INSTITUTO POLITÉCNICO NACIONAL

Centro de Investigación en Computación

ASIGNATURA:

Metaheurísticas

Actividad #15:

Guía Taller Laboratorio No.6

"Solución de problemas mediante Recocido Simulado"

PROFESORA:

Dra. Yenny Villuendas Rey

PRESENTA:

Juan René Hernández Sánchez

Adriana Montserrat García Carrillo

Centro de Investigación en Computación

Instituto Politécnico Nacional

1. Introducción

Muchos problemas de ingeniería, planificación y fabricación pueden ser modelados como minimizar o maximizar una función de coste sobre un conjunto finito de variables discretas. Esta clase de problemas, llamados de optimización combinatoria, ha recibido mucha atención en las dos últimas décadas y se han conseguido importantes logros en su análisis. Uno de estos logros es la separación de esta clase en dos subclases. La primera contiene los problemas que pueden resolverse de forma eficiente, es decir, los problemas para los que se conocen algoritmos que resuelven cada instancia de forma óptima en tiempo polinómico. La segunda subclase contiene los problemas que son notoriamente difíciles, denominados formalmente NP-duros.

Para un algoritmo NP-duro se cree que no existe ningún algoritmo que resuelva cada instancia en tiempo polinómico. En consecuencia, hay instancias que requieren un tiempo súper polinomial o exponencial para ser resueltas de forma óptima. Está claro que los problemas difíciles deben tratarse en la práctica, lo cual puede hacerse mediante dos tipos de algoritmos: de optimización, que encuentran soluciones óptimas posiblemente utilizando grandes cantidades de tiempo de cómputo, o algoritmos heurísticos que encuentran soluciones aproximadas en pequeñas cantidades de tiempo de cálculo.

Los algoritmos de búsqueda local son del tipo heurísticos. Constituyen un enfoque general muy utilizado para los problemas de optimización combinatoria. Suelen ser instancias de varios esquemas de búsqueda general, pero todas tienen la misma característica de una función de vecindad subyacente, que se utiliza para guiar la búsqueda de una buena solución.

El recocido simulado, es uno de los algoritmos de búsqueda local más conocidos, ya que tiene un rendimiento bastante bueno y es ampliamente aplicable.

En la física de la materia condensada, el recocido simulado se conoce como un proceso térmico para obtener estados de baja energía de un sólido en un baño de calor. El proceso consiste en los siguientes dos pasos [1]:

- aumentar la temperatura del baño de calor hasta un valor máximo en el que el sólido se funde
- disminuir cuidadosamente la temperatura del baño de calor hasta que las partículas se organicen en el estado básico del sólido.

A continuación, se presentará una explicación del recocido simulado, sus características, aplicaciones e implementación

2. Desarrollo

Asignatura: Metaheurísticas
Actividad No.15
Guía Taller Laboratorio No.6
Título: Solución de problemas mediante Recocido Simulado

Contenido:

- Métodos heurísticos de solución de problemas.
- Recocido Simulado.

Objetivo: Modelar problemas clásicos de búsqueda mediante el uso de algoritmos de Recocido Simulado, para la solución de problemas de la profesión.

1. Detalle el pseudocódigo del algoritmo de Recocido Simulado [1].

Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

```
Input: Un Estado Aleatorio del problema
Output: Un óptimo local o global
X ← Random String                                    ▷ Solución inicial aleatoria
edoAnterior ← costo(X)
temp ← tempMax
for temp ≥ tempMin; temp := nextTemp(temp) do
    repeat
        sucesor ← funVecindad(X)
        edoNuevo ← costo(sucesor)
        Δ ← edoNuevo − edoAnterior
        if Δ < 0 then
            if random() ≥ exp(delta/K * temp) then
                rechazar(sucesor)
            else
                edoAnterior := edoNuevo
                X := sucesor
            end if
        else
            edoAnterior := edoNuevo
            X := sucesor
        end if
        iteracion := iteracion + 1
    until iteracion < iteracionMax                   ▷ Número de Vecinos a Evaluar
end for
return edoAnterior
```

2. Compárelo con el algoritmo de RMHC

Algorithm 1 Heurística Random mutation hill-climbing (RMHC)

Input: Un Estado Aleatorio del problema
Output: Un óptimo local

```
1: iterations ← S                         ▷ Siendo S ∈ ℕ Un número determinado de Evaluaciones
2: bestEvaluated ← random string
3: bestFitness ← ComputeFitness(bestEvaluated)
4: length ← bestEvaluated.lenght()
5: repeat
6:     locus ← Rand(0, length)
7:     mutatedHilltop ← Mutate(bestEvaluated, locus)
8:     mutatedFitness ← ComputeFitness(mutatedHilltop)
9:     if mutatedFitness ≥ bestFitness then
10:        bestEvaluated := mutatedHilltop
11:        bestFitness := mutatedFitness
12:    end if
13:    iterations := iterations − 1
14: until iterations ≠ 0
15: return bestEvaluated
```

2.1 Similitudes

- Ambos algoritmos toman una cadena aleatoria como estado inicial y se calcula la función costo o fitness respectivamente.

Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

Input: Un Estado Aleatorio del problema
Output: Un óptimo local o global

```
1: X ← Random String                      ▷ Solución inicial aleatoria
2: edoAnterior ← costo(X)
3: temp ← tempMax
```

Algorithm 1 Heurística Random mutation hill-climbing (RMHC)

Input: Un Estado Aleatorio del problema
Output: Un óptimo local

```
1: iterations ← S                         ▷ Siendo S ∈ ℕ Un número determinado de Evaluaciones
2: bestEvaluated ← random string
3: bestFitness ← ComputeFitness(bestEvaluated)
4: length ← bestEvaluated.lenght()
```

- En ambos algoritmos cuando hay una mejor solución se acepta de forma directa. Para el SA si el delta marca una mejora se entra en la condición else (líneas 16, 17 y 18). Para el RMHC si el fitness encontrado mejora la solución se asigna como mejor fitness (y aplican las líneas 9, 10 y 11).

Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

```
9:      if Δ < 0 then
10:         if random() ≥ exp(delta/K * temp) then
11:             rechazar(sucesor)
12:         else
13:             edoAnterior := edoNuevo
14:             X := sucesor
15:         end if
16:     else
17:         edoAnterior := edoNuevo
18:         X := sucesor
19:     end if
```

```
Algorithm 1 Heurística Random mutation hill-climbing (RMHC)

  if mutatedFitness ≥ bestFitness then
      bestEvaluated := mutatedHilltop
      bestFitness := mutatedFitness
```

- En ambos algoritmos se busca entre los vecinos más cercanos.

```
Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

      sucesor ← funVecindad(X)
```

```
Algorithm 1 Heurística Random mutation hill-climbing (RMHC)

  locus ← Rand(0, length)
  mutatedHilltop ← Mutate(bestEvaluated, locus)
```

2.2 Diferencias

- En el SA existe una combinación de parámetros para determinar el número de evaluaciones a realizar (temperatura y número de vecinos). Para el caso del RMHC las evaluaciones son predefinidas con una sola variable (número determinado de evaluaciones).

```
Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

for temp ≥ tempMin; temp := nextTemp(temp) do

    until iteracion < iteracionMax                ▷ Número de Vecinos a Evaluar
end for
```

```
Algorithm 1 Heurística Random mutation hill-climbing (RMHC)

iterations ← S                 ▷ Siendo S ∈ ℕ Un número determinado de Evaluaciones

until iterations ≠ 0
```

- En el RMHC se rechazan por completo los estados deficientes, mientras que en el SA existe la posibilidad de empeorar la solución, ya que se aceptan estados con un costo menor al costo actual.

```
Algorithm 1 Heurística Recocido Simulado (S.A. - Simulated annealing)

      if Δ < 0 then
          if random() ≥ exp(delta/K * temp) then
              rechazar(sucesor)
          else
              edoAnterior := edoNuevo
              X := sucesor
          end if
```

2.3 Conclusiones:

El SA posee mayor flexibilidad en cuanto a las soluciones que va evaluando, a diferencia del RMHC que siempre conservaba la mejor solución encontrada. Por lo tanto, el SA puede caer en errores o peores soluciones o lograr salir de óptimos locales.

Otra diferencia es que los ciclos del RMHC dependen del número de evaluaciones determinadas, mientras que en el SA se realiza un proceso completo, que se cumplirá en su totalidad hasta alcanzar cierta temperatura mínima.

3. Realice la modelación matemática necesaria para la solución, mediante SA, del Problema de la mochila (Knapsack problem).

Recuerde que la modelación matemática incluye: definición de los estados inicial y final, definición del test objetivo, y definición de las acciones posibles (operadores).

- **Estado inicial:**
  n productos con valor C_i y peso P_i que se deben poner en una mochila, $i = 1, \ldots, n$. Se tiene como límite un peso P_m a cargar por la mochila. $n \in N$. Los valores C_i , $P_i \in R_*^+$. El estado de la mochila se representa mediante una cadena binaria x de longitud n. El estado inicial es una cadena x con n 0's.

- **Estado final:**
  Una cadena x' tal que x' = $\{ x_i' \,|\, x_i' \in \{0,1\},\ i = 1, \ldots, n\}$ y donde $f(x) = \sum_{i=1}^{n} C_i x_i$ de manera que $f(x') = \max_x f(x)$.

- **Test objetivo:**
  $f(x) = \sum_{i=1}^{n} C_i x_i \quad \boldsymbol{s.a.} \ \sum_{i=1}^{n} P_i x_i \leq P_m,\ x_i \in \{0,1\}$

- **Acciones posibles (operadores):**
  $g: x \to x' \ \ \boldsymbol{s.a.}\ \exists x_i \neq x_i',\ i \in \{0, \ldots, n\}$ donde $x_i = 1$ si el producto se empaqueta o está dentro de la mochila, de lo contrario $x_i = 0$ si el producto no está empaquetado o se sacó de la mochila. Resultando de ahí la condición binaria de la variable x_i.

4. Realice una corrida manual del algoritmo de Recocido Simulado sobre el problema anterior. Defina para ello un esquema de recocido de su preferencia.

Descripción 5 productos:

Productos	Peso	Valor
P1	2	6
P2	1	9
P3	3	2
P4	5	5
P5	1	4

```
#Definición de parámetros
pesoMax = 10 kg       #peso máximo de la mochila
interacionMax = 3     #número de vecinos
a = 0.5               #valor de alfa

#Se selecciona un estado inicial al azar
X = 10010
#Se calcula la función de costo
edo_anterior = 13

#Se designan las temperaturas
temp = 30° C
tempMax = temp
tempMin = 10° C

#Evaluación vecino 1 de 3
sucesor = 11010
edo_nuevo = 18

#Calculo delta = edo_nuevo – edo_anterior
delta = 5        #Como delta es positivo y se está maximizando entonces se acepta el
edo_nuevo (edo_anterior = edo_nuevo)
edo_anterior = 18
x = 11010

#Evaluación vecino 2 de 3
sucesor = 11011
edo_nuevo = 24
delta = 6
edo_anterior = 24
x = 11011

#Evaluación vecino 3 de 3
sucesor = 11111
edo_nuevo = 0   #Se asigna el valor de 0, ya que el peso máximo de la mochila se excedió
delta = -24

#Se entra al if y se evalúa si es aceptado el estado que empeora
random = 0.5
var = exp(-24/30) = 0.44   #exp(delta/K*temp)
                           #como 0.5 > 0.44 se rechaza la solución

#Disminución de la temp
temp = a * temp = 0.5 * 30 = 15

#Se vuelven a realizar las evaluaciones
```

#Evaluación vecino 1 de 3
sucesor = 01011
edo_nuevo = 20
delta = -4
random = 0.3
var = exp(-4/15) = 0.76   #como 0.3 < 0.76 se acepta la solución
edo_anterior = 20
x = 01011

#Evaluación vecino 2 de 3
sucesor = 00011
edo_nuevo = 15
delta = -5
random = 0.6
var = exp(-5/15) = 0.71   #como 0.6 < 0.71 se acepta la solución
edo_anterior = 15
x = 00011

#Evaluación vecino 3 de 3
sucesor = 00111
edo_nuevo = 17
delta = 2
edo_anterior = 17
x = 00111

#Disminución de la temp
temp = a * temp = 0.5 * 15 = 7.5   #Como temp es menor a 10° C se termina el algoritmo

5. Para el problema planteado, proponga las estructuras de datos necesarias para su implementación

- cadenas
- listas

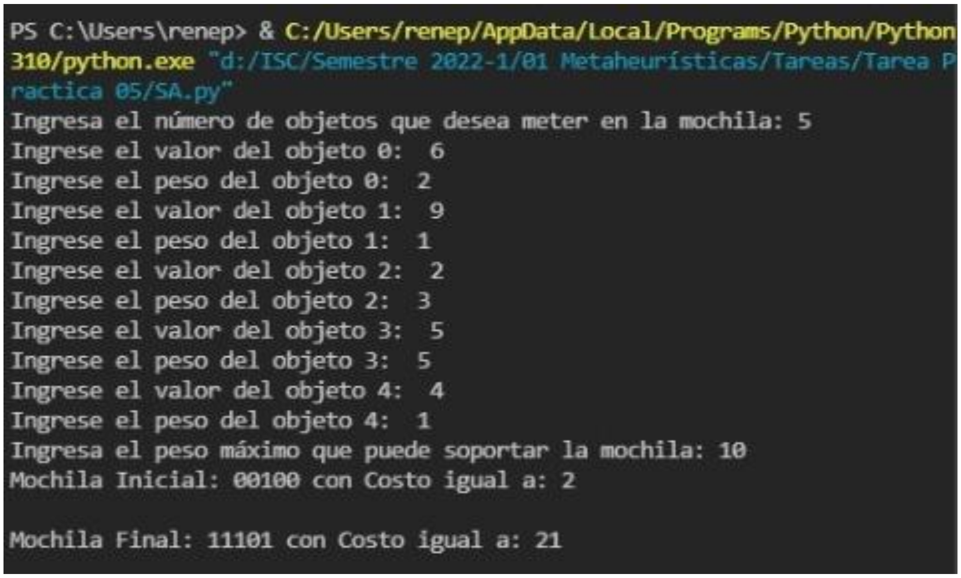
6. Diseñe la interfaz de usuario para la solución del problema planteado mediante Recocido Simulado.

```
import random
import math

class Objetos:
    def __init__(self, valor, peso):
        self.valor = valor
        self.peso = peso
```

```python
n = input('Ingresa el número de objetos que desea meter en la mochila: ')
n = int(n)

lista_objetos = []
for i in range(n):
    valor = input(f"Ingrese el valor del objeto {i}:  ")
    valor = int(valor)
    peso = input(f"Ingrese el peso del objeto {i}:  ")
    peso = int(peso)
    lista_objetos.append(Objetos(valor, peso))

W = input('Ingresa el peso máximo que puede soportar la mochila: ')
W = int(W)

def costo(mochila):
    Waux = W
    valmax = 0
    for j, objeto in enumerate(mochila):
        # Evaluando los objetos que están dentro de la mochila
        objeto_actual = lista_objetos[j]
        if objeto == '1':
            Waux = Waux - objeto_actual.peso
            valmax = valmax + objeto_actual.valor

    if Waux < 0:
        valmax = -1

    return int(valmax)

def vecindad(mochila):
    bit = 0
    locus = random.randint(0, n - 1)
    if mochila[locus] == '0':
        bit = 1
    new_mochila = mochila[0:locus] + str(bit) + mochila[locus+1:]
    return new_mochila

def generar_random(num):
    nmochila = ""
    for i in range(num):
        nmochila = nmochila + str(random.randint(0, 1))
    return nmochila

def rechazar():
    return ""
```

```python
temp_min = 0.1
temp_max = 300.0
vecinos = n - 1
alfa = 0.8
K = 1.0

mochila = generar_random(n)

edo_anterior = costo(mochila)

print("Mochila Inicial: " + mochila +
      " con Costo igual a: " + str(edo_anterior) + "\n")

temp = temp_max

while temp >= temp_min:
    vecinos_revisados = 0
    while vecinos_revisados < vecinos:
        sucesor = vecindad(mochila)
        edo_nuevo = costo(sucesor)
        delta = edo_nuevo - edo_anterior
        if delta < 0:
            if random.random() >= math.exp(delta/(K*temp)):
                sucesor = rechazar()
                edo_nuevo = 0
            else:
                edo_anterior = edo_nuevo
                mochila = sucesor
        else:
            edo_anterior = edo_nuevo
            mochila = sucesor
        vecinos_revisados = vecinos_revisados + 1
    temp = temp * alfa

print("Mochila Final: " + mochila +
      " con Costo igual a: " + str(edo_anterior) + "\n")
```

7. Ejecute la solución del problema planteado mediante Recocido simulado, utilizando para ello las estructuras de datos y la interfaz gráfica diseñadas.



```
PS C:\Users\renep> & C:/Users/renep/AppData/Local/Programs/Python/Python310/python.exe "d:/ISC/Semestre 2022-1/01 Metaheurísticas/Tareas/Tarea Practica 05/SA.py"
Ingresa el número de objetos que desea meter en la mochila: 5
Ingrese el valor del objeto 0:  6
Ingrese el peso del objeto 0:  2
Ingrese el valor del objeto 1:  9
Ingrese el peso del objeto 1:  1
Ingrese el valor del objeto 2:  2
Ingrese el peso del objeto 2:  3
Ingrese el valor del objeto 3:  5
Ingrese el peso del objeto 3:  5
Ingrese el valor del objeto 4:  4
Ingrese el peso del objeto 4:  1
Ingresa el peso máximo que puede soportar la mochila: 10
Mochila Inicial: 00100 con Costo igual a: 2

Mochila Final: 11101 con Costo igual a: 21
```

3. Conclusiones

Al realizar la corrida manual del algoritmo SA, se pudo apreciar que los algoritmos de búsqueda local pueden ser muy útiles si estamos interesados en el estado de la solución, pero no en el camino hacia ese objetivo. Estos algoritmos operan sólo en el estado actual y se mueven a estados vecinos. Al permitir un ascenso ocasional en el proceso de búsqueda, se puede escapar de la trampa de los mínimos locales, pero también existe la posibilidad de pasar óptimos globales después de alcanzarlos.

Se puede aplicar el SA para generar una solución a los problemas de optimización combinatoria asumiendo una analogía entre ellos y los sistemas físicos de muchas partículas con las siguientes equivalencias:

- Las soluciones del problema son equivalentes a los estados de un sistema físico.
- El costo de una solución es equivalente a la "energía" de un estado

Por último, es importante mencionar que los algoritmos de búsqueda local tienen dos ventajas fundamentales: usan muy poca memoria y pueden encontrar soluciones razonables en espacios de estados grandes o infinitos (continuos).

4. Referencias

[1] Burke & Kendall. Search Metodologies – 2005. Capítulo 7