Tercer Problema DAA

Karlos Alejandro Alfonso Rodríguez
Karel Camilo Manresa Leon

June 21, 2023

Problema

La Pregunta Estaba Karlos pasando (tal vez perdiendo) el tiempo cuando viene Karel y le hace una pregunta. Karlos quería responder que "no" a la pregunta, pero Karel le dijo que no era tan fácil, que la respuesta a esa pregunta iba a depender de un conjunto de pequeñas preguntas de "si" o "no" que este tenía. Luego, con las respuestas a esas preguntas de "si" o "no" armó una expresión booleana conformada por negaciones, expresiones "and", expresiones "or" e implicaciones de las pequeñas preguntas. La respuesta de la pregunta grande dependería de la expresión booleana que a la vez dependía de las pequeñas preguntas. Ayude a Karlos a encontrar si existe una distribución de respuestas a las pequeñas preguntas que le permitan responder que "no" a la pregunta grande.

Problema en términos matemáticos

Dada una expresión booleana, se quiere saber para que asignación de las variables la expresión evalúa $False$.

Análisis del problema

Sea P^* el problema **La Pregunta**, dicho problema es de optimización, ya que pide buscar una distribución de valores para las variables de la expresión booleana. Sea P la versión de decisión de P^*, es decir, se quiere saber si existe una alguna distribución de valores para las variables de la expresión tal que dicha expresión evalúe $False$. Demostremos que P es $NP-completo$, luego P^* será $NP-Hard$.

NP-completitud de P

NP

Sea A un algoritmo que puede verificar P. Una de las entradas de A es una codificación estándar de una fórmula lógica F (literales, negaciones, expresiones and, expresiones or, implicaciones). La otra entrada es un certificado correspondiente a una asignación de un valor booleano a cada una de las variables de la fórmula F.
El algoritmo A se construye de la siguiente manera. Se genera una FNC de $\neg F$, luego por cada cláusula de FNC se verifica el valor de sus literales en la asignación. Si una cláusula C contiene al menos un literal que su asignación sea positiva, entonces dicha cláusula será positiva, en caso de que todos los literales de C tengan asignaciones negativos, entonces C será negativa. Si todas las cláusulas de FNC son positivas, entonces FNC será positiva; en caso contrario FNC será negativa. La respuesta de A será la satisfacibilidad de FNC.

Generar la FNC de $\neg F$ se realiza en tiempo polinomial utilizando el algoritmo de Tseitin [1]. Luego el recorrido por cada cláusula y cada literal se realiza en $O(|cnf|)$, donde cnf es la forma normal conjuntiva de la fórmula inicial, por lo tanto A es polinomial. Luego el algoritmo A tiene tiempo polinomial. Luego P es NP.

Reducción

Se seleccionó el problema $NP-completo$ SAT (Boolean satisfiability problem) como problema conocido para aplicarle una reducción polinómica al problema P.
El problema SAT, tiene como entrada una expresión lógica en forma normal conjuntiva (FNC). El problema P, tiene como entrada una expresión lógica cualquiera F, podemos transformar F en una entrada de SAT (en tiempo polinomial) aplicando el algoritmo de Tseitin sobre $\neg F$. Entonces FNC será la entrada de SAT.
Sea S una salida del problema SAT, S será la salida correcta de P, ya que como $FNC \cong \neg F$, entonces $FNC = True \Rightarrow F = False$ y $FNC = False \Rightarrow F = True$. Luego S soluciona el problema P.

Algoritmo de Tseitin

El algoritmo de Tseitin se basa en la introducción de variables adicionales para representar subfórmulas de la fórmula original. La idea es reemplazar cada subfórmula de la fórmula original con una nueva variable, y luego agregar cláusulas que relacionen estas nuevas variables con las variables originales. Este proceso se repite recursivamente hasta que toda la fórmula original se ha reemplazado por variables nuevas.
Por ejemplo, consideremos la fórmula lógica $F = (p \Rightarrow q) \wedge (r \Rightarrow s)$. Primero, se introducen dos variables nuevas, A y B, para representar las subfórmulas $(p \Rightarrow q)$ y $(r \Rightarrow s)$, respectivamente. Luego, se agregan cláusulas que relacionan estas nuevas variables con las variables originales:

- $A \vee \neg p \vee q$
- $B \vee \neg r \vee s$
- $F \equiv A \wedge B$

En estas cláusulas, la primera y sengunda cláusula representan las implicaciones $(p \Rightarrow q)$ y $(r \Rightarrow s)$ como disyunciones de literales, y la tercera cláusula establece que la fórmula F es equivalente a la conjunción de las variables nuevas A y B. Este algoritmo se puede aplicar de manera recursiva a las subfórmulas A y B, si es necesario. El resultado final es una fórmula en FNC que es lógicamente equivalente a la fórmula original.

Soluciones para el problema

Podemos clasificar el problema en dos casos según el número de literales por cláusula. Si cada cláusula tiene a lo sumo dos literales, es una variante de 2-SAT. En este caso, el problema puede resolverse eficientemente en tiempo polinomial.
Por otro lado, si cada cláusula contiene tres o más literales, la complejidad del problema aumenta significativamente y solo se podrá resolver en tiempo exponencial.

Fuerza bruta

La solución más sencilla para el problema es la fuerza bruta. En este caso, se evalúa cada posible asignación de valores a las variables de la expresión y se verifica si la expresión es verdadera o falsa. Si la expresión es verdadera, se devuelve la asignación de valores que la hace verdadera. En caso contrario, se devuelve que la expresión no es satisfacible.

Pseudocódigo

```
function exhaustive_enumeration(exp) do
    cnf = to_CNF(exp)
    n = |vars in exp|
    vars = [n]
    pos = 0
    return generate_assignments(vars, pos, n, |cnf|, cnf)
end

function generate_assignments(vars, pos, n, m, cnf) do
    if pos == n do
        if evaluate(vars, cnf) do
            return vars
        end
    end else do
        vars[pos] = 0
        if generate_assignments(vars, pos + 1, n, m, cnf) do
            return vars
        end

        vars[pos] = 1
        if generate_assignments(vars, pos + 1, n, m, cnf) do
            return vars
        end
    end
    return Null      You, 2 days ago • documentation update
end
```

El algoritmo de fuerza bruta tiene una complejidad exponencial en el peor caso. En particular, si la fórmula tiene n variables y m cláusulas, entonces el número de posibles asignaciones de valores de verdad es 2^n, lo que significa que se necesitan 2^n iteraciones para evaluar todas las posibles asignaciones. En cada iteración, el algoritmo debe evaluar la fórmula lógica para determinar si la asignación actual satisface la fórmula, esto implica recorrer todos los literales de

cada cláusula, para verificar si dicha asignación satisface la fórmula. Luego la complejidad temporal sería $O(2^n * |F|)$.

Algoritmo DPLL

El algoritmo DPLL (Davis-Putnam-Logemann-Loveland) es un algoritmo de resolución de satisfacibilidad booleana (SAT) que se utiliza para determinar si una fórmula lógica proposicional es satisfacible.Este algoritmo trabaja de manera recursiva, y se basa en la siguiente idea: si una fórmula lógica es satisfacible, entonces debe ser posible encontrar una asignación de valores de verdad a las variables proposicionales que satisfaga la fórmula. Por lo tanto, el algoritmo DPLL busca una asignación de valores de verdad que satisfaga la fórmula, y utiliza técnicas de poda para reducir el espacio de búsqueda.
El algoritmo DPLL comienza con una fórmula lógica en forma CNF, y utiliza dos tipos de reglas para simplificar la fórmula:

- Regla de unidad: Si una cláusula contiene solo un literal, entonces ese literal debe ser verdadero para satisfacer la cláusula. Por lo tanto, se puede asignar un valor de verdad a esa variable proposicional y eliminar todas las cláusulas que contienen ese literal.

- Regla de purga: Si una variable proposicional aparece solo en una polaridad en todas las cláusulas restantes, entonces se puede asignar un valor de verdad a esa variable de manera que satisfaga todas las cláusulas que contienen esa polaridad. Luego, se pueden eliminar todas las cláusulas que contienen esa variable.

Después de aplicar estas reglas, el algoritmo DPLL verifica si la fórmula se ha simplificado lo suficiente como para que sea trivialmente satisfacible o insatisfacible.
Si ninguna de las reglas anteriores se puede aplicar, entonces el algoritmo DPLL elige una variable proposicional arbitraria y prueba dos casos diferentes: asignarle un valor de verdad verdadero y un valor de verdad falso. Luego, el algoritmo DPLL aplica recursivamente las reglas de simplificación a la fórmula resultante en cada caso. Si alguna de las ramas recursivas devuelve una solución satisfactoria, entonces el algoritmo DPLL devuelve esa solución. Si ambas devuelven contradicciones, entonces es insatisfacible.

Pseudocódigo

```
function dpll(exp) do
    cnf = to_CNF(exp)
    n = |vars in exp|
    vars = [n]
    assigns = [n]
    return dpll_rec(vars, assigns, n, |cnf|, cnf)
end

function dpll_rec(vars, assigns, n, |cnf|, cnf) do
    if evaluate(vars, cnf) do
        return vars
    end

    literal = select_next_literal(vars, assigns)
    assigns[literal-1] = true
    values = select_literal_values(random)

    foreach (value in values) do
        vars[literal-1] = value
        new_formula, valid_solution = unit_propagation(literal, value, vars, assigns, cnf)

        if valid_solution do
            result, valid_solution = dpll_rec(vars, assigns, n, |cnf|, new_formula)
        end
    end       You, 2 days ago • documentation update …
end
```

Métodos utilizados:

- **select_next_literal:** Selecciona el proximo literal al que se le va a asignar un valor. En este caso se selecciono dicho literal de manera aleatoria. No obstante existen heurísticas más complejas para esta selección que aprovechan la cantidad de variables y particularidades de la fórmula, obteniendo resultados con mayor velocidad.
- **select_literal_values:** Selecciona el valor que tomará el literal, al igual que el método **select_next_literal** lo hace de manera aleatoria.
- **unit_propagation:** Dado el valor y la varible seleccionados, evalúa dicha asignación y la propaga por la fórmula, verificando asignaciones triviales.

La complejidad temporal del algoritmo DPLL depende del tamaño de la fórmula booleana, es decir, del número de variables y de cláusulas en la fórmula. En el peor caso, el número total de posibles asignaciones de valores de verdad a las variables es 2^n ($n = cantidad\_variables$). Además las operaciones de simplificación (reglas de purga y unidad) se aplican por cada cláusula por tanto esto influye también en la complejidad temporal del algoritmo, siendo esta $O(2^n * m)$.
Obsérvese que DPLL tiene la misma complejidad temporal que el algoritmo por fuerza bruta; no obstante, en la práctica DPLL es significativamente más eficiente que la fuerza bruta debido a las técnicas de poda que utiliza (reglas de purga y unidad).

Algoritmo genético

En el contexto del problema SAT, los algoritmos genéticos pueden ser útiles para encontrar una solución satisfactoria al problema. A diferencia de los algoritmos de backtracking y los algoritmos basados en DPLL, los algoritmos genéticos no garantizan encontrar una solución óptima al problema SAT. Sin embargo,

pueden ser útiles en casos donde la fórmula FNC es muy grande o compleja y se requiere de una solución satisfactoria en un tiempo razonable.
El algoritmo genético implementado recibe una fórmula en FNC como entrada y sigue los siguientes pasos:

1. Representación de cromosomas: Cada posible asignación de valores de verdad a las variables proposicionales en la fórmula CNF se representa como un cromosoma. En una representación binaria, cada cromosoma puede ser una cadena de bits, donde cada bit representa el valor de verdad de una variable proposicional.

2. Evaluación de la aptitud: La calidad de cada cromosoma se evalúa utilizando una función de fitness que mide la satisfacibilidad de la fórmula CNF para esa asignación de valores de verdad. Esta función devuelve un valor de aptitud que indica el número de cláusulas que se satisfacen para esa asignación de valores de verdad.

3. Creación de la población inicial: Se crea una población inicial de cromosomas aleatorios, donde cada cromosoma representa una posible asignación de valores de verdad a las variables proposicionales.

4. Selección: Se seleccionan los cromosomas más aptos (mayor fitness) de la población actual para la reproducción. Se seleccionan los $\frac{n}{2}$ mejores cromosomas como parte de la nueva población y para generar $\frac{n}{2}$ nuevos cromosomas.

5. Operador de cruce: Se combinan dos cromosomas para producir un nuevo cromosoma que comparta características de ambos progenitores.

6. Operador de mutación: Se aplica el operador de mutación (con una probabilidad definida al inicio del algoritmo) a los nuevos cromosomas generados para introducir variabilidad en la población. El operador de mutación cambia aleatoriamente uno o más bits en el cromosoma.

7. Evaluación de la aptitud: Se evalúa la aptitud de los nuevos cromosomas utilizando la función de fitness.

8. Sustitución: Se sustituyen los cromosomas menos aptos de la población actual con los nuevos cromosomas generados.

9. Comprobación de la condición de terminación: Se comprueba si se ha encontrado una solución satisfactoria o si se ha alcanzado un número de generaciones. Si se ha encontrado una solución satisfactoria, se devuelve el mejor cromosoma como solución.

10. Repetición del proceso: Si no se ha encontrado una solución satisfactoria, se repiten los pasos 4 a 9 hasta que se cumpla la condición de terminación.

Pseudocódigo

```
function genetic(generations, population_len, mutation_rate, cnf) do
    population = create_population(population_len)

    for (i = 0, i < generations, i++) do
        selection = selection(population, cnf)

        if fitness(selection[0], cnf) == |cnf| do
            return selection[0]
        end

        population = breed(selection)
    end

    population = selection(population, cnf)
    return population[0]      You, yesterday • genetic pseudo …
end
```

Para mayor comodidad, digámosle n al tamaño de la población, v a la cantidad de variables de la fórmula, g a la cantidad de generaciones del algoritmo y l a la cantidad de cromosomas restantes luego de seleccionar los mejores.
En la complejidad temporal de este algoritmo genético influyen directamente varios procesos:

1. Generar población inicial: se crean n cromosomas de tamaño v de manera aleatoria, por lo que la complejidad temporal sería $O(nv)$.

2. Simular generaciones: en la simulación de las generaciones se tienen en cuenta los siguientes procesos:

   - Seleccionar mejores cromosomas: para seleccionar los mejores cromosomas, se aplica la función fitness sobre cada uno. La función fitness tiene costo $O(|cnf|)$, ya que recorre la fórmula para evaluar el cromosoma en ella. Esto se realiza por cada cromosoma, por lo que el costo de seleccionar sería $O(n|cnf|)$.
   - Cruzar cromosomas seleccionados: para cruzar cromosomas se recorre la selección anterior y se mezclan los cromosomas. El proceso de mezcla tiene costo $O(v)$, por lo que este proceso tiene complejidad $O(lv) \leq O(nv)$.
   - Ciclo: los procesos anteriores se ejecutan g veces. Por tanto este ciclo tiene costo $O(gn(|cnf| + v))$. Como $v \leq |cnf|$, entonces la complejidad sería $O(gn|cnf|)$.

3. Selección final: para la selección del cromosoma final se realiza nuevamente el proceso de selección, con costo $O(n|cnf|)$.

Tras el análisis anterior se puede determinar que el costo total del algoritmo genético es $O(gn|cnf|)$.

Algoritmo 2-Sat

Uno de los casos particulares del problema SAT, es el problema 2-SAT (solamente dos variables por cada cláusula). Se decidió diferenciar el caso 2-SAT del

resto ya que se puede resolver en tiempo polinomial, lo que para las instancias 2-SAT del problema sería una mejora sustancial.
Para solucionar el problema 2-SAT se crea un grafo dirigido donde las variables y sus negaciones están representadas por nodos. Supongamos que tenemos una cláusula $(a \vee b)$, es equivalente a $((\neg a \Rightarrow b) \wedge \neg b \Rightarrow a)$; luego en el grafo el nodo $\neg a$ tendrá una arista hacia b y $\neg b$ tendrá una arista hacia a. De esta forma se representa la fórmula en términos de teoría de grafos, finalmente la cantidad de nodos será el doble de la cantidad de variables y la cantidad de arcos el doble de la cantidad de clausulas.
Luego se hallan las componentes fuertemente conexas (SCC_s) del grafo. Esto tiene como ventaja:

- Si una variable booleana a y su negación a, pertenecen a una misma SCC, entonces no existe una asignación booleana que satisfaga todas las cláusula. Esto se debe a que cualquier asignación booleana que se haga a a también se hará a $\neg a$, lo cual es una contradicción.

- Si todas las variables a y sus negaciones $\neg a$ pertenecen a diferentes SCC_s, entonces se puede construir una asignación booleana que satisfaga todas las cláusulas a partir de las SCC_s. En particular cada SCC, se puede asignar el valor verdadero a todas sus variables booleanas y el valor falso a todas sus negaciones, y sabemos que esto es posible ya que las variables y sus negaciones estan en SCC_s diferentes.

Una vez se comprueba que se puede establecer una asignación de valores de verdad que satisfaga la fórmula, entonces se procede construir dicha asignación. Para ello se realiza un recorrido en orden topológico a cada SCC y se asigna valor verdadero a las variables que no tengan valor asignado en el momento de la visita, y valor falso a su opuesto directamente. Cuando se visite la primera SCC se le asiganará valor verdadero a todas sus variables. De esta forma se satisfacen todas la implicaciones del grafo y a su vez las cláusulas de la fórmula inicial.

Pseudocódigo

```
1   function two_sat(cnf) do
2       g = build_graph(cnf)
3       scc = get_sccs(g)
4
5       if have_contrad(scc) do
6           return Null
7       end else do
8           return set_assignments(scc)
9       end
10  end
```

Para analizar la complejidad temporal del algoritmo se tienen en cuenta los siguientes aspectos:

1. Construcción del grafo: para construir el grafo es necesario recorrer la fórmula para hallar las variables (para crear los nodos) y crear las aristas. Esto tiene costo $O(|cnf|)$. Como por cada cláusulas existen solamente dos literales, entonces podemos decir que $|cnf| = 2m$, donde m es la cantidad de cláusulas. Luego construir el grafo tiene costo $O(m)$. El grafo tendrá $2n$ nodos (n es la cantidad de variables) y $2m$ aristas (m es la cantidad de cláusulas).

2. Hallar las componentes fuertemente conexas tiene costo $O(2n + 2m) = O(n + m)$.

3. Comprobar si existe una contradicción en las SCC_s tiene costo $O(n)$ ya que para ello se realiza un recorrido por los nodos y se puede comprobar en $O(1)$ si el opuesto de dicho nodo pertenece a la misma componente conexa. Esta comprobación puede realizarse teniendo guardado en una estructura de datos donde indexar tenga costo $O(1)$ (diccionario o array en caso que las variables se traten con enteros) la SCC a donde pertenezca cada nodo.

4. Para construir las asignaciones es necesario recorrer los nodos de cada SCC y se puede guardar en un array de tamaño $2n$ el valor asigando a cada nodo. Si se tratan como enteros las variables esta indexación puede hacerse en $O(1)$, por lo que el costo será $O(n)$.

Apoyándose en el análisis anterior se puede determinar que el algoritmo tiene complejidad temporal $O(n + m)$.

K-aproximación

La idea de este algoritmo es hacer una especie de algoritmo greedy con daños colaterales. Por cada iteración del algoritmo se determina cual es el literal más frecuente en la fórmula que aun no tenga valor asignado. A ese literal se le asigna valor de verdad 1 y a su opuesto 0, esto convierte todas las cláusulas donde se encuentre en verdaderas. Esta decisión tiene como efecto negativo que cláusulas que contengan el opuesto de un literal con valor 1, potencialmente pueden resultar negativas. Pero como el literal escogido es el más frecuente, tendrá una ocurrencia mayor o igual a su opuesto, por tanto con esta elección de valor para esa variable se obtiene una cantidad de cláusulas verdaderas mayor o igual a la cantidad de falsas. Esto implica que en el peor de los casos en cada iteración, la cantidad de cláusulas verdaderas sea la mitad del total de cláusulas evaluadas. Por tanto al finalizar el algoritmo al menos la mitad de las cláusulas son verdaderas.

Proof. El algoritmo termina ya que por cada iteración se elimina al menos una cláusula de la fórmula original, y dicha fórmula es finita.

Correctitud:
Sea l_i el literal más frecuente en una iteración cualquiera del algoritmo y $-l_i$ su opuesto. Sea cl_i la cantidad de cláusulas que se evaluaron como verdaderas en esta iteración y cl_i^{-1} las que se evaluaron en falso.
Supongamos que $cl_i^{-1} > cl_i$, entonces la cantidad de cláusulas donde no aparece l_i y si $-l_i$ es mayor a la cantidad de cláusulas donde aparece l_i, lo cual es una contradicción ya que l_i era el literal más frecuente. Por tanto $cl_i^{-1} \leq cl_i$.
Como

$$cl_1^{-1} \leq cl_1, cl_1^{-1} \leq cl_1, \ldots, cl_m^{-1} \leq cl_m$$

Entonces

$$cl_1^{-1} + cl_2^{-1} + \cdots + cl_m^{-1} \leq cl_1 + cl_2 + \cdots + cl_m$$

Por tanto la cantidad total de cláusulas verdaderas es como mínimo la mitad del total de cláusulas. □

Este algoritmo resulta ser una 2-Aproximación para el problema.

Pseudocódigo

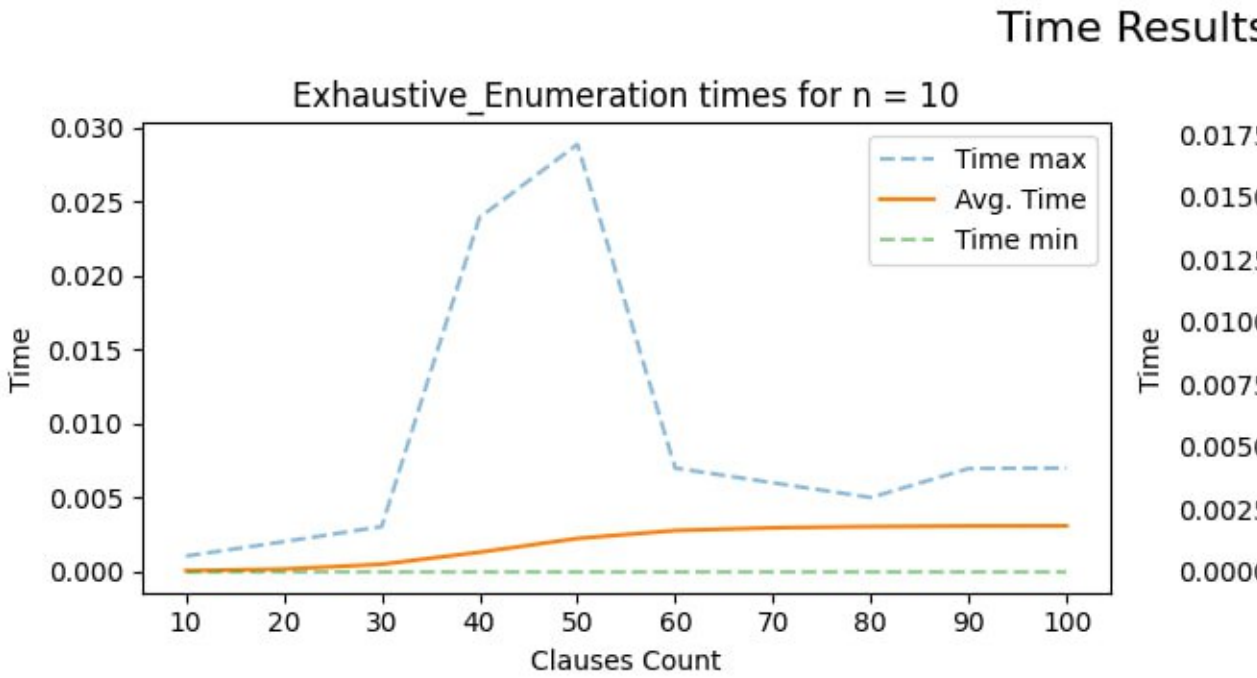
```
1  function two_aprox(cnf) do
2      aux_cnf = cnf
3
4      while aux_cnf != Empty do
5          lit = get_most_frecuent_lit(aux_cnf)
6          var = abs(lit)
7          value = 1 if lit >= 0 else value = 0
8          evaluate_and_remove(aux_cnf, var, value)
9      end
10 end
```

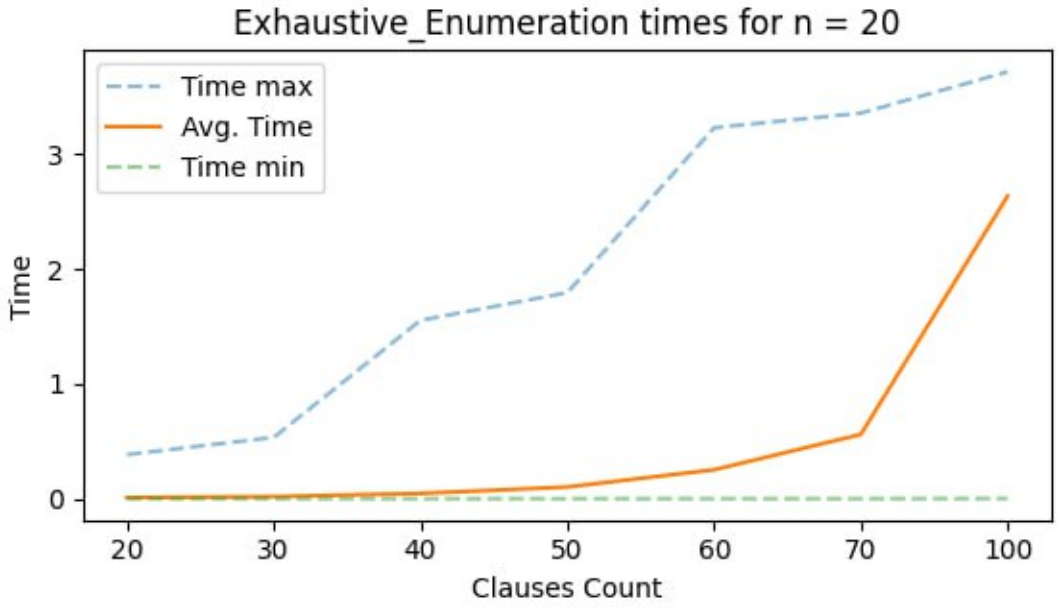
La complejidad temporal de este algoritmo está determina por la cantidad de veces que se ejecuta el ciclo y el costo de las operaciones realizadas detro.
En cada iteración del ciclo al menos una cláusula es eliminada de la fórmula, por lo que a lo sumo se ejecutará m veces, donde m es la cantidad de cláusulas de la fórmula.
Luego seleccionar el literal más frecuente tiene costo $O(|cnf|)$. Para ello puede realizarse un recorrido por cnf contando las repeticiones de cada literal y guardando el valor en un array. Las variables se toman como enteros para indexar en $O(1)$ y sus negaciones ocuparán la posición $variable+len(variables)$, de modo que el array tendrá tamaño $2*len(variables)$. Luego se recorre dicho array para seleccionar el literal más frecuente. Esto tiene costo $O(2 * len(variables))$, como $len(variables) \leq |cnf|$, entonces tiene costo $O(|cnf|)$.
Resta otorgar valor 1 al literal (0 a su opuesto) y eliminar las cláusulas en las que aparece. En un recorrido por la fórmula se puede detectar las cláusulas que contienen al literal y se puede otorgar valor 0 a su opuesto, esto tiene complejidad $O(|cnf|)$. Luego teniendo las cláusulas a eliminar, se puede construir

una nueva fórmula que no contenga dichas cláusulas, con complejidad $O(m)$. Entonces todo el proceso anterior tendrá costp $O(|cnf| + m)$, como $m \leq |cnf|$, el costo sería $O(|cnf|)$.
Finalmente el algoritmo tiene complejidad temporal $O(m * |cnf|)$.

Experimentación

Se generaron [PONER CUANTOS CASOS SE GENERARON] casos de prueba para analizar los resultados arrojasdos por cada algoritmo, para así comprobar de manera práctica su efectividad. Los resultados fueron almacenados en archivos *.json* anexados junto al reporte.
Una comparativa interesante resulta de comparar el algoritmo por fuerza bruta con el algoritmo DPLL. A pesar de tener ambos la misma complejidad temporal, en la práctica DPLL tiene a tener mucha más eficiencia que la fuerza bruta.





[EXPLICAR LAS GRÁFICOS]

References

[1] G. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967-1970*, pp. 466–483, 1970.