Departamento de Matemática y Ciencia de la Computación

Algoritmo para determinar si una expresión descrita en lógica proposicional es satisfacible

Benjamin Cáceres
benjamin.caceres.c@usach.cl

Cristóbal Gallardo
cristobal.gallardo.c@usach.cl

Lógica - Cod.lógica
Licenciatura en Ciencia de la Computación

Semestre Otoño 2025

1 Introducción

El presente informe describe el desarrollo de un programa en lenguaje C diseñado para el análisis y evaluación de fórmulas de lógica proposicional escritas en notación LaTeX. Esto permite interpretar dichas fórmulas, construir una representación basada en un árbol de sintaxis abstracta, transformar las expresiones a una forma lógica, y evaluar su satisfacibilidad a través de la exploración exhaustiva de combinaciones de valores de verdad.

El programa utiliza la herramienta Flex(Lex) para el análisis léxico, paréntesis y variables. A partir de estos tokens, se construye un árbol que modela la estructura lógica de la fórmula. Posteriormente, se realiza la traducción para tener como resultado la fórmula en su Forma Normal Conjuntiva (CNF).

Por ultimo , una de las funcionalidades centrales del sistema es la verificación de satisfacibilidad. Para ello, se identifican las variables presentes en la fórmula y se generan todas las posibles combinaciones de valores de verdad, esto se realiza de manera recursiva.

A contimuación, se detalla el procedimiento y el funcionamiento general del programa, seguido por los algoritmos principales implementados. Finalmente, se presenta una conclusión que resume los resultados obtenidos y trabajos futuros.

2 Procedimiento

Para el desarrollo del programa, primero se tiene un archivo de texto como entrada, que contiene una fórmula lógica escrita en notación LaTeX y delimitada por signos de doble dólar ($$). Esta fórmula se lee y procesa por un analizador léxico implementado con la herramienta `Flex`. El lexer recorre el texto identificando símbolos lógicos (como `\wedge`, `\vee`, etc.). Cada elemento reconocido se traduce internamente a un `TOKEN` específico (en este caso `AND`, `OR`, `IMPLIES`, etc.) que son almacenados en un arreglo de tokens.

Una vez la entrada es tokenizada, se inicia el análisis sintáctico mediante funciones recursivas en `C`, las cuales procesan estos tokens para construir un árbol sintáctico abstracto (AST) que representa la estructura lógica de la fórmula. Así, cada nodo del árbol corresponderá a un operador lógico, constante o variable, y puede tener subnodos izquierdo y derecho dependiendo de su tipo.

Luego, con el árbol ya generado, el programa transforma la fórmula a una versión equivalente sin implicaciones (SAT lineal), reemplazando cada implicación por su forma lógica: $\neg(\phi \wedge \neg\psi)$. Posteriormente, este nuevo árbol es recorrido por una función que empuja las negaciones hacia las hojas, aplicando leyes de De Morgan y simplificando dobles negaciones, lo que convierte la fórmula en una

forma normal negativa (NNF). A partir de esta forma, el árbol es transformado a su forma normal conjuntiva (CNF) mediante distribución de disyunciones sobre conjunciones y la simplificación de expresiones que involucren constantes lógicas como $\top$ y $\bot$.

Una vez obtenida la CNF, se identifican y almacenan todas las variables proposicionales utilizadas en la fórmula. A continuación, el programa generará todas las combinaciones posibles de valores de verdad para estas variables (nótese que puede llegar hasta un total de 2^n, siendo n el número de variables distintas) y evalúa cada una de ellas recorriendo el árbol CNF con una función recursiva que interpreta los nodos lógicos bajo cada asignación de valores. Si alguna de estas combinaciones satisface la fórmula (es decir, la evalúa como verdadera), entonces el programa concluye que la fórmula es satisfacible; si ninguna combinación logra lo anterior, se determina que la fórmula no es satisfacible.

Finalmente, el resultado de la evaluación se muestra como una de las siguientes salidas posibles: `SATISFACIBLE`, `NO-SATISFACIBLE`, o `NO-SOLUTION`, dependiendo del resultado del análisis.

A continuación, se presentan los algoritmos principales que conforman este procedimiento.

3 Algoritmo

Algorithm Traducción a SAT

Input: Árbol sintáctico `nodo` que representa una fórmula proposicional lógica.
`nodo` $\in \mathcal{A}$, donde $\mathcal{A}$ incluye nodos de tipo:
{VAR, NEG, AND, OR, IMPLIES, TOP, BOT}.
Output: Árbol equivalente `nodo`$'$ que representa la fórmula traducida a forma válida para *SAT lineal*, utilizando únicamente los conectores VAR, NEG y AND.

```
1  Función traducir_SAT_lineal(nodo)
2      if nodo = nulo then
3          retornar nulo
4      según tipo de nodo hacer
5          caso VAR: retornar copia(nodo)
6          caso NEG:
7              hijo_traducido ← traducir_SAT_lineal(nodo.hijo)
8              retornar NEG(hijo_traducido)
9          caso AND:
10             izq ← traducir_SAT_lineal(nodo.izq)
11             der ← traducir_SAT_lineal(nodo.der)
12             retornar AND(izq, der)
13         caso OR:
14             izq ← traducir_SAT_lineal(nodo.izq)
15             der ← traducir_SAT_lineal(nodo.der)
16             retornar NEG(AND(NEG(izq), NEG(der)))
17         caso IMPLIES:
18             izq ← traducir_SAT_lineal(nodo.izq)
19             der ← traducir_SAT_lineal(nodo.der)
20             retornar NEG(AND(izq, NEG(der)))
21         caso TOP o BOT: retornar copia(nodo)
```

Algorithm Evaluador de Satisfacibilidad

Input: Árbol sintáctico `raiz` que representa una fórmula proposicional lógica.
`raiz` $\in \mathcal{A}$, donde $\mathcal{A}$ es el conjunto de árboles con nodos de tipo:
{VAR, NEG, AND, OR, TOP, BOT}.
Output: Valor booleano: `VERDADERO` si la fórmula es satisfacible,
`FALSO` en caso contrario.

```
1  función es_satisfacible(raiz)
2      vars ← lista de variables únicas extraídas del árbol
3      n ← cantidad de variables
4      total ← 2^n      // número de combinaciones posibles
5      para i ← 0 hasta total −1 hacer
6          asignacion ← arreglo de tamaño n
7          para j ← 0 hasta n − 1 hacer
8              asignacion[j] ← (i ≫ j) & 1
9          si evaluar(raiz, vars, asignacion) = VERDADERO entonces
10             retornar VERDADERO
11     retornar FALSO

12 función evaluar(nodo, vars, asignacion)
13     según tipo de nodo hacer
14         caso VAR:
15             índice ← posición de nodo.nombre en vars
16             retornar asignacion[índice]
17         caso NEG:
18             retornar ¬ evaluar(nodo.izq, vars, asignacion)
19         caso AND:
20             retornar evaluar(nodo.izq, vars, asignacion) ∧ evaluar(nodo.der, vars, asignacion)
21         caso OR:
22             retornar evaluar(nodo.izq, vars, asignacion) ∨ evaluar(nodo.der, vars, asignacion)
23         caso TOP:
24             retornar VERDADERO
25         caso BOT:
26             retornar FALSO
```

4 Implementación

```
Algorithm Traducción a SAT lineal

Input: AST con fórmula a traducir
Output: un AST equivalente con fórmula traducida respetando SAT lineal
(usando sólo negación, conjunción y variables)

struct Nodo* traducir(struct Nodo *nodo) {
    struct Nodo *izq_t = NULL, *der_t = NULL;
    if (!nodo) {
        return NULL;
    }

    switch (nodo->tipo) {
        case VAR:
            return copiar_nodo(nodo);

        case NEG:
            return negacion(traducir(nodo->izq));

        case AND:
            return conjuncion(traducir(nodo->izq), traducir(nodo->der));

        case OR: {
            // T(1  2) = ¬(¬T(1)  ¬T(2))
            izq_t = traducir(nodo->izq);
            der_t = traducir(nodo->der);
            return negacion(conjuncion(negacion(izq_t), negacion(der_t)));
        }

        case IMPLIES: {
            // T(1 → 2) = ¬(T(1)  ¬T(2))
            izq_t = traducir(nodo->izq);
            der_t = traducir(nodo->der);
            return negacion(conjuncion(izq_t, negacion(der_t)));
        }

        default:
            return NULL;
    }
}
```

```
Algorithm Determina Satisfacibilidad


Input: fórmula ya traducida a SAT lineal
Output: 1 si la fórmula es satisfacible, 0 de caso contrario


int eval(struct Nodo *n, char **vars, int *vals, int n_vars) {
    int i, a, b;
    if (!n) {
        return 0;
    }

    switch (n->tipo) {
        case VAR:
            for (i = 0; i < n_vars; i = i + 1) {
                if (son_iguales(vars[i], n->nombre)) {
                    return vals[i];
                }
            }
            return 0;

        case NEG:
            return !eval(n->izq, vars, vals, n_vars);

        case AND:
            return eval(n->izq, vars, vals, n_vars) && eval(n->der, vars, vals, n_vars);

        case OR:
            return eval(n->izq, vars, vals, n_vars) || eval(n->der, vars, vals, n_vars);

        case IMPLIES: {
            a = eval(n->izq, vars, vals, n_vars);
            b = eval(n->der, vars, vals, n_vars);
            return !a || b;
        }

        case TOP:
            return 1;
        case BOT:
            return 0;
    }
    return 0;
```

```
}
```

Algoritmo de evaluación exhaustiva

```
int es_satisfacible(struct Nodo *n) {
    char **vars_tmp, **vars;
    int i, j, capacidad, n_vars, *vals, total, result;

    capacidad = 10;
    n_vars = 0;
    vars_tmp = calloc(capacidad, sizeof(char *));

    recolectar_vars(n, vars_tmp, &n_vars);

    vars = calloc(n_vars, sizeof(char *));
    vals = calloc(n_vars, sizeof(int));

    for (i = 0; i < n_vars; i = i + 1) {
        vars[i] = vars_tmp[i];
    }

    free(vars_tmp);

    total = 1 << n_vars; // 2^n combinaciones

    for (i = 0; i < total; i = i + 1) {
        for (j = 0; j < n_vars; j = j + 1) {
            vals[j] = (i >> j) & 1;
        }
        result = eval(n, vars, vals, n_vars);
        if (result == 1) {
            free(vars);
            free(vals);
            return 1; // satisfacible
        }
    }

    free(vars);
    free(vals);
    return 0; // no satisfacible
}
```

5 Conclusiones

En este informe se logró construir un algoritmo funcional para determinar la satisfacibilidad de fórmulas expresadas en lógica proposicional, respetando las reglas del sistema SAT lineal. La implementación fue realizada mediante una secuencia de funciones que transforman la expresión de entrada hasta llevarla a su forma normal conjuntiva (CNF). Esto nos permite que cada cláusula pueda ser evaluada de forma independiente, ya que basta con que al menos un literal en cada cláusula sea satisfecho por alguna asignación.

El algoritmo de evaluación implementado es de fuerza bruta, ya que explora todas las posibles asignaciones de valores de verdad para cada variable proposicional. Esto implica una complejidad temporal exponencial de orden $O(2^n \cdot m)$, donde n es el número de variables distintas en la fórmula y m es el total de nodos del DAG.

Como trabajo futuro, se propone incorporar un algoritmo de resolución de cláusulas de tipo Horn, lo que permitiría reducir significativamente la complejidad computacional en casos donde la fórmula cumpla con dicha restricción. Esta mejora no solo optimizaría el tiempo de ejecución, sino que también permitiría manejar instancias más grandes y complejas de forma eficiente.