Algoritmos de optimización - Seminario

Nombre y Apellidos: Sergi Ribera Ortells Url: https://github.com/riorser/AlgoritmosOptimizacion-.git 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.

Para empezar con el análisis del problema vamos a contar el número de operaciones que se tiene que hacer si resolvemos esto por fuerza bruta, para ello vemos que tenemos que elegir 5 números de 9 sin repeticiones, esto es: 9x8x7x6x5 todo esto por las posibles combinaciones de los 4 operadores, que sería: 4x3x2x1, esto nos da un resultado de 9!, lo que equivale a 362.880 posibles combinaciones, con un mínimo 4 operaciones, a lo que se le añaden las asignaciones, concatenaciones y diferentes operaciones que se deben hacer para que el código funcione.

Esto no es algo que tenga un coste computacional excesivo, ya que con un ordenador actual esta cantidad de operaciones se puede hacer en unos segundos, pero si por casualidad decidimos añadir más operaciones, o más cifras o hacer que se puedan repetir cifras y operadores, podría llegar a aumentar mucho el tiempo de ejecución, por ese motivo se tiene que buscar alguna solución para disminuir el número de operaciones que se realizan.

Para empezar vamos a ver cuales son las posibles soluciones que nos puede devolver el problema. Para eso si que vamos a usar la fuerza bruta y nos quedaremos con las soluciones enteras.

```
#primero vamos a importar las librerias necesarias para este trabajo
en esta celda:
from itertools import permutations, product
import math
import random
import itertools

def evaluate_expression(numbers, operators):
  # A esta función se le pasa una lista de 5 digitos y 4 operadores, y
los coloca en el orden correspondeinte.
  # Por ejemplo: (1, 2, 3, 4, 5) y ('+', '-', '*', '/') no devuelve la
expresión  '1+2-3*4/5' y la evalua.
    expression = ""
    for i in range(len(numbers)):
        expression += str(numbers[i])
```

```
        if i < len(operators):
            expression += operators[i]
    return eval(expression)

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
operators = ['+', '-', '*', '/']
iter = 0
iter_int = 0
posibles_valores = []


for number_perm in permutations(numbers, 5):
    for operator_comb in permutations(operators):
        result = evaluate_expression(number_perm, operator_comb)
        iter += 1
        if result-math.trunc(result)==0:
            iter_int+=1
            posibles_valores.append(result)


print(f"Valor máximo: {max(posibles_valores)}")
print(f"Valores mínimo: {min(posibles_valores)}")
print(f"Valores posibles: {set(posibles_valores)}")
print(f"Total iteraciones: {iter}")
print(f"Total iteraciones con resultado entero: {iter_int}")
```

```
Valor máximo: 77.0
Valores mínimo: -69.0
Valores posibles: {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0,
21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0,
32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0, 41.0, 42.0,
43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0, 50.0, 51.0, 52.0, 53.0,
54.0, 55.0, 56.0, 57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0,
65.0, 66.0, 67.0, 68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0,
76.0, 77.0, -69.0, -68.0, -67.0, -66.0, -65.0, -64.0, -63.0, -62.0, -
61.0, -60.0, -59.0, -58.0, -57.0, -56.0, -55.0, -54.0, -53.0, -52.0, -
51.0, -50.0, -49.0, -48.0, -47.0, -46.0, -45.0, -44.0, -43.0, -42.0, -
41.0, -40.0, -39.0, -38.0, -37.0, -36.0, -35.0, -34.0, -33.0, -32.0, -
31.0, -30.0, -29.0, -28.0, -27.0, -26.0, -25.0, -24.0, -23.0, -22.0, -
21.0, -20.0, -19.0, -18.0, -17.0, -16.0, -15.0, -14.0, -13.0, -12.0, -
11.0, -1.0, -9.0, -8.0, -7.0, -6.0, -5.0, -4.0, -3.0, -2.0, -10.0}
Total iteraciones: 362880
Total iteraciones con resultado entero: 90000
```

Con esto hemos obtenido el número de resultados enteros que se obtienen al iterar sobre toas las posibles combinaciones. también el valor máximo y mínimo que se puede obtener con estas combinaciones, y comprobamos que el número de permutaciones posibles es el que hemos

comentado antes. Vamos ahora a ver si entre el máximo y el mínimo hay algun valor que no se pueda obtener:

```
for i in range(int(min(posibles_valores)),
int(max(posibles_valores))):
  if i not in posibles_valores:
    print(f"falta {i}")
```

Como no nos ha devuelto nada sabemos que todos los números enteros entre -69 y 77 se puede obtener con una combinación como las anteriores.

Ahora vamos a construir la función por furza bruta y luego veremos una alternativa a la fuerza bruta:

```
def find_expression(target):
  # Con esta función hacemos todas las permutaciónes de los 9 digitos
y las 4 operaciones para buscar el valor target.
  # Si no encuentra ninguna expresion que devuelva ese valor nos
devuelve un none.
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    operators = ['+', '-', '*', '/']
    iter = 0
    posibles_valores = []


    for number_perm in permutations(numbers, 5):
        for operator_comb in permutations(operators):
            result = evaluate_expression(number_perm, operator_comb)
            iter += 1
            if result == target:
                expression = ""
                for i in range(len(number_perm)):
                    expression += str(number_perm[i])
                    if i < len(operator_comb):
                        expression += operator_comb[i]
                return expression, iter

    return None, iter


expression, iter = find_expression(77)

if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 7/1-2+8*9, en 242517 iteraciones
```

```
expression, iter = find_expression(-69)

if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 1+4/2-8*9, en 10661 iteraciones
```

```
expression, iter = find_expression(7)

if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 1-2*3/6+7, en 298 iteraciones
```

```
expression, iter = find_expression(87)

if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
No se encuentra expresión. Número de iteraciones totales: 362880.
```

En función de que valor busquemos tardará más o menos, , cuando más centrico sea el número más posibilidades de encontrarlo pronto, ya que más posibles combinaciones hay para este valor.

Cuando le pasamos un valor fuera del rango explora todas las posibles combinaciones y devuelve un None, vemos que ha tardado 3 segundos en hacer esta exploración.

Esto se podría mejorar considerando que hay ciertos valores que se repiten, por ejemplo es lo mismo '3+4x5-6/7' que '4x5+3-6/7'. incluso esta solución no se debería considerar ya que es un número no entero. Se puede buscar la forma de excluir ciertas operaciones buscando si la fración produce un valor válido y evitando que se evaluen expresiones con los mismo valores, esto mejoraría mucho la eficiencia de este código, con unas buenas restricciones para excluir todas las posibles soluciones que no sean necesarias.

```
def generate_initial_state(operations, numbers):
    # Generar una expresión inicial aleatoria
    random_numbers = random.sample(numbers, 5)
    random_operations = random.sample(operations,4)
    expression = str(random_numbers[0])
    for i in range(1, len(random_numbers)):
        expression += random_operations[i-1] + str(random_numbers[i])
```

```
    return expression

def generate_neighbor(expression):
    # Generar un vecino aleatorio modificando la expresión actual
    # Permutamos dos operadores y cambiamos una cifra por otra no
presente en la expresion
    expr_list = list(expression)
    index = random.sample([1,3,5,7], 2)
    expr_list[index[0]], expr_list[index[1]] = expr_list[index[1]],
expr_list[index[0]]
    index_num = random.choice([0, 2, 4, 6 , 8])
    expr_list[index_num] = str(random.choice([ x for x in numbers if
str(x) not in expression]))
    return ''.join(expr_list)

def bajar_temp(T):
    # Mutiplicamos la temperatura por el factor de enfriamiento
    return T*0.9




def simulated_annealing(target):
    """Encuentra una combinación de números y operaciones que resulte
en el número objetivo usando Recocido Simulado"""
    operations = ['+', '-', '*', '/']
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]



    current_state = generate_initial_state(operations, numbers)
    current_value = eval(current_state)
    current_heuristic = abs(current_value - target)

    T = 1.0  # Temperatura inicial
    T_min = 0.000000001  # Temperatura mínima
    iter = 1 # contar el número de iteraciones

    while T > T_min:
        for _ in range(100):  # Número de iteraciones por temperatura
            neighbor_state = generate_neighbor(current_state)
            neighbor_value = eval(neighbor_state)
            neighbor_heuristic = abs(neighbor_value - target)
            iter += 1

            if neighbor_heuristic == 0:
                return neighbor_state, iter

            delta_heuristic = neighbor_heuristic - current_heuristic
            if delta_heuristic < 0 or math.exp(-delta_heuristic / T) >
```

```
random.random():
                current_state = neighbor_state
                current_value = neighbor_value
                current_heuristic = neighbor_heuristic

        T = bajar_temp(T)

    return None, iter


expression, iter = simulated_annealing(8)


if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 9*1-3+4/2, en 186 iteraciones
```

```
expression, iter = simulated_annealing(76)


if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 8/1*9-2+6, en 485 iteraciones
```

```
expression, iter = simulated_annealing(-69)


if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
Expresión : 1+6/3-8*9, en 168 iteraciones
```

```
expression, iter = simulated_annealing(90)


if expression:
    print(f"Expresión : {expression}, en {iter} iteraciones")
else:
    print(f"No se encuentra expresión. Número de iteraciones totales:
{iter}.")
```

```
No se encuentra expresión. Número de iteraciones totales: 19701.
```

Con este algoritmo heurístico de recocido simulado se disminuye la cantidad de iteraciones que se realizan para encontrar un valor pero también es probable que si no encuentra el valor en un número determiado de iteraciones nos devuelva que no hay expresión, y esto puede ser un problema, aunque con el análisis previo sobre el valor mínimo y ,máximo sabemos que si no encuentra una solución para un valor que si que deberia tener podemos ajustar los parametros de temperatura o el factor de enfriamiento para que realice más operaciones, aún perdiendo un poco de eficiencia.

Es una buena solución para disminuir el tiempo de de ejecución y el número de veces que realiza las operaciones.

Otra solución con algoritmos heuristicos podria ser por algoritmos genéticos, usando el algoritmo de la coloniade hormigas por ejemplo.

Pero de momento no es algo que crea que sea necesario implementar ya que con el recocido simulado se ve como el número de iteracioes que se realizan tiende a ser bastante bajo.

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

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Respuesta

```python
def evaluate_expression(numbers, operators):
  # A esta función se le pasa una lista de 5 digitos y 4 operadores, y los coloca en el orden correspondeinte.
  # Por ejemplo: (1, 2, 3, 4, 5) y ('+', '-', '*', '/') no devuelve la expresión  '1+2-3*4/5' y la evalua.
    expression = ""
    for i in range(len(numbers)):
        expression += str(numbers[i])
        if i < len(operators):
            expression += operators[i]
    return eval(expression)

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
operators = ['+', '-', '*', '/']
iter = 0
iter_int = 0
posibles_valores = []


for number_perm in permutations(numbers, 5):
    for operator_comb in permutations(operators):
        result = evaluate_expression(number_perm, operator_comb)
        iter += 1
        if result-math.trunc(result)==0:
            iter_int+=1
            posibles_valores.append(result)
```

```
print(f"Total iteraciones: {iter}")
print(f"Total iteraciones con resultado entero: {iter_int}")
```

```
Total iteraciones: 362880
Total iteraciones con resultado entero: 90000
```

como hemos visto antes, hay 362.880 posibles permutaciones, y si excluimos las soluciones con números decimales podemos reducir esto a 90.000 posibilidades. Si a esto le añadimos que no todas las permutaciones de los operadores son necesarias, podemos resumir en que de las 24 posibles permutaciones solo se necesitan las siguientes:

- ['+', '-', 'x', '/'] ~ ['+', '-', '/', 'x'], ['-', 'x', '/', '+'], ['-', '/', 'x', '+']
- ['+', 'x', '-', '/'] ~ ['x', '+', '-', '/'], ['-', '/', '+', 'x'], ['x', '-', '/', '+']
- ['+', '/', '-', 'x'] ~ ['/', '-', 'x', '+'], ['-', 'x', '+', '/'], ['/', '+', '-', 'x']
- ['/', '+', 'x', '-'] ~ ['x', '-', '+', '/'], ['x', '+', '/', '-'], ['/', '-', '+', 'x']
- ['+', 'x', '/', '-'] ~ ['+', '/', 'x', '-'], ['-', '+', '/', 'x'], ['-', '+', 'x', '/'], ['/', 'x', '+', '-'], ['/', 'x', '-', '+'], ['x', '/', '+', '-'], ['x', '/', '-', '+']

Esto nos indica que el número de posibles combinaciones excluyendo combinaciones que son equivalentes y combinaciones que nos llevan a soluciones con decimales es de : 90000x5/24 = 18.750, un número bastante pequeño para la aplitud inicial que tenia el problema.

Queda bastante claro que si aplicamos algunas restricciones a el problema por fuera bruta, podando las soluciones no viables, se podría llegar a optimizar mucho este algoritmo. aunque por otro lado tendremos que ver todas las permutaciones númericas y realizar un mínimo de operaciones en cada iteración para ver si la solución es o no viable.

Así que de momento nos quedamos con el algoritmo de recocido simulado.

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

Para este problema en concreto creo que la estructura que mejor se adapta sería la estructura de Arbol, empezando por cada uno de los dígitos y creando ramas con las diferentes opciones de operaciones.

Esto puede beneficiarnos para descartar ramas que no son viables, haciendo evaluaciones parciales de las soluciones para poder directamente descartar conjuntos de soluciones de forma eficiente.

También por su estructura es una forma muy clara y visual, que puede ayudar a entender mejor el problema.

Según el modelo para el espacio de soluciones **(*)¿Cual es la función objetivo?**

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

Respuesta

Como no se trata de un problema de optimización, si no de busqueda, no es un problema que tenga una funcion objetivo asociada, ya que no hay una funcion que se tenga que maximizar o minimizar.Lo único que podemos definir es:

f(x,expresion) = x - eval(expresion)

para alguna expresión de las que se a definido en el problema. Pero esto no se puede considerar función objetivo, ya que un valor mínimo o máximo para esta función no se puede considerar solución, solo se puede considerar que la expresión es solución si

f(x, expresion) = 0

Esto por tanto responde a la siguiente pregunta, no se trata de un problema de minimización ni de maximización, se trata de un problema de busqueda.

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

```
def find_expression(target):
  # Con esta función hacemos todas las permutaciónes de los 9 digitos
y las 4 operaciones para buscar el valor target.
  # Si no encuentra ninguna expresion que devuelva ese valor nos
devuelve un none.
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    operators = ['+', '-', '*', '/']
    iter = 0
    posibles_valores = []


    for number_perm in permutations(numbers, 5):
        for operator_comb in permutations(operators):
            result = evaluate_expression(number_perm, operator_comb)
            iter += 1
            if result == target:
                expression = ""
                for i in range(len(number_perm)):
                    expression += str(number_perm[i])
                    if i < len(operator_comb):
                        expression += operator_comb[i]
                return expression, iter

    return None, iter
```

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

Este es el algoritmo que hay más arriba, la complejidad sería, para este problema en cuentión, realizando 4 operaciones sobre 5 dígitos elegidos entre una lista de 9 dígitos, como hemos dicho antes el numero total de permutacíones es de 362.880 esto lo tenemos que mutiplicar por el núero de operaciones que se realizan en cada permutacion, que son 4 operaciones más la concatenación de los dígitos con los operadores, que son 9 (esto no estoy del todo seguro de que se tenga que considerar) por lo tanto 13 x 362880 = 4.717.440 es el total de operaciones que realiza el cógigo (esto en el caso de que se ejeute para todas las permutaciones y no encuentre el valor introducido) y a esto se le sumarian las asignaciones iniciales de los vectores numbers y operations +2.

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

Respuesta

El algoritmo que he diseñado al inicio del seminario es un ejemplo de optimización del problema, pero como he comentado antes puede tener un problema, y es que si no encuentra la solución antes de que la tempereature alcance el mínimo, teniendo solucion puede fallar. la solución es aumentar la temperatura. Pero al estar acercandose constantemente al valor buscado es más facil que llegue a la solución antes de fallar.

```
def generate_initial_state(operations, numbers):
    # Generar una expresión inicial aleatoria
    random_numbers = random.sample(numbers, 5)
    random_operations = random.sample(operations,4)
    expression = str(random_numbers[0])
    for i in range(1, len(random_numbers)):
        expression += random_operations[i-1] + str(random_numbers[i])
    return expression

def generate_neighbor(expression,numbers):
    # Generar un vecino aleatorio modificando la expresión actual
    # Permutamos dos operadores y cambiamos una cifra por otra no
presente en la expresion
    expr_list = list(expression)
    index = random.sample([1,3,5,7], 2)
    expr_list[index[0]], expr_list[index[1]] = expr_list[index[1]],
expr_list[index[0]]
    index_num = random.choice([0, 2, 4, 6 , 8])
    expr_list[index_num] = str(random.choice([ x for x in numbers if
str(x) not in expression]))
    return ''.join(expr_list)

def bajar_temp(T):
    # Mutiplicamos la temperatura por el factor de enfriamiento
    return T*0.9



def simulated_annealing(target):
```

```
    """Encuentra una combinación de números y operaciones que resulte
en el número objetivo usando Recocido Simulado"""
    operations = ['+', '-', '*', '/']
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]



    current_state = generate_initial_state(operations, numbers)
    current_value = eval(current_state)
    current_heuristic = abs(current_value - target)

    T = 1.0  # Temperatura inicial
    T_min = 0.000000001  # Temperatura mínima
    iter = 1 # contar el número de iteraciones

    while T > T_min:
        for _ in range(100):  # Número de iteraciones por temperatura
            neighbor_state = generate_neighbor(current_state, numbers)
            neighbor_value = eval(neighbor_state)
            neighbor_heuristic = abs(neighbor_value - target)
            iter += 1

            if neighbor_heuristic == 0:
                return neighbor_state, iter

            delta_heuristic = neighbor_heuristic - current_heuristic
            if delta_heuristic < 0 or math.exp(-delta_heuristic / T) >
random.random():
                current_state = neighbor_state
                current_value = neighbor_value
                current_heuristic = neighbor_heuristic

        T = bajar_temp(T)

    return None, iter
```

Como he comentado antes, otra forma de optimizar el código de uana forma más directa seria evitar todas las posibles permutaciones, evitando expresiones equivalentes, y también excluir las soluciones donde el operador de división nos devuelve un valor decimal. esto reduciría de una forma muy considerable el número total de operaciones.

(*)Calcula la complejidad del algoritmo

Para los valores introducidos de temperatura, temperatura mínima,... el número final de iteraciones que realiza es de 19.701, igual que antes habría que mutiplicar esto por las operaciones, que en este caso son 3 asignaciones (ya que no construye la expresión, si no que la modifica) y las 4 operaciones del eval. esto nos da 19.701 x 7 = 137.907 más la primera asignacion y el primer calculo.

Hay una diferencia bastante grande entre el coste de este algoritmo en comparación con la fuerza bruta.

Respuesta

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

Respuesta

En el apartado anterior está la funcion generate_initial_state, que rea una solución aleatoria para este problema

Aplica el algoritmo al juego de datos generado

Respuesta

Esto está presente en el algoritmo del recocido simulado.

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

Respuesta

No he usado ninguna referencia externa más que los apuntes de clase y los códigos de las entregas AG.

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

Otra buena solución para este problema sería por algoritmos geneticos, creo que aumentar el tamaño de los números de entrada, o el hacer modificaciones del tamaño de las expresiones o aumentar las posibles soluciones a no solo enteros pueden empeaorar mucho el funcionamiento de la fuerza bruta. Optaría por algoritmos heurísticos aunque disminuya la precisión, ya que con este tipo de algoritmos que se van acercando a la solución y es bastante sencillo que acabe dando una solución buena en pocas iteraciones.

Espero haya resultado interesante, no he tenido mucho tiempo para realizar la entrega por el trabajo, pero me he intentado esforzar y no dar una solución determinista, evitar la ramificación y poda que es la solución más intuitiva para este tipo de problemas de busqueda, y analizar todo bien para que no quede un trabajo simple.

Un saludo profe!