

Algoritmos de optimización - Seminario

Nombre y Apellidos: Grevin Alonso Marin Umaña

https://github.com/GreivinMarin/VIU_Algo

Nota: es el mismo repositorio para el proyecto y las actividades guiadas

Problema: Combinar cifras y operaciones

- 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 ?
- Nota: Es posible usar la función de python "eval" para evaluar una expresión:

(*) La respuesta es obligatoria

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

Respuesta

¿Cuántas formas hay de combinar números y operadores sin importar si se repiten o no, o si están alternados correctamente?

Supuestos Mínimos

- Queremos formar una expresión con 5 números y 4 operadores.
- Elegimos números del 1 al 9, con repetición permitida.
- Elegimos operadores entre + - * /, también con repetición permitida.

Entonces:

- Cantidad de formas de escoger 5 números (con repetición): **$9^5 = 59,049$**
- Cantidad de formas de escoger 4 operadores (con repetición): **$4^4 = 256$**
- Total sin restricciones: **$9^5 = 59,049 * 4^4 = 256 = 15,119,744$**

(*) ¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones?.

Respuesta

Restricciones

- Usar 5 números distintos tomados del 1 al 9 (sin repetir)
- Eso es una permutación de 5 elementos de un conjunto de 9:
 $P(9,5)=9\times8\times7\times6\times5=15,120$
- Usar los 4 operadores + - * /, sin repetir y en cualquier orden: $4!=24$
- Solo se permiten expresiones estrictamente alternadas, y cada número/operador solo se usa una vez por expresión (ya cubierto arriba).
- total: **15,120 (permutaciones de números) * 24 (permutaciones de operadores) = 362,880 expresiones posibles**

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

itertools.permutations

No es una estructura como tal, sin embargo para la versión inicial pensé en utilizar listas puras pero usando recursividad para generar las permutaciones, sin embargo investigando mejores opciones he encontrado que usando itertools.permutations es más eficiente, ya que es un generador, consume menos memoria ya que no guarda la información y permite recorrer millones de combinaciones de manera eficiente

tuplas

En el caso específico de string devuelto por las permutaciones se ha preferido utilizar tuplas en lugar de listas debido a que las tuplas son inmutables y ordenadas lo cual hace mas eficiente iterarlas cuando no se necesita modificar el contenido.

Según el modelo para el espacio de soluciones

(*)¿Cual es la función objetivo?

Respuesta

Evaluar todas las expresiones válidas (usando 5 dígitos del 1 al 9 y 4 operadores distintos alternados), y maximizar y minimizar el resultado entero obtenido. $f(\text{números}, \text{operadores}) = \text{resultado entero de la expresión evaluada}$

Sujeta a:

- Los 5 números son distintos, seleccionados del conjunto {1, ..., 9}.
- Los 4 operadores son una permutación de {+, -, *, /}.
- La expresión alterna número-operador-número-operador-...-número.
- El resultado debe ser un número entero.

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

Respuesta

Este es un problema de optimización que incluye tanto maximización como minimización. La función objetivo es el resultado entero de evaluar cada expresión válida formada por 5 dígitos distintos (del 1 al 9) y 4 operadores distintos (+ - * /) colocados alternadamente.

Se busca

- Maximizar ese resultado para encontrar el valor más alto posible.
- Minimizar ese resultado para encontrar el valor más bajo posible.

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

Se desarrolla una versión inicial del algoritmo que resuelve el problema por fuerza bruta.

In [3]:

```
"""
*****
Código del algoritmo de fuerza bruta
*****
import itertools
import time

# Iniciar cronómetro
inicio = time.time()

# Conjunto de cifras del 1 al 9 como strings
digitos = [str(i) for i in range(1, 10)]

# Conjunto de operadores posibles
operadores = ['+', '-', '*', '/']

# Conjunto para almacenar los resultados enteros únicos
resultados_enteros = set()

total_exp = 0 # contador de expresiones evaluadas

# Generar todas las permutaciones posibles de 5 dígitos sin repetir
for numeros in itertools.permutations(digitos, 5):

    # Generar todas las permutaciones posibles de 4 operadores sin repetir
    for ops in itertools.permutations(operadores, 4):
```

```

# Construir la expresión alternando número-operador
expresion = ''
for i in range(4):
    expresion += numeros[i] + ops[i]
expresion += numeros[4] # último número

try:
    resultado = eval(expresion)
    if resultado == int(resultado): # si es entero
        resultados_enteros.add(int(resultado))
except ZeroDivisionError:
    continue # ignorar divisiones por cero
except Exception:
    continue # capturar otros errores inesperados

total_exp += 1

# Finalizar cronómetro y calcular duración
fin = time.time()
duracion = fin - inicio

""" ****
De acá hacia abajo el código para la búsqueda
*****"""

# Mostrar resultados
print(f"Duración total: {duracion:.2f} segundos (~{duracion/60:.2f} minutos)")
print(f"Total de expresiones evaluadas: {total_exp}")
print(f"Cantidad de resultados enteros únicos: {len(resultados_enteros)}")

minimo = min(resultados_enteros)
maximo = max(resultados_enteros)

print(f"Valor mínimo entero encontrado: {minimo}")
print(f"Valor máximo entero encontrado: {maximo}")

# Revisar si hay huecos
faltantes = [i for i in range(minimo, maximo + 1) if i not in resultados_enteros]
if faltantes:
    print("Valores enteros faltantes entre mínimo y máximo:")
    print(faltantes)
else:
    print("No hay huecos: todos los enteros entre mínimo y máximo están presentes.")

```

Duración total: 143.38 segundos (~2.39 minutos)
 Total de expresiones evaluadas: 362880
 Cantidad de resultados enteros únicos: 147
 Valor mínimo entero encontrado: -69
 Valor máximo entero encontrado: 77
 No hay huecos: todos los enteros entre mínimo y máximo están presentes.

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

Respuesta

En el primer algoritmo que se realiza por fuerza bruta, observamos que el valor mínimo encontrado fue: -69 y el máximo fue: 77, además encontramos que esta versión del algoritmo logra extraer todos los valores entre ese mínimo y máximo. En las siguientes 2 versiones donde se optimiza el algoritmo a travez del uso de la librería Fractions y backtracking podemos observar que el mínimo encontrado es: -68 y el máximo encontrado fue: 133, acá cambia un poco el rango y además no es capaz de encontrar algunos de los valores entre el mínimo y el máximo

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

Respuesta

En la primera versión se logra encontrar todos los valores entre el mínimo y el máximo, sin embargo en la segunda versión del algoritmo donde utilizamos la librería Fractions no se logra encontrar algunos de ellos, esto se achaca a que el uso de fraction() en lugar del uso de eval() ya que este último podría estar haciendo redondeos de resultados que realmente no son enteros.

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

Ok, yo considero que la parte principal y la que lleva más trabajo de procesamiento computacional es la parte de las permutaciones, si bien es cierto luego de generar y evaluar las permutaciones se hace un proceso de revisión sobre esos resultados, el cual sería de complejidad lineal $O(n)$, me parece que el algoritmo debe ser considerado de Orden Factorial debido a la parte de las permutaciones. La formula sería: $P(n,k) = n! / (n-k)!$. Si usaramos todos los dígitos se tendría que evaluar: $P(9,9) = 9! = 362,880$ Así que según la teoría de este curso haciendo referencia al capítulo 1 apartado 4.2 (Ordenes de complejidad) este algoritmo de fuerza bruta es de

Orden Factorial $O(n!)$

Este tipo de complejidad crece más rápido que los órdenes exponenciales, haciendolo prácticamente inviable cuando tenemos valores de n mayores a 9 o 10

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

Respuesta

Mejora al algoritmo de fuerza bruta

Ok, acá es importante registrar los diferentes intentos y las situaciones encontradas en dichos intentos de mejora. **IMPORTANTE:** Las diferentes versiones del algoritmo se van a agregar en una carpeta llamada "VersionesAlgoritmo" dentro del repositorio, para no hacer este documento muy grande agregando el código de todas las versiones y cambios

Filtrar las operaciones que tengan division por cero

A la versión inicial del algoritmo de fuerza bruta se le ha agregado un try/catch para controlar el error en caso de que alguna operación de error de división por cero, en algún momento pensé que en una gran mayoría de corridas o de las evaluaciones de las expresiones se iba a estar disparando este error, sin embargo me he llevado las sorpresa de que no es así. Primero dado que al estar considerando solamente los números del 1 al 9 pues no hay posibilidad que alguna de las expresiones generadas quede con una división por cero. por lo que el agregar una restricción o filtro if "/0" in expr no sirvió de nada. Segundo, pensé que al tratarse de una generación de más de 362 mil de expresiones la cantidad de resultados, al evaluar expresiones internas, que iban a generar una división por cero sería mayor, pero he corrido el algoritmo en múltiples ocasiones y hasta el momento no se ha dado ninguna. Apesar de esto pues he agregado la versión del código con estas modificaciones para evidenciar este primer intento fallido de optimización del algoritmo.

Uso de método Fractions.fraction() en lugar de eval()

Investigando un poco he visto que de la librería Fractions el método Fraction es más eficiente que el método eval() para determinar enteros, recordando que solo debemos considerar números enteros en los resultados de las permutaciones, podemos notar que efectivamente hubo una gran mejoría en el tiempo de ejecución, de 3.4 minutos a solo 5.30 segundos, ahora bien tenemos resultados diferentes, sin embargo esto se justifica, incluso diría que el resultado con Fraction es más eficiente desde el punto de vista matemático ya que usando eval podríamos obtener valores que por redondeo no son enteros, por ejemplo 4.0000000001 o 3.999999999 podría terminar redondeándose a 4 y ser retornado como parte de los resultados, con Fraction se están usando fracciones exactas lo que evita el asunto del redondeo. y al final se considera solo si el denominador es = 1 (un entero exacto). con este cambio tuvimos mejora de tiempo aún que se determina que si queda "huecos" o valores no obtenidos entre el mínimo y máximo

Evaluación parcial con backtracking

Este enfoque me pareció interesante ya que podemos segmentar la expresión completa, verificar resultados parciales y en caso de encontrar un resultado parcial que nos afecte el resto de la expresión simplemente se descarta y no se continua con evaluación de la

expresión completa, es decir no se continúa con esa rama de evaluación y se regresa para continuar con la siguiente rama. En este caso al menos en mi equipo local se ha logrado una mejora en el tiempo del procesamiento, de 5.3 segundos a 3.4 segundos.

Poda Heurística

En esta versión se hace una poda heurística, simplemente validamos las operaciones de multiplicación y división, en el caso de la multiplicación podamos las multiplicaciones por el número cero o por el número 1, que al final no tiene tanto sentido evaluarlas, para el caso de las divisiones estamos podando cualquier expresión que divida por cero o cuyo residuo sea diferente de cero. Acá hemos encontrado que obtuvimos una mejoría en el tiempo de procesamiento de 3.4 segundos a solo 3 segundos, sin embargo si se vio impactado el resultado ahora tenemos faltantes de al menos 10 valores entre el mínimo y el máximo. En este caso creo que debemos evaluar lo que se quiere lograr con el algoritmo, si el objetivo es encontrar todos los números enteros posibles creo que esta poda es muy agresiva y no nos va a funcionar adecuadamente, ahora si lo que buscamos es una mejoría en el rendimiento podemos observar que hubo una mejora de alrededor de un 22% del tiempo, lo cual no es nada despreciable y podemos considerar que estamos obteniendo un subconjunto representativo del espacio de soluciones. Para efectos de este proyecto descartaré esta tercera versión me quedaré con la versión 2.

```
In [6]: import itertools
import time
from fractions import Fraction

# Iniciar cronómetro
inicio = time.time()

# Lista de todos los dígitos disponibles
digitos = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Lista de operadores disponibles
operadores = ['+', '-', '*', '/']

# Conjunto para almacenar resultados enteros únicos
resultados_enteros = set()

# Contador de expresiones evaluadas
total_exp = 0

def aplicar_operacion(acumulado: Fraction, operador: str, numero: int):
    num_frac = Fraction(numero)
    if operador == '+':
        return acumulado + num_frac
    elif operador == '-':
        return acumulado - num_frac
    elif operador == '*':
        return acumulado * num_frac
    elif operador == '/':
        if num_frac == 0:
```

```

        raise ZeroDivisionError
    return acumulado / num_frac

# Función recursiva para backtracking con evaluación parcial
def backtrack(numeros, operadores_usados, pos, resultado_parcial):
    global total_exp

    if pos == 5:
        # Ya hay 5 números y 4 operadores → resultado final
        if resultado_parcial.denominator == 1:
            resultados_enteros.add(int(resultado_parcial))
        total_exp += 1
        return

    for i, op in enumerate(['+', '-', '*', '/']):
        if op in operadores_usados:
            continue # no repetir operadores

        try:
            nuevo_resultado = aplicar_operacion(resultado_parcial, op, numeros[pos])
            backtrack(numeros, operadores_usados + [op], pos + 1, nuevo_resultado)
        except ZeroDivisionError:
            continue
        except Exception:
            continue

# Generar todas las permutaciones posibles de 5 números distintos del 1 al 9
for perm in itertools.permutations(digitos, 5):
    # Empezamos desde el primer número
    resultado_inicial = Fraction(perm[0])
    backtrack(perm, [], 1, resultado_inicial)

# Finalizar cronómetro y calcular duración
fin = time.time()
duración = fin - inicio

```

"""\n*****\n

De acá hacia abajo el código para la búsqueda

*****\n

```

print(f"Duración total: {duración:.2f} segundos (~{duración/60:.2f} minutos)")
print(f"Total de expresiones evaluadas: {total_exp}")
print(f"Cantidad de resultados enteros únicos: {len(resultados_enteros)}")

mínimo = min(resultados_enteros)
máximo = max(resultados_enteros)

print(f"Valor mínimo entero encontrado: {mínimo}")
print(f"Valor máximo entero encontrado: {máximo}")

# Verificar si hay huecos

```

```

faltantes = [i for i in range(minimo, maximo + 1) if i not in resultados_enteros]
if faltantes:
    print(f"Hay {len(faltantes)} valores faltantes entre mínimo y máximo:")
    print(faltantes)
else:
    print("No hay huecos: todos los enteros entre mínimo y máximo están presentes.")

```

Duración total: 3.52 segundos (~0.06 minutos)

Total de expresiones evaluadas: 362880

Cantidad de resultados enteros únicos: 198

Valor mínimo entero encontrado: -68

Valor máximo entero encontrado: 133

Hay 4 valores faltantes entre mínimo y máximo:

[107, 120, 127, 128]

(*)Calcula la complejidad del algoritmo

Respuesta

En la versión optimizada el algoritmo no genera todas las expresiones de entrada sino que las construye recursivamente y el número de llamadas depende de la cantidad de pasos válidos desde cada nodo. Al usar $n = 9$ y tomar 5 dígitos se tienen $O(n^5)$ caminos posibles. Aunque el total sigan siendo 262,880 combinaciones el enfoque reduce significativamente el costo interno de cada evaluación, el orden de complejidad es:

$O(n^5)$ Orden polinomial

Tabla comparativa entre todas las versiones trabajadas en este proyecto

Versión	Estrategia	Complejidad asintótica	Comentario
Fuerza bruta con <code>eval()</code>	Permutaciones completas + evaluación con <code>eval()</code>	$O(n!)$ (orden factorial)	Evalúa todas las combinaciones posibles sin filtrado, con imprecisión por flotantes
Fuerza bruta con <code>Fraction</code>	Permutaciones completas + evaluación con <code>fractions.Fraction</code>	$O(n!)$ (orden factorial)	Mismo espacio de búsqueda que la fuerza bruta, pero con mayor precisión numérica
Backtracking sin poda	Construcción recursiva paso a paso + evaluación parcial con <code>Fraction</code>	$O(n^5)$ (orden polinomial)	Mismo número total de expresiones, pero más rápido al evitar construcción de strings y <code>eval()</code>
Backtracking con poda heurística	Recursión + poda anticipada según reglas numéricas	$O(n^5)$ (igual teóricamente)	Reduce el tiempo de ejecución al evitar combinaciones inútiles, pero puede perder exactitud

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

Respuesta

Abajo tenemos un código el cual tiene el mismo código que el algoritmo principal simplemente no se añadió el método del backtracking dado que vamos a ejecutarlo con una entrada aleatoria y pequeña de datos, en la celda hay 3 ejecuciones con 3 diferentes sets de datos, 3 conjuntos de 5 dígitos junto con los 4 operadores, y en la respuesta vemos que efectivamente nos demuestra si el valor obtenido es entero o no, que es la misma funcionalidad del algoritmo completo.

```
In [11]: import random
from fractions import Fraction

# Función para aplicar una operación entre fracciones
def aplicar_operacion(acumulado, operador, numero):
    num_frac = Fraction(numero)
    if operador == '+':
        return acumulado + num_frac
    elif operador == '-':
        return acumulado - num_frac
    elif operador == '*':
        return acumulado * num_frac
    elif operador == '/':
        if num_frac == 0:
            raise ZeroDivisionError
        return acumulado / num_frac

# Función para evaluar una expresión dada una lista de números y operadores
def evaluar_expresion(digitos, operadores, caso_id=1):
    print(f"\nCaso {caso_id}:")
    print(f"Números: {digitos}")
    print(f"Operadores: {operadores}")

    try:
        resultado = Fraction(digitos[0])
        expresion = str(digitos[0])

        for i in range(4):
            op = operadores[i]
            num = digitos[i + 1]
            resultado = aplicar_operacion(resultado, op, num)
            expresion += f" {op} {num}"

        print(f"Expresión: {expresion}")
        print(f"Resultado fraccional exacto: {resultado}")
        if resultado.denominator == 1:
            print(f"Resultado entero: {int(resultado)}")
        else:
            print("No es un entero")

    except ZeroDivisionError:
```

```

        print("¡División por cero detectada!")
    except Exception as e:
        print(f"Error al evaluar la expresión: {e}")

# Generar y evaluar 3 conjuntos aleatorios
for i in range(1, 4):
    digitos = random.sample(range(1, 10), 5) # 5 números distintos
    operadores = random.sample(['+', '-', '*', '/'], 4) # 4 operadores distintos
    evaluar_expresion(digitos, operadores, caso_id=i)

```

Caso 1:

Números: [3, 7, 8, 6, 4]
 Operadores: ['+', '-', '/', '*']
 Expresión: $3 + 7 - 8 / 6 * 4$
 Resultado fraccional exacto: 4/3
 No es un entero

Caso 2:

Números: [8, 1, 4, 7, 2]
 Operadores: ['/', '*', '+', '-']
 Expresión: $8 / 1 * 4 + 7 - 2$
 Resultado fraccional exacto: 37
 Resultado entero: 37

Caso 3:

Números: [4, 2, 6, 9, 7]
 Operadores: ['/', '*', '+', '-']
 Expresión: $4 / 2 * 6 + 9 - 7$
 Resultado fraccional exacto: 14
 Resultado entero: 14

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

Respuesta

Referencias

- Python Software Foundation. (2024). *fractions — Rational numbers*. En *The Python Standard Library*.
<https://docs.python.org/3/library/fractions.html>
- GeeksforGeeks. (2023, abril 18). *Backtracking – Introduction*.
<https://www.geeksforgeeks.org/backtracking-introduction/>
- Real Python. (2023, junio 7). *Working with fractions in Python*.
<https://realpython.com/python-fractions/>

Describe brevemente las líneas 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

Me parece que una posible forma de seguir trabajando este problema podría ser analizando en qué pasaría si se aumenta el número de dígitos usados, por ejemplo 6 o 7 en lugar de 5. Esto nos daría un espacio de búsqueda bastante más grande, pienso que así se podría intentar algún estilo de poda más agresiva, ya vimos que aplicando una poda heurística no tan agresiva funcionó para darnos un subconjunto que podría ser funcional dependiendo de lo que estemos buscando. También se podría pensar en una búsqueda herística o incluso algún tipo de programación genética que al parecer están dando muy buenos resultados. Otra línea interesante podría ser el permitir repetir operadores o incluso números, lo cual abriría muchísimas más combinaciones posibles y cambiaría completamente el comportamiento del algoritmo. En general el problema tiene mucho potencial para escalar o adaptarse a distintas versiones.