Tercer Problema DAA

Karlos Alejandro Alfonso Rodríguez
Karel Camilo Manresa Leon

June 18, 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 $Tseitin$. Luego el recorrido por cada cláusula y cada literal se realiza en $O(cant\_clausulas)$, 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 $Tseitin$ (explicado a continuación) 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

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

```
1  def exhaustive_enumeration(expresion):
2      cnf = to_CNF(expresion)
3      n = len(expresion.vars)
4      vars = [0] * n
5      pos = 0
```

```
6       result = generate_assignments(vars, pos, n, len(cnf),
            cnf)
7       return result
8
9   def generate_assignments(vars, pos, n, m, cnf):
10      if pos == n:
11          if evaluate(vars, cnf):
12              return vars
13          else:
14              return None
15      else:
16          vars[pos] = 0
17          if generate_assignments(vars, pos + 1, n, m, cnf)
                :
18              return vars
19
20          vars[pos] = 1
21          if generate_assignments(vars, pos + 1, n, m, cnf)
                :
22              return vars
23
24          return None
```

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

```
1  def dpll(expresion):
2      cnf = to_CNF(expresion)
3      n = len(expresion.vars)
4      vars = [0] * n
5      assigns = [False] * n
6      return dpll_rec(vars, assigns, n, len(cnf), cnf)
7
8  def dpll_rec(vars, assigns, n, len(cnf), cnf):
9      if evaluate(vars, cnf):
10         return vars
11
12     literal = select_next_literal(vars, assigns)
13     assigns[literal-1] = True
14     values = select_literal_values(random)
15
16     for value in values:
17         vars[literal-1] = value
18         new_formula, valid_solution = unit_propagation(
               literal, value, vars, assigns, cnf)
19
20         if valid_solution:
21             result, valid_solution = dpll_rec(vars,
                   assigns, n, len(cnf), new_formula)
22
```

```
23                 if valid_solution:
24                     return result
25
26         return None
```

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.