Técnicas Heurísticas y Metaheurísticas

Búsqueda local genética para 3−SAT

Ángel Ríos San Nicolás

15 de junio de 2021

Introducción

En 1971 Stephen Coook demostró en su articulo *The Complexity of Theorem Proving Procedures* que el problema **SAT** de satisfactibilidad booleana pertenece a la clase complejidad **NP** − completo, es decir, que cualquier problema de la clase **NP**, de los problemas de decisión verificables en tiempo polinomial, se puede reducir en tiempo polinomial a **SAT**. Éste fue el primer problema demostrado con esta propiedad y supuso un avance en la comprensión de la complejidad computacional porque permitió clasificar fácilmente otros problemas de decisión en la clase **NP**. S. Cook también demostró que se sigue teniendo **NP**−completitud en el problema de satisfactibilidad booleana cuando se restringen las fórmulas a conjunciones de disyunciones de tres variables o sus negaciones, lo que se conoce como el problema 3-**SAT**, que es en el que nos centraremos en este trabajo.

Como no conocemos la respuesta a la conjetura ¿**P** = **NP**?, no sabemos si existen algoritmos que decidan los problemas **NP**, y en particular 3-**SAT**, en tiempo polinomial. Para resolver estos problemas en tiempo razonable desarrollamos técnicas heurísticas y metaheurísticas que pretenden resolver eficazmente o con menos recursos una gran parte de sus instancias. Vamos a describir una estrategia genética para resolver 3-**SAT** que utiliza una búsqueda local para maximizar el número disyunciones que se satisfacen en la fórmula en cada generación del algoritmo.

Durante todo el trabajo nos basamos principalmente en [2]. En la Sección 1 recordamos el lenguaje del cálculo proposicional y enunciamos el problema 3-**SAT**. En la Sección 2 describimos teóricamente el algoritmo de búsqueda local genética que utilizaremos. En la Sección 3 detallamos la implementación del algoritmo en Python. En la Sección 4 explicamos los resultados obtenidos y concluimos el trabajo.

1 El problema de satisfactibilidad booleana

La sintaxis del **cálculo proposicional** consiste en el alfabeto de

- Variables: $\{X_1, X_2, \ldots\}$.
- Conectores lógicos: $\{\neg, \vee\}$.
- Paréntesis: $\{(,)\}$.

El lenguaje de las **fórmulas bien formadas** es el generado por la siguiente gramática.

1. Las variables son fórmulas bien formadas.

2. Si A es una fórmula bien formada, entonces $\neg(A)$ es una fórmula bien formada.

3. Si A y B son fórmulas bien formadas, entonces $(A) \vee (B)$ son fórmulas bien formadas.

A las fórmulas bien formadas del cálculo proposicional las denominaremos **fórmulas booleanas**. Denotaremos por $\Phi(X_1, \ldots, X_n)$ a una fórmula booleana en la que puedan aparecer solo las variables $X_1, \ldots, X_n$. Como abuso de notación, omitiremos los paréntesis de la manera habitual y definimos $A \wedge B := \neg A \vee \neg B$, $A \to B := \neg A \vee B$ y $A \leftrightarrow B := (A \to B) \wedge (B \to A)$.

Una **interpretación** de una fórmula booleana $\Phi(X_1, \ldots, X_n)$ es una tupla $(\epsilon_1, \ldots, \epsilon_n) \in \{0,1\}^n$. Definimos de manera recursiva la **evaluación** de una fórmula booleana en una interpretación, que lo denotamos por $\Phi(\epsilon_1, \ldots, \epsilon_n)$.

1. Si $\Phi(X_i) = X_i$ con $i \in \mathbb{N}$, entonces $\Phi(0) = 0$ y $\Phi(1) = 1$.

2. Si $\Phi(\epsilon_1, \ldots, \epsilon_n) = 0$, entonces $(\neg\Phi)(\epsilon_1, \ldots, \epsilon_n) = 1$.
   Si $\Phi(\epsilon_1, \ldots, \epsilon_n) = 1$, entonces $(\neg\Phi)(\epsilon_1, \ldots, \epsilon_n) = 0$.

3. Si $\Phi(\epsilon_1, \ldots, \epsilon_n) = 1$ o $\Psi(\epsilon_1, \ldots, \epsilon_n) = 1$, entonces $(\Phi \vee \Psi)(\epsilon_1, \ldots, \epsilon_n) = 1$, y en otro caso, si $\Phi(\epsilon_1, \ldots, \epsilon_n) = 0$ y $\Psi(\epsilon_1, \ldots, \epsilon_n) = 0$, entonces $(\Phi \vee \Psi)(\epsilon_1, \ldots, \epsilon_n) = 0$.

Una fórmula booleana $\Phi(X_1, \ldots, X_n)$ es **satisfecha** por una interpretación $(\epsilon_1, \ldots, \epsilon_n)$ si la evaluación de la fórmula en la interpretación es $\Phi(\epsilon_1, \ldots, \epsilon_n) = 1$ y es **satisfactible** si es satisfecha por alguna interpretación.

Problema SAT. Dada una fórmula booleana, decidir si es satisfactible.

Para poder enunciar el problema 3-**SAT**, debemos introducir los conceptos de cláusula y forma normal conjuntiva.

Una **cláusula** es una fórmula booleana de la forma $\Phi(X_1, \ldots, X_n) = X_{i_1} \vee X_{i_2} \vee \cdots \vee X_{i_r}$. Se dice que una cláusula con n variables es una $k-$**cláusula** si contiene exactamente k de sus variables. Una fórmula está en **forma normal conjuntiva** si es de la forma

$$\Phi(X_1, \ldots, X_n) = C_1(X_1, \ldots, X_n) \wedge \cdots \wedge C_m(X_1, \ldots, X_n),$$

donde $C_i(X_1, \ldots, X_n)$ es una cláusula para cada $i = 1, \ldots, m$.

Problema $3-$**SAT.** Dada una fórmula booleana en forma normal conjuntiva de $3-$cláusulas, decidir si es satisfactible.

2 Algoritmo genético con búsqueda local

Describimos de forma teórica y en pseudocódigo el algoritmo que implementaremos.

2.1 Componentes del algoritmo genético

Entrada. El algoritmo toma como entrada una fórmula bien formada del cálculo proposicional en forma normal conjuntiva de $3-$cláusulas

$$\Phi(X_1, \ldots, X_n) = C_1(X_1, \ldots, C_n) \wedge \cdots \wedge C_m(X_1, \ldots, X_n).$$

Cromosomas. Los **cromosomas** son las posibles interpretaciones $(\epsilon_1, \ldots, \epsilon_n) \in \{0,1\}^n$ y definen el espacio de búsqueda. Denominamos **gen** a cada una de las componentes e_i de un cromosoma y **alelos** a sus posibles valores en $\{0,1\}$. El objetivo es encontrar una interpretación que satisfaga la fórmula.

Función fitness. Definimos la función $f : \{0,1\}^n \longrightarrow \{0, \ldots, m\}$ que a cada cromosoma le asocia el número de clausulas de la fórmula satisfechas por la interpretación parcial dada por el cromosoma.

Método Fitness(ϵ)

Entrada: Un cromosoma ϵ.
Salida: El fitness de ϵ, el número $f(\epsilon)$ de cláusulas satisfechas por la interpretación definida por el cromosoma.
Pasos:

- **si** $\epsilon = ()$ **entonces**
  - **devolver** 0
- fitness = 0
- **para cada** C *cláusula de* $\Phi(X_1, \ldots, X_n)$ **hacer**
  - **si** $C(\epsilon) = 1$ **entonces**
    - fitness = fitness +1
- **devolver** fitness

Población inicial. Tomamos una muestra aleatoria de k interpretaciones $\{\epsilon_1, \ldots, \epsilon_k\} \subseteq \{0,1\}^n$ como **población inicial** de la que se obtendrán las sucesivas generaciones.

Selección. El proceso de **selección** de los padres se hace mediante ruleta proporcional al fitness. Dada una población P y un cromosoma $\epsilon_0 \in P$, la probabilidad de escogerlo como padre es el cociente

$$\frac{f(\epsilon_0)}{\sum_{\epsilon \in P} f(\epsilon)}.$$

Además, a partir de la primera generación incluimos de manera elitista los dos cromosomas de mayor fitness de la generación anterior.

Método Selección(P)

Entrada: Una población P de cromosomas.
Salida: Dos cromosomas $(\epsilon_1^0, \ldots, \epsilon_n^0), (\epsilon_1^1, \ldots, \epsilon_n^1) \in P$ escogidos con probabilidad proporcional al fitness.
Pasos:

- fitnessP = [Fitness(ϵ) **para cada** ϵ *en* P]
- fitness_total = $\sum_{\epsilon \in P}$ Fitness(ϵ) = suma (fitnessP)
- prob = [Fitness(ϵ)/ fitness_total **para cada** ϵ *en* P]
- **devolver** *Muestra aleatoria de* P *según la distribución de probabilidad determinada por* prob.

Método Elitismo(P)

Entrada: Una población $P = (\epsilon_1, \ldots, \epsilon_k)$ de cromosomas
Salida: Dos cromosomas con mayor fitness en P.
Pasos:

- $P' = P$
- Ordenamos P' en orden ascendente según el fitness.
- **devolver** *El último y el penúltimo elemento de* P.

Operadores genéticos. Dados dos padres, generamos un único **hijo** mediante cruce y mutación.

Cruce. La operación de **cruce** es uniforme mediante una máscara de cruce. Dados dos padres $(\epsilon_1^0, \ldots, \epsilon_n^0), (\epsilon_1^1, \ldots, \epsilon_n^1)$, tomamos de manera aleatoria con distribución uniforme un elemento $(m_1, \ldots, m_n) \in \{0,1\}^n$ que denominaremos **máscara de cruce**. El hijo $(x_1, \ldots, x_n) \in \{0,1\}^n$ está dado por

$$x_i = \begin{cases} \epsilon_i^0 & \text{si } m_i = 0, \\ \epsilon_i^1 & \text{si } m_i = 1, \end{cases}$$

es decir, el alelo de un gen es del primer padre o del segundo si la correspondiente componente en la máscara de cruce es 0 o 1 respectivamente.

Método CruceUniforme(ϵ^0, ϵ^1)

Entrada: Dos cromosomas padre $\epsilon^0 = (\epsilon_1^0, \ldots, \epsilon_n^0), \epsilon^1 = (\epsilon_1^1, \ldots, \epsilon_n^1)$ seleccionados de una población.
Salida: Un cromosoma hijo $(x_1, \ldots, x_n)$ fruto del cruce de los padres.
Pasos:

- mascara = [sample(0, 1) **para cada** ϵ *en* P] `// sample(0,1) es 0 o 1 con probabilidad 1/2`
- hijo = $(0 \ldots, 0)$ `// (k) ceros`
- **para** $i = 1, \ldots, k$ **hacer**
  - **si** mascara[i] == *1* **entonces**
    - hijo[i] = e_i^0
  - **en otro caso**
    - hijo[i] = e_i^1
- **devolver** hijo

Mutación. A cada hijo $(x_1, \ldots, x_n) \in \{0,1\}^n$ fruto de un cruce le aplicamos, con probabilidad p_{m_1}, la **mutación** correspondiente a recorrer cada uno de sus genes e invertir su alelo con probabilidad p_{m_2}.

Método Mutación(x, p_{m_1}, p_{m_2})

Entrada: Un cromosoma $x = (x_1, \ldots, x_n)$ y las probabilidades de mutación p_{m_1} y p_{m_2}.
Salida: Un cromosoma $x' = (x'_1, \ldots, x'_n)$ mutado.
Pasos:

- **si** random()$\leq 1 - p_{m_1}$ `// random() es un número aleatorio uniforme en (0,1)` **entonces**
  - **para** $i = 1, \ldots, k$ **hacer**
    - **si** random()$\leq 1 - p_{m_2}$ **entonces**
      - $x[i] = x[i] + 1 \mod 2$
- **devolver** x

2.2 Búsqueda local

Observamos que la selección de los padres y el elitismo tienen en cuenta la función fitness, pero no así los cruces y las mutaciones. Para mejorar el funcionamiento del algoritmo, podemos implementar una búsqueda local heurística que transforme los hijos generados en soluciones parcialmente óptimas para inversiones de un gen, es decir, que no exista otra interpretación que difiera en una única componente y satisfaga más cláusulas.

Primero necesitamos un método que calcule la ganancia de un intercambio definida como la diferencia del número de cláusulas satisfechas después del cambio y el número de cláusulas satisfechas antes.

Método Ganancia(ϵ, i)

Entrada: Un cromosoma ϵ y un índice $i \in \{1, \ldots, n\}$.
Salida: El cromosoma δ resultado de invertir la componente i-ésima y la ganancia del cambio, es decir, $f(\delta) - f(\epsilon)$.
Pasos:

```
clausulas = []
para cada C cláusula de Φ(X_1, ..., X_n) hacer
    si X_i aparece en C entonces
        Añadir C a clausulas.
δ = ϵ
δ[i] = δ[i] + 1  mod 2
fit_0 = 0
fit_1 = 0
para cada C en clausula hacer
    si C(ϵ) = 1 entonces
        fit_0 = fit_0 + 1
    si C(δ) = 1 entonces
        fit_1 = fit_1 + 1
devolver (fit_1 − fit_0, δ)
```

El algoritmo de búsqueda local toma como entrada un cromosoma, es decir, una interpretación de la fórmula y hace lo siguiente: Recorre los índices de las variables en orden aleatorio y calcula la ganancia del cambio de la variable correspondiente. Si la ganancia es positiva, entonces se acepta el cambio. Este proceso se repite mientras mejoremos, es decir, hasta que todas las ganancias en una ejecución del bucle sean negativas. La razón de seguir un orden aleatorio en las variables es que, de lo contrario, la búsqueda local se especializaría en satisfacer más cláusulas solo con las primeras variables, lo que es un sesgo inadecuado.

Método FlipHeuristic(ϵ)

Entrada: Un cromosoma ϵ.
Salida: Un cromosoma ϵ' que maximiza el fitness para inversiones de una única variable.
Pasos:

```
index_var = permutation(1, ..., n)
mejora = 1
mientras mejora > 0 hacer
    mejora = 0
    para i = 1, ..., n − 1 hacer
        (ϵ', ganancia) = Ganancia(ϵ, index_var[i])
        si ganancia ≥ 0 entonces
            ϵ = ϵ'
            mejora = mejora + ganancia
devolver ϵ
```

2.3 Algoritmo genético con búsqueda local

Describimos a continuación el pseudocódigo del algoritmo genético que implementa la búsqueda local.

Método GeneticAlgorithm($k, t_{\max}, p_{m_1}, p_{m_2}$, busq)

Entrada: El número k de individuos de la población, el número máximo de generaciones $t_{\max}$, las probabilidades de mutación p_{m_1} y p_{m_2} y un valor booleano busq que indica si se aplica o no la búsqueda local.

Salida: Si el algoritmo encuentra una interpretación ϵ_S que satisface Φ, entonces se devuelve **True**, ϵ_S y el número de generaciones necesarias; en caso contrario, se devuelve **False** y un individuo con el mayor fitness de la población.

Pasos:

```
P = [sample(0,1,n) para cada i = 1, ..., k]
si busq = True entonces
    P = [FlipHeuristic(ϵ)  para cada  ϵ ∈ P]
t = 0
mientras t < t_max  y ∀ϵ ∈ P, Φ(ϵ) = 0 hacer
    t = t + 1
    P_0 = P
    P = []
    P = P ∪ Elitismo(P_0)
    mientras |P| < |P_0| hacer
        padre_1, padre_2 = Selección(P)
        hijo = CruceUniforme(padre_1, padre_2)
        hijo = Mutación(hijo, p_m1, p_m2)
        si busq = True entonces
            hijo = FlipHeuristic(hijo)
        si Φ(hijo) = 1 entonces
            devolver True, hijo, t
si ∃ϵ_S ∈ P : Φ(ϵ_S) = 1 entonces
    devolver True, ϵ_S, t
en otro caso
    devolver False, Elitismo(P)[2]
```

---

3 Implementación

Para la implementación utilizamos el paquete `sympy` de Python que permite definir fórmulas booleanas y evaluarlas. Como debemos manejar fórmulas que contienen un gran número de variables y cláusulas, no podemos pretender introducirlas a mano. Vamos a trabajar con el formato `cnf`, que es un tipo de fichero de texto. Como ejemplo, vemos la estructura de un fichero que codifica la fórmula

$$(X_1 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee X_1)$$

que es, generalmente, de la forma

```
c
c comentarios
c
p cnf 3 2
1 -3 0
2 3 -1 0
%
```

Observamos que tiene un primer bloque de líneas de comentario que empiezan por el carácter `c`. Después, tenemos una línea que comienza por `p cnf` seguido de dos números que indican el número de variables y el número de cláusulas respectivamente. Cada una del resto de líneas hasta el símbolo opcional `%` codifica una de las cláusulas y siempre terminan en 0. El formato es claro, los números enteros que aparecen indican el índice de la variable correspondiente y el signo indica si está o no negada en la fórmula.

Importamos los paquetes necesarios para la ejecución del algoritmo.

```
from sympy import *
from random import random, shuffle, choices
from re import match
from datetime import *
```

Implementamos la siguiente función en Python para leer fórmulas en formato `cnf`, que requiere que la codificación termine en %.

Lectura

```
# Función que lee una fórmula bien formada del cálculo proposicional en forma
    normal
# conjuntiva de 3-cláusulas y la guarda como una fórmula del paquete sympy.
def lee_cnf(nom_archivo):
    fichero = open(nom_archivo, "r")  # Abrimos el fichero.
    linea = ""
    while not match("p", linea):  # Buscamos la línea de parámetros.
        linea = fichero.readline()
    linea = linea.split()  # La línea de parámetros.
    n_var = int(linea[2])  # Número de variables.
    n_cla = int(linea[3])  # Número de cláusulas.
    lineas = []
    linea = fichero.readline()  # Primera cláusula.
    while not match("%", linea):  # Guardamos cada línea hasta %.
        lineas.append(linea.split())
        linea = fichero.readline()
    fichero.close()  # Cerramos el fichero.
    lineas = [[int(_) for _ in lin][:-1] for lin in lineas]  # Quitamos los
        ceros.
    var("x1:" + str(n_var))  # Generamos las variables simbólicas.
    form = [[0] * 3 for _ in range(n_cla)]  # Inicializamos las cláusulas.
    for i in range(len(lineas)):  # Cambiamos de números a variables.
        for j in range(3):
            if lineas[i][j] > 0:
                form[i][j] = symbols("x" + str(lineas[i][j]))
            else:
                form[i][j] = ~symbols("x" + str(abs(lineas[i][j])))
    form = [_[0] | _[1] | _[2] for _ in form]  # Disyunciones
    FORM = And(*form)  # Fórmula en forma normal conjuntiva de 3-cláusulas.
    return FORM
```

Una vez que tenemos la entrada, podemos programar los diferentes métodos del algoritmo genético.

Variables

```
def variables(f):  # Devuelve las variables de la fórmula.
    return var("x1:" + str(len(f.atoms()) + 1))
```

Satisfacción

```
def satisface(pob, f):  # Comprueba si algún individuo de la población satisface
    la fórmula.
    xi = variables(f)
    for _ in pob:  # Recorremos la población.
        if F.subs(zip(xi, _)):  # Evaluamos la fórmula y, si se satisface,
            retornamos True y la interpretación.
            return True, _
    return False, None
```

Fitness

```
def fitness(inter, f):  # Número de cláusulas satisfechas por la interpretación.
    xi = variables(f)
    if not inter:  # Si es la interpretación vacía, es 0.
        return 0
    fit = 0
    for _ in f.args:  # Recorremos las cláusulas
        if _.subs(zip(xi, inter)):  # Si se satisface,
            fit += 1  # sumamos 1.
    return fit
```

Ganancia

```
def ganancia(inter, pos, f):  # Diferencia de fitness antes y después de cambiar
    la posición en la interpretación.
    xi = variables(f)
    clausulas = []
    for _ in F.args:  # Recorremos las cláusulas de la fórmula y guardamos las
        que contengan la variable.
        if xi[pos] in _.atoms():
            clausulas.append(_)
    inter2 = inter.copy()
    inter2[pos] ^= True  # Consideramos la interpretación con el opuesto para la
        variable.
    fit0, fit1 = 0, 0
    for _ in clausulas:  # Recorremos las cláusulas con la variable y calculamos
        la diferencia de fitness.
        if _.subs(zip(xi, inter)):
            fit0 += 1
        if _.subs(zip(xi, inter2)):
            fit1 += 1
    return fit1 - fit0, inter2
```

Búsqueda local

```
def flip_heuristic(inter, f):  # Búsqueda local que maximiza el fitness por
    intercambio de variables.
    list_var = list(f.atoms())
    n_var = len(list_var)
    index_var = list(range(n_var))
    shuffle(index_var)  # Mezclamos los índices de las variables en orden
        aleatorio.
    mejora = 1
    interr = inter.copy()
    while mejora > 0:  # Mientras sigamos mejorando
        mejora = 0
        for i in range(n_var - 1):  # Para cada variable en orden aleatorio.
            gain, inter2 = ganancia(interr, index_var[i], f)  # calculamos la
                ganancia al invertirla.
            if gain >= 0:  # Si la ganancia es positiva,
                interr = inter2  # aceptamos la inversión
                mejora += gain  # Hemos mejorado el fitness de la interpretación.
    return interr
```

Elitismo

```
def elitista2(pob):  # Toma una población y devuelve los dos individuos con
    mayor fitness para la selección elitista.
    pob2 = pob.copy()
    pob2.sort(key=fitness)  # Ordenamos la población de manera creciente según
        el fitness.
    return pob2[-1], pob2[-2]
```

Selección

```
def selec_padres(pob):  # Escoge dos padres con probabilidad proporcional al
    fitness.
    fit = [fitness(_) for _ in pob]
    total = sum(fit)
    prob = [_ / total for _ in fit]  # Lista de probabilidades proporcional al
        fitness.
    return choices(pob, prob, k=2)
```

Cruce uniforme

```
def cruce_uniforme(padre1, padre2):  # Devolvemos un hijo fruto de un cruce
    uniforme de los padres.
    mascara = [choices([True, False], k=len(padre1))]  # Generamos uniformemente
        una máscara de cruce.
    hijo = [False] * len(padre1)  # Inicializamos un hijo de ceros.
    for _ in range(len(mascara)):  # Recorremos la máscara.
        if mascara[_]:  # Si el elemento es 1,
            hijo[_] = padre1[_]  # el gen del hijo es el del padre1.
        else:  # Si el elemento es 0,
            hijo[_] = padre2[_]  # el gen del hijo es el del padre2.
    return hijo
```

Mutación

```
def mutacion(inter, pmut1, pmut2):  # Mutación del cromosoma inter.
    hijo = inter
    if random() <= pmut1:  # Decidimos si mutamos al hijo.
        for _ in range(len(hijo)):  # Recorremos los genes del hijo.
            if random() <= pmut2:  # Decidimos si mutamos el gen.
                hijo[_] ^= True  # Invertimos el valor del gen en la
                    interpretación.
    return hijo
```

Algoritmo genético

```
def genetic_sat(f, n_ind=10, tmax=30000, pmut1=0.9, pmut2=0.5, busqueda=True):
    n_var = len(F.atoms())
    P = [choices([True, False], k=n_var) for _ in range(n_ind)]  # Generamos la
        población inicial.
    if busqueda:
        P = [flip_heuristic(_) for _ in P]  # Aplicamos la búsqueda local a cada
            individuo de la población.
    t = 0
    while t < tmax and not satisface(P, f)[
        0]:  # Si no hemos llegado al número máximo de generaciones y la
            población no satisface la fórmula.
        t += 1
        print("Generación: ", t)
        P0 = P  # Reasignamos la población actual.
        P = []
        P.extend(elitista2(P0))  # Adición elitista de los dos mejores
            individuos de la población anterior.
        while len(P) < len(P0):
            padre1, padre2 = selec_padres(P0)  # Selección por ruleta de los
                padres.
            hijo = cruce_uniforme(padre1, padre2)  # Cruce uniforme de los
                padres.
            hijo = mutacion(hijo, pmut1, pmut2)  # Mutación del hijo.
            if busqueda:
                hijo = flip_heuristic(hijo, f)
            P.append(hijo)  # Añadimos el hijo a la población.
            print("Hijo", len(P))
            if satisface([hijo], f)[0]:
                print("La fórmula booleana introducida es satisfecha por la
                    interpretación:\n", hijo)
                return True, hijo, t
    sat = satisface(P, f)
    if sat[0]:
        print("La fórmula booleana introducida es satisfecha por la
            interpretación:\n", sat[1])
        return True, sat[1], t
    else:
        print("Se ha alcanzado el número máximo de generaciones.")
        return False, elitista2(P)[-1]
```

Ejecución

```
F = lee_cnf("uf20-01.cnf")  # Leemos la fórmula booleana del cnf.
tiempo0 = datetime.now()
genetic_sat(F)  # Ejecutamos el algoritmo genético con la búsqueda local.
tiempo1 = datetime.now()
print("Tiempo de cálculo: ", tiempo1 - tiempo0)  # Tiempo de cálculo.
```

Hemos probado los métodos con las siguientes instancias de 3-**SAT** satisfactibles obtenidas del repositorio de fórmulas booleanas SATLIB[1]:

1. Familia `UF` de instancias satisfactibles uniformes en la transición de fase ($m/n \simeq 4.26$).

   `uf20` Cuatro instancias con 20 variables y 91 cláusulas.

   `uf50` Cuatro instancias con 50 variables y 218 cláusulas.

   `uf75` Cuatro instancias con 75 variables y 325 cláusulas.

   `uf100` Cuatro instancias con 100 variables y 430 cláusulas

2. Dos fórmulas booleanas de la familia `AIM` de instancias satisfactibles con 50 variables, 80 cláusulas y solución única.

Como el algoritmo es una metaheurística con un gran número de parámetros, hemos escogido los valores de los mismos según [2]. Por lo tanto, fijamos que el número de individuos en una población es $k = 10$, el número máximo de generaciones $t_{\max} = 30000$, la probabilidad de mutar un cromosoma hijo $p_{m_1} = 0.9$ y la probabilidad de mutar un gen $p_{m_2} = 0.5$.

4 Resultados y conclusión

Para una fórmula booleana de `uf20`, el algoritmo encuentra una interpretación que la satisface en unos 7 segundos aplicando exclusivamente la búsqueda local sobre la población inicial sin necesidad de los operadores genéticos. En el caso de `uf50`, la mayoría de ejecuciones termina en 1 minuto y medio solo con la búsqueda local o en unos 5 minutos calculando 2 generaciones. Sin embargo, en algunos casos el proceso requiere 25 minutos y 5 generaciones. Para las instancias de `uf75` el resultado es similar, en algunas ejecuciones conseguimos encontrar una interpretación satisfactoria en unos 4 minutos sin necesidad de entrar en la parte genética del algoritmo, pero otras requieren entorno a 12 minutos y 2 generaciones o, como era de esperar, más de 25 minutos, superando el tiempo del caso anterior. Respecto a `uf100`, hemos conseguido encontrar una interpretación satisfactoria en unos 25 minutos con 2 generaciones, pero la mayoría de las veces el proceso continúa más allá de la hora sin encontrar ninguna interpretación válida.

Para las fórmulas booleanas de la familia `AIM`, el algoritmo avanza más rápidamente que en `uf50` porque tenemos el mismo número de variables pero muchas menos cláusulas. Sin embargo, el proceso sobrepasa la hora de ejecución y se obtienen decenas de generaciones sin encontrar la única interpretación que satisface la fórmula. Una posible explicación es la unicidad de la interpretación, lo que la hace difícil de encontrar, o también que el algoritmo genético funcione peor cuando el ratio del número de cláusulas entre el número de variables es menor. En cualquier caso, la lentitud general del algoritmo se podría mejorar con una programación más eficiente.

Tenemos que destacar la importancia de la búsqueda local. Cuando intentamos buscar una solución para `uf20` sin utilizar la búsqueda local, se obtienen centenares de generaciones requiriendo más de una hora de proceso sin encontrar ninguna interpretación satisfactoria, lo que contrasta con los 7 segundos en los que la búsqueda local resuelve estas instancias. Como sabemos, para fórmulas más complicadas se requieren algunas generaciones hasta llegar a una interpretación válida, con lo que la combinación del algoritmo genético con la búsqueda local es crucial.

[1] SATLIB: https://www.cs.ubc.ca/%7Ehoos/SATLIB/benchm.html

Bibliografía

[1] S. Cook, *The complexity of Theorem Proving Procedures*, 1971.

[2] E. Marchiori, C. Rossi, *A Flipping Genetic Algorithm for Solving Hard 3-SAT Problems*, 2000.

[3] G. Syswerda, *Uniform Crossover in Genetic Algorithms*. In J. Shaffer (ed.), Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989, 2-9.

[4] DIMACS (Center for Discrete Mathematics And Theoretical Computer Science), *Satisfiability Suggested Format*. 1993.