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:

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

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

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

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:

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

El primer paso es aplicar la codificación mencionada anteriormente

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

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:

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

Para transformar 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 luego de esta, la conjunción.

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

3 Algoritmo

3.1 Codificación de fórmula

Para simplificar el procesamiento, transformamos la fórmula booleana en una forma equivalente que usa solo negación y conjunción, eliminando implicaciones y disyunciones mediante equivalencias lógicas. A continuación se muestra algoritmo recursivo $T(\phi)$ aplicando estas reglas:

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
```

3.3 Backtracking

Para encontrar la satisfacibilidad de la fórmula se usa un algoritmo de búsqueda que explora un árbol de posibles soluciones probando distintos caminos hasta encontrar una respuesta válida o se haya explorado todas las posibilidades. Si un camino lleva a contradicción, el algoritmo ”retrocede” a un paso anterior y prueba **T** si antes era **F** o viceversa.

Algorithm 3: Backtracking para verificar si una fórmula CNF es satisfacible

Input: Fórmula $T(\phi)$, asignación booleana $A[0 \ldots n{-}1]$, número de variables n, índice actual i

Output: **true** si existe una asignación que satisface F, **false** en otro caso

if $i = n$ **then**
    $c \leftarrow T(\phi)$ **while** $c \neq nulo$ **do**
        if ***not*** *is_satisfied(c, A, n)* **then**
            return false
        end
        $c \leftarrow c.derecha$
    end
    return true
end
$A[i] \leftarrow$ **true**
if *backtrack(*$T(\phi)$*, A, n, i+1)* **then**
    return true
end
$A[i] \leftarrow$ **false**
if *backtrack(*$T(\phi)$*, A, n, i+1)* **then**
    return true
end
return false

3.4 Satisfacibilidad

Comprueba si una cláusula específica se cumple bajo una asignación de valores de verdad a los átomos.

Algorithm 4: Resuelve el problema de satisfacibilidad (SAT) para una fórmula en CNF

Input: Fórmula $T(\phi)$ en forma normal conjuntiva, número de variables n

Output: **true** si $T(\phi)$ es satisfacible, **false** en otro caso

Crear arreglo booleano $A[0 \ldots n{-}1]$ inicializado en **false**
if *no se puede reservar memoria para* A **then**
    Imprimir “Error al asignar memoria”
    return false
end
$result \leftarrow$ backtrack($T(\phi)$, A, n, 0)
Liberar memoria de A
return $result$

4 Implementación

4.1 Codificación de fórmula

La construcción del árbol sintáctico a partir de la fórmula en notación postfija se basa en el uso de una pila y las operaciones fundamentales `push` y `pop` para manejar los nodos.

- Las **variables** se convierten en nodos hoja y se insertan en la pila.
- Los **operadores unarios** como $\neg$ toman un operando de la pila y crean un nodo unario.
- Los **operadores binarios** como $\wedge$, $\vee$ e $\rightarrow$ extraen dos operandos de la pila, crean un nuevo nodo con la operación correspondiente y lo reinsertan.
- Las **transformaciones lógicas** se aplican para eliminar operadores como $\rightarrow$ y $\vee$ mediante equivalencias:
  - $A \rightarrow B \equiv \neg(A \wedge \neg B)$
  - $A \vee B \equiv \neg(\neg A \wedge \neg B)$

El siguiente fragmento ejemplifica el manejo de operadores y la manipulación de la pila para construir el árbol:

Listing 1: Manejo de pila y reescritura lógica durante la construcción del árbol

```
    "\\rightarrow" {
        Node* right = pop();
        Node* left = pop();
        Node* neg_right = create_not(right);
        Node* and_node = create_op(AND, left, neg_right);
        push(create_not(and_node));
    }

    "\\neg" {
        Node* child = pop();
        push(create_not(child));
    }

    "\\wedge" {
        Node* right = pop();
        Node* left = pop();
        push(create_op(AND, left, right));
    }

    "\\vee" {
        Node* right = pop();
        Node* left = pop();
        Node* neg_left = create_not(left);
        Node* neg_right = create_not(right);
        Node* and_node = create_op(AND, neg_left, neg_right);
        push(create_not(and_node));
    }
```

4.2 Notación Postfix

Esta transformación simplifica el análisis posterior de la fórmula.

El código en C:

- Define operadores lógicos con su precedencia y asociatividad.
- Procesa la entrada carácter por carácter, reconociendo paréntesis, operadores y variables.
- Aplica el algoritmo de Shunting Yard usando una pila.
- Genera la expresión postfija y detecta errores de paréntesis.

Este módulo permite transformar expresiones como `$((p \rightarrow q) \wedge (\neg r \vee p))$` en `p q \rightarrow r \neg p \vee \wedge`.

A continuación, se muestra el fragmento más importante del código:

Listing 2: Fragmento clave de la conversión a notación postfija

```c
Operator operators[] = {
    {"\\neg", 3, 1},        // Mayor precedencia, asociativo a la derecha
    {"\\wedge", 2, 0},      // Conjuncion
    {"\\vee", 1, 0},        // Disyuncion
    {"\\rightarrow", 0, 0},// Implicacion
    {NULL, 0, 0}
};

char* convert_to_postfix(const char* input) {
    char output[MAX_OUTPUT] = "";
    const char* stack[MAX_STACK];
    int top = 0;

    // Procesamiento de tokens y aplicacion del algoritmo de Shunting Yard
    while (...) {
        // Manejo de operadores
        if (is_operator(token)) {
            while (top > 0 && is_operator(stack[top - 1]) &&
                ((precedence(stack[top - 1]) > precedence(token)) ||
                (precedence(stack[top - 1]) == precedence(token) &&
                !is_right_associative(token)))) {
                strcat(output, stack[--top]);
                strcat(output, " ");
            }
            stack[top++] = strdup(token);
        }
        // Manejo de parentesis y operandos...
    }

    // Vaciar la pila al final
    while (top > 0) {
```

```
            strcat(output, stack[--top]);
            strcat(output, " ");
        }

        return strdup(output);
    }
```

4.3 Construcción del Árbol Sintáctico

La representación de una fórmula lógica se realiza mediante estructuras de árbol donde cada nodo representa un operador o una variable.

- Las **variables proposicionales** como `P` o `Q` se almacenan como nodos hoja.
- Los **operadores binarios** (`AND`, `OR`, `IMPL`) generan nodos internos con dos hijos.
- El **operador unario** `NOT` crea un nodo con un solo hijo.

Listing 3: Creación de nodos para el árbol sintáctico

```
    Node* create_var(const char* name) {
        Node* node = (Node*)malloc(sizeof(Node));
        node->type = VAR;
        node->var_name = strdup(name);
        node->left = node->right = NULL;
        return node;
    }

    Node* create_op(NodeType type, Node* left, Node* right) {
        Node* node = (Node*) malloc(sizeof(Node));
        node->type = type;
        node->left = left;
        node->right = right;
        node->var_name = NULL;
        return node;
    }

    Node* create_not(Node* child) {
        Node* node = (Node*) malloc(sizeof(Node));
        node->type = NOT;
        node->left = child;
        node->right = NULL;
        node->var_name = NULL;
        return node;
    }
```

4.4 Copia y Liberación de Árboles

Se incluyen funciones auxiliares para copiar un árbol lógico o liberar toda su memoria, evitando fugas.

Listing 4: Copia y liberación de árboles lógicos

```c
Node* copy_tree(Node* node) {
    if (!node) return NULL;

    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->type = node->type;
    new_node->var_name = node->var_name ? strdup(node->var_name) : NULL
        ;
    new_node->left = copy_tree(node->left);
    new_node->right = copy_tree(node->right);
    return new_node;
}

void free_tree(Node* node) {
    if (!node) return;

    free_tree(node->left);
    free_tree(node->right);

    if (node->var_name) free(node->var_name);
    free(node);
}
```

4.5 Conteo de Variables Proposicionales

Esta función identifica cuántas variables únicas (A–Z) aparecen en el árbol lógico. Usa un arreglo booleano como bandera para marcar la presencia de cada letra.

Listing 5: Conteo de variables proposicionales

```c
static void mark_vars(Node* node, bool vars[26]) {
    if (!node) return;

    if (node->type == VAR && node->var_name) {
        char c = node->var_name[0];
        if (c >= 'A' && c <= 'Z')
            vars[c - 'A'] = true;
    }

    mark_vars(node->left, vars);
    mark_vars(node->right, vars);
}

int get_num_vars(Node* node) {
    bool vars[26] = {false};
```

```
        mark_vars(node, vars);

        int count = 0;
        for (int i = 0; i < 26; i++)
            if (vars[i]) count++;

        return count;
    }
```

4.6 Evaluación de Fórmulas Lógicas

Se puede evaluar una fórmula con una asignación booleana específica. Las variables se asocian a valores mediante su posición (A=0, B=1, ..., Z=25).

Listing 6: Evaluación de una fórmula lógica

```
    bool eval_formula(Node* node, bool assignment[26]) {
        if (!node) return false;

        switch (node->type) {
            case VAR:
                return assignment[node->var_name[0] - 'A'];
            case NOT:
                return !eval_formula(node->left, assignment);
            case AND:
                return eval_formula(node->left, assignment) &&
                    eval_formula(node->right, assignment);
            case OR:
                return eval_formula(node->left, assignment) ||
                    eval_formula(node->right, assignment);
            default:
                return false;
        }
    }
```

4.7 Verificación de Satisfacibilidad (SAT)

Se utiliza backtracking para verificar si existe alguna combinación de valores de verdad que satisfaga la fórmula lógica.

Listing 7: Backtracking para resolver SAT

```
    bool solve_rec(Node* node, bool assignment[26], int idx, int max_vars)
        {
        if (idx == max_vars)
            return eval_formula(node, assignment);

        assignment[idx] = false;
        if (solve_rec(node, assignment, idx + 1, max_vars)) return true;

```

```
        assignment[idx] = true;
        if (solve_rec(node, assignment, idx + 1, max_vars)) return true;

        return false;
    }

    bool solve_sat(Node* formula, int num_vars) {
        bool assignment[26] = {false};
        return solve_rec(formula, assignment, 0, num_vars);
    }
```

4.8 Evaluación de Cláusulas bajo una Asignación

Para verificar si una cláusula está satisfecha con una asignación específica de valores booleanos a las variables, se utiliza la función `is_satisfied`. Esta función analiza un nodo que representa una cláusula (puede ser un literal o su negación) y devuelve `true` si la cláusula es verdadera con la asignación dada.

- Si el nodo es una variable, simplemente se consulta su valor en el arreglo de asignación.
- Si el nodo es una negación, se niega el valor asignado a la variable correspondiente.

Listing 8: Función para evaluar si una cláusula está satisfecha

```
    bool is_satisfied(Node* clause, bool* assignment, int num_vars){
        if (!clause) return false;

        if (clause->type == VAR){
            int var_index = atoi(clause->var_name);
            return assignment[var_index];
        }
        else if (clause->type == NOT){
            return !assignment[atoi(clause->left->var_name)];
        }
        return false;
    }
```

4.9 Algoritmo de Backtracking para SAT

El algoritmo de backtracking recorre todas las posibles asignaciones de valores de verdad para las variables, utilizando recursión:

- Cuando se asignan valores a todas las variables, se verifica si todas las cláusulas de la fórmula están satisfechas.
- Si se encuentra una asignación que satisface todas las cláusulas, se devuelve `true`.
- Si ninguna asignación es válida, se devuelve `false`.

La función `solve_sat` es la interfaz principal que inicializa la asignación y llama a la función recursiva `backtrack`.

Listing 9: Backtracking para resolver SAT

```c
bool backtrack(Node* formula, bool* assignment, int num_vars, int index
    ){
    if (index == num_vars){
        Node* current_clause = formula;
        while (current_clause){
            if (!is_satisfied(current_clause, assignment, num_vars)){
                return false;
            }
            current_clause = current_clause->right;
        }
        return true;
    }

    assignment[index] = true;
    if (backtrack(formula, assignment, num_vars, index + 1)) return
        true;

    assignment[index] = false;
    if (backtrack(formula, assignment, num_vars, index + 1)) return
        true;

    return false;
}

bool solve_sat(Node* formula, int num_vars){
    bool* assignment = malloc(num_vars * sizeof(bool));
    if (!assignment) {
        fprintf(stderr, "Error al asignar memoria.\n");
        return false;
    }

    for (int i = 0; i < num_vars; i++){
        assignment[i] = false;
    }

    bool result = backtrack(formula, assignment, num_vars, 0);

    free(assignment);
    return result;
}
```

5 Conclusiones

Hemos presentado la implementación de un algoritmo que determina si las variables de una fórmula booleana pueden ser reemplazadas con **T** o **F** de modo que la fórmula entregue resultado **T**, es decir si es satisfacible o no. La tarea fundamental es codificar una fórmula de lógica proposicional escrita en LaTeX de tal forma que los únicos operadores que se evalúen sean la negación y conjunción lo cual facilita la asignación de valores de verdadero o falso usando un árbol sintáctico.