primer paso es aplicar la codificación mencionada anteriormente

$$\begin{aligned}
\phi &= ((p \to q) \wedge (\neg r \vee p)) \\
T(\phi) &= T(((p \to q) \wedge (\neg r \vee p))) \\
&= T(p \to q) \wedge T(\neg r \vee p) \\
&= \neg(T(p) \wedge \neg T(q)) \wedge \neg(\neg T(\neg r) \wedge \neg T(p)) \\
T(\phi) &= \neg(p \wedge \neg q) \wedge \neg(\neg\neg r \wedge \neg p)
\end{aligned}$$

El siguiente paso es transformar la fórmula codificada a su notación postfix, para esto hay que descomponer la fórmula en *tokens* de la siguiente manera:

$$[\neg, (, p, \wedge, \neg, q,), \wedge, \neg, (, \neg, \neg, r, \wedge, \neg, p,)]$$

Para transformas a su notación postfix se tiene que saber que se evalúan los operadores según la precedencia, en donde la negación tiene la mayor precedencia por lo que se evalúa primero y despúes la conjunción.

Token	**Acción**	**Salida**	**Stack**
$\neg$	Apilar operador		$\neg$
(	Apilar paréntesis		$\neg$, (
p	Agregar a salida	p	$\neg$, (
$\wedge$	Apilar operador	p	$\neg$, (, $\wedge$
$\neg$	Apilar operador	p	$\neg$, (, $\wedge$, $\neg$
q	Agregar a salida	$p\ q$	$\neg$, (, $\wedge$, $\neg$
)	Desapilar hasta (	$p\ q\ \neg\ \wedge$	$\neg$
$\wedge$	Apilar operador	$p\ q\ \neg\ \wedge$	$\neg$, $\wedge$
$\neg$	Apilar operador	$p\ q\ \neg\ \wedge$	$\neg$, $\wedge$, $\neg$
(	Apilar paréntesis	$p\ q\ \neg\ \wedge$	$\neg$, $\wedge$, $\neg$, (
$\neg$	Apilar operador	$p\ q\ \neg\ \wedge$	$\neg$, $\wedge$, $\neg$, (, $\neg$
$\neg$	Apilar operador	$p\ q\ \neg\ \wedge$	$\neg$, $\wedge$, $\neg$, (, $\neg$, $\neg$
r	Agregar a salida	$p\ q\ \neg\ \wedge\ r$	$\neg$, $\wedge$, $\neg$, (, $\neg$, $\neg$
$\wedge$	Apilar operador	$p\ q\ \neg\ \wedge\ r$	$\neg$, $\wedge$, $\neg$, (, $\neg$, $\neg$, $\wedge$
$\neg$	Apilar operador	$p\ q\ \neg\ \wedge\ r$	$\neg$, $\wedge$, $\neg$, (, $\neg$, $\neg$, $\wedge$, $\neg$
p	Agregar a salida	$p\ q\ \neg\ \wedge\ r\ p$	$\neg$, $\wedge$, $\neg$, (, $\neg$, $\neg$, $\wedge$, $\neg$
)	Desapilar hasta (	$p\ q\ \neg\ \wedge\ r\ p\ \neg\ \wedge\neg\neg$	$\neg$, $\wedge$, $\neg$
	Vaciar pila	$p\ q\ \neg\ \wedge\ r\ p\ \neg\ \wedge\neg\ \neg\ \neg\ \wedge\ \neg$	

3 Algoritmo

3.1 Codificación de fórmula

Algorithm 1: Algoritmo recursivo $T(\phi)$ para codificar una fórmula booleana ϕ

Input: ϕ
Output: $T(\phi)$
if ϕ *es un átomo proposicional* p **then**
  return p
end
if ϕ *es una negación* $(\neg\phi)$ **then**
  return $\neg T(\phi)$
end
if ϕ *es una conjunción* $(\phi_1 \wedge \phi_2)$ **then**
  return $T(\phi_1) \wedge T(\phi_2)$
end
if ϕ *es una disyunción* $(\phi_1 \vee \phi_2)$ **then**
  return $\neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$
end
if ϕ *es una implicancia* $(\phi_1 \rightarrow \phi_2)$ **then**
  return $\neg(T(\phi_1) \wedge \neg T(\phi_2))$
end
if ϕ *tiene una fórmula no reconocida* **then**
  Reportar error: fórmula inválida
end

3.2 Conversión a Notación Postfija

Para transformar las fórmulas lógicas desde su forma infija a una notación postfija (también conocida como notación polaca inversa), se implementó un algoritmo inspirado en el *Shunting Yard Algorithm* de Edsger Dijkstra. Esta notación facilita el análisis y la evaluación de fórmulas lógicas, ya que elimina la ambigüedad del orden de operaciones y permite un procesamiento más eficiente.

Algorithm 2: Conversión a notación postfija (inspirado en Shunting Yard Algorithm)

```
Input: Fórmula codificada T(ϕ)
Output: Fórmula T(ϕ) en notación postfija
Inicializar una pila vacía stack
Inicializar una cadena vacía output
for cada token en la fórmula de entrada do
    if el token es un operando then
        Agregar token a output
    end
    else if el token es un operador then
        while pila no vacía y tope operador con mayor o igual precedencia do
            Desapilar operador y agregarlo a output
        end
        Apilar el operador actual
    end
    else if el token es un paréntesis de apertura then
        Apilar el token
    end
    else if el token es un paréntesis de cierre then
        while tope de pila no es paréntesis de apertura do
            Desapilar operador y agregarlo a output
        end
        if pila vacía then
            Reportar error: paréntesis desbalanceados
        end
        else
            Desapilar paréntesis de apertura
        end
    end
end
while pila no vacía do
    if tope es paréntesis then
        Reportar error: paréntesis desbalanceados
    end
    else
        Desapilar operador y agregarlo a output
    end
end
return output
```

4 Implementación

5 Conclusiones