Algoritmos de optimización - Trabajo práctico

Nombre y Apellidos: Mikel Pérez de Mendiola Rubio
Url: https://github.com/mikelpzm/03MIAR---Algoritmos-de-Optimizacion

Url Colab: https://drive.google.com/file/d/1zbIbzze5_gVUT8-FahlifQrntN9UPS-r/view?usp=sharing

Problema:

> 1. Combinar cifras y operaciones

Descripción del problema:
El problema consiste en analizar el siguiente problema y diseñar un algoritmo que lo resuelva.

- Disponemos de las 9 cifras del 1 al 9 (excluimos el cero) y de los 4 signos básicos de las operaciones fundamentales: suma(+), resta(-), multiplicación(*) y división(/)
- Debemos combinarlos alternativamente sin repetir ninguno de ellos para obtener una cantidad dada.
  Un ejemplo sería para obtener el 4:
  4+2-6/3*1 = 4

Debe analizarse el problema para encontrar todos los valores enteros posibles planteando las siguientes cuestiones:

- ¿Qué valor máximo y mínimo se pueden obtener según las condiciones del problema?
- ¿Es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo ?

....

(*) La respuesta es obligatoria

(*)¿Cuántas posibilidades hay sin tener en cuenta las restricciones?

Respuesta

Si calculamos de cuántas formas podemos combinar 9 cifras sin repetición tomadas de 5 en 5 con las 4 operaciones básicas, tenemos:

$$\frac{9!}{(9-5)!} \cdot 4! = 362880$$

Si pudieran repetirse las cifras y los operadores, tendríamos variaciones con repetición de 9 cifras tomadas de 5 en 5 con las combinaciones con repetición de 4 operadores:

$$9^5 \cdot 4^4 = 15116544$$

Modelo para el espacio de soluciones

(*) ¿Cual es la estructura de datos que mejor se adapta al problema? Argumentalo. (Es posible que hayas elegido una al principio y veas la necesidad de cambiar, arguentalo)

Respuesta

Se ha optado por usar listas para representar los números y las operaciones. Esto permite ir combinando los elementos para obtener todas las combinaciones posibles.
Al combinar los elementos, se van generando soluciones que, en caso de ser válidas, se añaden a la lista de soluciones en forma de string. De esta forma, mediante la función eval() se puede obtener el resultado de la expresión directamente.

Para resolver usando ramificación y poda, se genera una estructura de árbol que permite ir podando las ramas que no sean prometedoras.

Según el modelo para el espacio de soluciones

(*)¿Cual es la función objetivo?

Se trata de un problema de búsqueda, por lo que la función objetivo es encontrar la solución la expresión que de como resultado el número deseado.
A diferencia de otros tipos de problemas, no se trata aquí de ir maximizando o minimizando una función, sino de encontrar la expresión que de como resultado el número objetivo.

(*)¿Es un problema de maximización o minimización?

Como se ha comentado, se trata de un problema de búsqueda. En lugar de maximizar o minimizar, el objetivo es encontrar la expresión que al evaluarla de como resultado el número objetivo.

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

In [ ]:
```python
import itertools


def obtener_expresion_para_valor(objetivo):

  # conjunto de cifras y operadores
  cifras = [1, 2, 3, 4, 5, 6, 7, 8, 9]
  operadores = ['+', '-', '*', '/']

  expresion = ''

  # Generar todas las posibles combinaciones de cifras y operadores
  for c in itertools.permutations(cifras, 5):
```

```
        for o in itertools.permutations(operadores, 4):
            # Convertir la combinación a una expresión matemática
            expresion = str(c[0]) + o[0] + str(c[1]) + o[1] + str(c[2]) + o[2]
            # Evaluar la expresión
            try:
                resultado = eval(expresion)
                # Verificar si el resultado es un valor entero
                if resultado == int(resultado):
                    if resultado == objetivo:
                      return expresion, resultado

            except ZeroDivisionError:
                continue


expresion, resultado = obtener_expresion_para_valor(77)
print('expresion=', expresion, 'resultado=', resultado)
```

```
expresion= 7/1-2+8*9 resultado= 77.0
```

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

El algoritmo por fuerza bruta tiene una complejidad de es factorial ya que se generan todas las combinaciones posibles de los elementos de la lista de números y operaciones. Siendo n el número de elementos de la lista de cifras, el número de combinaciones posibles es $n!$, y siendo m el número de elementos de la lista de operaciones, el número de combinaciones posibles es $m!$. Entonces tendrá un orden de complejidad $O(n! \cdot m!)$.

¿Qué valor máximo y mínimo se pueden obtener según las condiciones del problema?

El máximo valor que se puede obtener es 77 y el mínimo es -69. Se presenta a continuación un método que devuelve la lista con todas las posibles soluciones y la solución máxima y mínima que se puede obtener. Este método genera las combinaciones y hace uso de poda para reducir el número de estas que se exploran, para descartar aquellas que no son prometedoras o válidas.

Comprobamos si la expresión contiene una multiplicación. Si es así, comprobamos si es la más alta posible (72).
Si es así, no seguimos explorando. Esto es válido para encontrar el máximo y el mínimo valor posible, ya que tendremos que tener 8*9 al maximizar, y algún dígito menos 89 *al minimizar*.
Se evalua la expresión entera construida hasta el momento, para evitar problemas con la precedencia de las operaciones para no descartar expresiones como $8/19$ que pueden llevar a resultados máximos o mínimos.
También descartamos las combinaciones que no lleven a soluciones enteras.

In [ ]:
```
OPERATIONS = ['*','+','-','/']
NUMBERS = [1,2,3,4,5,6,7,8,9]

counter = 0
```

```python
# lista que contiene todas las soluciones, cada solución se representa como
solutions = []

def obtener_todas_soluciones():
    max_value = float("-inf")
    max_expresion = ""

    # el valor mínimo inicial es -infinito
    min_value = float("inf")
    min_expresion = ""
    no_explorados = []

    def crear_operacion(numbers, operations):
        operation = str(numbers[0])
        for i in range(1,len(numbers)):
            operation += operations[i-1] + str(numbers[i])
        return operation

    def resolver(numbers, operations):

        nonlocal max_value, max_expresion
        nonlocal min_value, min_expresion
        nonlocal no_explorados

        def resolver_anadir_cifras(numbers, operations):
            for n in set(NUMBERS)-set(numbers):
                numbers.append(n)
                if resolver(numbers, operations):
                    return True
                numbers.pop()
            return False


        if len(operations) == 4:
            expresion = str(numbers[0]) + operations[0] + str(numbers[1]) +
            result = eval(expresion)
            if result == int(result):
                if result > max_value:
                    max_value = result
                    max_expresion = expresion
                if result  < min_value:
                    min_value = result
                    min_expresion = expresion

                solutions.append((expresion, result))
        elif not numbers:
            return resolver_anadir_cifras(numbers, operations)
        else:
            # print('Numbers, operations', numbers, operations)

            # comprobamos si la expresión contiene una multiplicación. Si es
            # Si es así, no seguimos explorando. Esto es válido para encontr
            # tener 8*9 al maximizar, y algún dígito menos 8*9 al minimizar.
            # Se evalua la expresión entera construida hasta el momento, par
            # para no descartar expresiones como 8/1*9 que pueden llevar a r

            if '*' in operations:
                valor_multiplicacion = eval(crear_operacion(numbers, operati
                if valor_multiplicacion < 72:
                    #print('No se explora por multiplicación', numbers, oper
                    return False

            #comprobamos si las operaciones contienen división y multiplicac
```

```
            if '/' in operations and '*' in operations:
                resultado_division = eval(crear_operacion(numbers, operation
                # si la expresión contiene una división y el resultado no es
                if resultado_division != int(resultado_division):
                    #print('No se explora por división', numbers, operations
                    return False

            for o in set(OPERATIONS)-set(operations):
                operations.append(o)
                if resolver_anadir_cifras(numbers, operations):
                    return True
                operations.pop()
        return False

    numbers = []
    operations = []

    resolver(numbers, operations)

    return max_expresion, max_value, min_expresion, min_value, solutions

max_expresion, max_value, min_expresion, min_value, solutions = obtener_toda
print('Máximo valor:', max_value, 'Expresión:', max_expresion)
print('Mínimo valor:', min_value, 'Expresión:', min_expresion)


#recorrer todas las soluciones y comprobar que existen todos los valores des
soluciones_faltantes = []
for i in range(-69, 78):
    if i not in [s[1] for s in solutions]:
        soluciones_faltantes.append(i)

print('Soluciones faltantes:', soluciones_faltantes)
```

```
Máximo valor: 77.0 Expresión: 7-2/1+8*9
Mínimo valor: -69.0 Expresión: 1+4/2-8*9
Soluciones faltantes: []
```

¿Es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo?

Sí, hemos comprobado en la ejecución anterior que se encuentran todos los valores entre -69 y 77 en las soluciones encontradas.

(*)Diseña un algoritmo que mejore la complejidad del algortimo por fuerza bruta. Argumenta por qué crees que mejora el algoritmo por fuerza bruta

Respuesta

Además del algoritmo anterior que generaba todos los resultados enteros posibles, se presenta a continuación una alternativa al algoritmo de fuerza bruta que obtiene la expresión que da como resultado el número dado.

En este enfoque, usando técnica de divide y vencerás, se pretende no tener que explorar todas las combinaciones posibles sino centrarse en las ramas prometedoras que puedan llevarnos al resultado deseado.

In [ ]:

```
from random import sample

random.seed(42)
cifras = ["1","2","3","4","5","6","7","8","9"]
operaciones = ["+","-","*","/"]

def generar_nodo(padre = None, cifras = cifras, operaciones = operaciones):
    """
    Dado un nodo añade una operacion y una cifra de las que quedan.
    """
    if padre is None:
        return sample(operaciones,1)[0].join(sample(cifras,2))
    else:
        op_left = list(set(operaciones) - set(padre) - set(cifras))
        cif_left = list(set(cifras) - set(padre) - set(operaciones))
        if len(op_left) >= 1:
            return padre + sample(op_left,1)[0] + (sample(cif_left,1))[0]
        else:
            return padre

def calcular_cotas(nodo, cifras = cifras, operaciones = operaciones):
    """
    Calculamos cotas aproximadas.
    """
    if eval(nodo) > 0:
        op_max = ["*","+"]
        op_min = ["/","-"]
    else:
        op_max = ["/","+"]
        op_min = ["*","-"]

    op_left_max = list(set(op_max) - set(nodo))
    op_left_min = list(set(op_min) - set(nodo))
    cif_left = list(set(cifras)-set(nodo))
    if len(op_left_max) > 0:
        cota_max = eval(nodo + op_left_max[0] + max(cif_left))
    else:
        cota_max = eval(nodo)

    if len(op_left_min) > 0:
        cota_min = eval(nodo + op_left_min[0] + max(cif_left))
    else:
        cota_min = eval(nodo)

    return cota_max, cota_min


def evaluar(nodo, lista, objetivo):
    """
    Decidimos si continuar con ese nodo o no. Si es no se añade a la lista d
    Esto se realiza comprobando las cotas máxima y mínima estimadas que pued
    faltan por usarse y las cifras restantes.
    También se deja de explorar en caso de que se hayan usado los operadores
    entero. En estos casos, no se va a poder llegar nunca a una solución ent
    """

    cota_max, cota_min = calcular_cotas(nodo)

    if '/' in nodo and '*' in nodo:
        resultado_division = eval(nodo)
        # si la expresión contiene una división y el resultado no es un núme
        if resultado_division != int(resultado_division):
          lista.append(nodo)
```

```
            #print('No se explora por división', nodo)
            return False, lista


    if cota_max > objetivo and cota_min < objetivo:
        #print("Continuar, cotas: ", cota_max, cota_min)
        return True, lista
    else:
        #print("Atras, cotas: ", cota_max, cota_min)
        lista.append(nodo)
        return False, lista


def ramif_y_poda(objetivo, nodo=None, lista = [], cifras = cifras, operacion
    """
    Funcion principal recursiva
    """
    if contador_repeticiones > 7: #Si no encontramos la solucion por ese cam
        return ramif_y_poda(objetivo = objetivo, nodo=None, lista = lista, c

    # 1. Generar nodo a partir del que llega.
    nuevo_nodo = generar_nodo(nodo)
    #print("Nuevo nodo: ", nuevo_nodo)

    if nuevo_nodo in lista: # Si está en la lista no lo exploramos
        contador_repeticiones += 1
        return ramif_y_poda(objetivo = objetivo, nodo = nodo, lista = lista,

    if len(nuevo_nodo) == 9: # Si ya tenemos los 9 elementos evaluamos si es
        if eval(nuevo_nodo) == objetivo:
            return nuevo_nodo
        else: # Si no es se añade a la lista y se vuelve atrás
            lista.append(nuevo_nodo)
            return ramif_y_poda(objetivo = objetivo, nodo = nodo, lista = li

    # Si el nodo no tiene todos los elementos se evaluan sus cotas y se deci
    continuar, lista = evaluar(nuevo_nodo, lista, objetivo)
    if continuar:
        return ramif_y_poda(objetivo = objetivo, nodo = nuevo_nodo, lista =
    else:
        return ramif_y_poda(objetivo = objetivo, nodo = nodo, lista = lista,

resultado = ramif_y_poda(77, nodo=None, lista = [], cifras = cifras, operaci

print("Expresión: ", resultado, "Resultado: ", eval(resultado))
```

```
Expresión:  7+8/1*9-2 Resultado:  77.0
```

(*)Calcula la complejidad del algoritmo

Respuesta

En este caso, la complejidad del algoritmo es menor que la del algoritmo usando fuerza bruta, ya que no se tienen que explorar todas las combinaciones gracias a la poda. En este tipo de algoritmos es complicado calcular la complejidad, ya que no es fácil saber cuántas combinaciones van a ser descartadas. Podríamos quedarnos con que en el peor caso, en el que no consiga podar ninguna candidata y tengamos complejidad de orden factorial. En la práctica estaríamos aún así mejorando la complejidad de la fuerza bruta, por descartarse combinaciones no prometedoras que en el caso de la fuerza bruta nos veríamos obligados a comprobar.

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

Respuesta

En este caso no tiene sentido plantear un juego de datos de entrada aleatorios, ya que el problema consiste en generar todas las combinaciones de números y operadores.

Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo

Respuesta

Se ha revisado los apuntes de clase y bibliografía, así como la documentación de la librería itertools, usada para obtener las combinaciones de los elementos de la lista de números y operaciones.

Describe brevemente las lineas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

Respuesta

Se podría mejorar el método de poda usado para descartar las combinaciones que no llevan al resultado deseado.

Si el problema creciese en tamaño, aumentando el número de cifras y operaciones que se deberían explorar, tener un mecanismo de poda eficiente sería necesario para poder resolver el problema.

Como alternativa, se podrían usar otros tipos de algoritmos para intentar obtener soluciones aceptables, como el ejemplo presentado a continuación para intentar encontrar el valor máximo que se puede obtener con las combinaciones de cifras y operadores.

En este caso se realiza un enfoque probabilístico, generando aleatoriamente combinaciones de números y operadores y comprobando si se obtienen soluciones válidas. En cada iteración se va guardando la mejor solución encontrada hasta el momento, y se genera una nueva combinación, modificando cifras y operadores basándose en una probabilidad.

In [ ]:
```
import itertools
import random


# Conjunto de cifras y operadores
cifras = [1, 2, 3, 4, 5, 6, 7, 8, 9]
operadores = ['+', '-', '*', '/']

mejor_solucion = -99999
```

```python
mejor_expresion = ''


def generar_vecino(c,o):

    c_vecino = c.copy()
    o_vecino = o.copy()

    #con una probabilidad p se hace un swap de c de la posición i con la pos
    #con probabilidad 1-p se obtiene otra cifra de las disponibles y se camb
    p = 0.8
    if random.random() < p:
        i = random.randint(0,4)
        j = random.randint(0,4)
        while i == j:
           j = random.randint(0,4)
        c_vecino[i], c_vecino[j] = c_vecino[j], c_vecino[i]
    else:
        i = random.randint(0,4)
        c_vecino[i] = random.choice([x for x in cifras if x not in c_vecino]


    #se hace un swap de o de la posición i con la posición j. i y j son núme
    #con probabilidad p se realiza el swap de operadores

    p = 0.2
    if random.random() < p:
        i = random.randint(0,3)
        j = random.randint(0,3)
        while i == j:
            j = random.randint(0,3)
        o_vecino[i], o_vecino[j] = o_vecino[j], o_vecino[i]

    return c_vecino, o_vecino


# Conjunto vacío para almacenar los valores enteros posibles
valores_posibles = set()

# Inicializar una lista tabú vacía
lista_tabu = []

# Generar una solución inicial al azar
c = random.sample(cifras, 5)
o = random.sample(operadores, 4)
expresion = str(c[0]) + o[0] + str(c[1]) + o[1] + str(c[2]) + o[2] + str(c[3

# Evaluar la solución inicial y verificar si es un valor entero
try:
    resultado = eval(expresion)
    if resultado == int(resultado):
        valores_posibles.add(int(resultado))
        lista_tabu.append(expresion)
        if(int(resultado) > mejor_solucion):
            mejor_solucion = int(resultado)
            mejor_expresion = expresion
            print(expresion, '=', resultado)
except ZeroDivisionError:
    pass

#condición de parada: cuando se encuentren 1000 valores enteros o cuando se
# sin encontrar nuevos valores enteros (en este caso, se puede detener el al

#numero de iteraciones
```

```
iteracionesSinMejora = 0
totalIteraciones = 0

while len(valores_posibles) < 1000 and iteracionesSinMejora < 1000:
    iteracionesSinMejora += 1
    totalIteraciones += 1
    # Generar una solución vecina (cambiando un operador o un número en la s

    c_vecino, o_vecino = generar_vecino(c,o)

    expresion_vecina = str(c_vecino[0]) + o_vecino[0] + str(c_vecino[1]) + c

    # Verificar si la solución vecina está en la lista tabú
    if expresion_vecina not in lista_tabu:
        # Evaluar la solución vecina
        try:
            resultado_vecino = eval(expresion_vecina)

            # print(expresion_vecina, resultado_vecino)
            # print(len(valores_posibles))

            if resultado_vecino == int(resultado_vecino):
                valores_posibles.add(int(resultado_vecino))

                if(int(resultado_vecino) > mejor_solucion):
                    mejor_solucion = int(resultado_vecino)
                    mejor_expresion = expresion_vecina
                    iteracionesSinMejora = 0

                    # Actualizar la solución actual
                    c = c_vecino
                    o = o_vecino
                    expresion = expresion_vecina

                    print(expresion_vecina, '=', resultado_vecino)
        except ZeroDivisionError:
            pass

        # Añadir la solución vecina a la
        lista_tabu.append(expresion_vecina)
        # Actualizar la lista tabú
        if len(lista_tabu) > 50:
            lista_tabu.pop(0)
```

```
4*3/6-5+8 = 5.0
4*3/1-5+8 = 15.0
4*8/1-5+3 = 30.0
4*9/1-5+3 = 34.0
8*9/1-5+3 = 70.0
8*9/1-3+5 = 74.0
8*9/1-2+5 = 75.0
8*9/1-2+7 = 77.0
```