TrabPractico_Augusto_IbanezGarcia

February 29, 2024

1 Algoritmos de optimización - Seminario

Nombre y Apellidos: AUGUSTO JAVIER IBAÑEZ GARCIA
GitHub Url: https://github.com/cibergus/VIU-AlgOptimizacion
Problema: 3) Combinar cifras y operaciones

DESCRIPCIÓN 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?

NOTAS:
- (*) La respuesta es obligatoria
- ¡Importante! Además de resolver el problema será necesario responder algunas preguntas relacionadas con: Complejidad / Justificar la estructura de datos elegida / Generar y probar con diferentes juegos de datos de entrada
- Es posible usar la función de python “eval” para evaluar una expresión:

[1]:
```
expresion = "4-2+6/3*1"
print(eval(expresion))
```

```
4.0
```

[2]:
```
# INSTALLS + IMPORTS

import random
from math import factorial
from itertools import permutations
```

2 Combinar cifras y operaciones

2.1 (*) Posibilidades

¿Cuantas posibilidades hay sin tener en cuenta las restricciones?
¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

2.1.1 Respuesta (Obligatoria):

[3]:
```python
# Calcular permutaciones de n elementos tomados de r en r
def calcular_permutaciones(n, r):
    return factorial(n) / factorial(n - r)

# Calcular el total de posibilidades con las restricciones especificadas
def calcular_posibilidades_con_restricciones():
    permutaciones_cifras = calcular_permutaciones(9, 5)
    permutaciones_signos = factorial(4)
    total_posibilidades = permutaciones_cifras * permutaciones_signos
    return total_posibilidades

# Llamadas
total_posibilidades_sin_restricciones = factorial(9) * factorial(4)
total_posibilidades_restricciones = calcular_posibilidades_con_restricciones()
print(f"Sin restricciones:\t{total_posibilidades_sin_restricciones:>10}")
print(f"Con restricciones:\t{int(total_posibilidades_restricciones):>10}")
```

```
Sin restricciones:          8709120
Con restricciones:           362880
```

Sin Restricciones: - Al considerar el problema sin ninguna restricción, se tienen 9! formas de ordenar las cifras del 1 al 9 y 4! maneras de ordenar los signos de operación. Esto resulta en un total de 9! x 4! = 8.709.120 combinaciones posibles, ya que cualquier cifra o signo puede ocupar cualquier posición en la secuencia.

Con Restricciones: - Bajo las restricciones especificadas, el problema se concentra en formar secuencias que alternen entre las 9 cifras disponibles y los 4 signos de operación, siguiendo el patrón cifra-operación-cifra-operación, y así sucesivamente, comenzando y terminando con una cifra. Específicamente, se utilizan 5 de las 9 cifras y todos los 4 signos de operación exactamente una vez, lo que nos lleva a calcular las permutaciones de 9 cifras tomadas de 5 en 5 (9P5) multiplicadas por las permutaciones de los 4 signos (4!). Esto simplifica el cálculo inicial a 9P5 x 4!, resultando en un total de 362.880 combinaciones posibles que cumplen con las restricciones del problema.

- La configuración permitida implica usar exactamente 5 de las 9 cifras y los 4 signos de operación en una secuencia específica:
  cifra_1 - operación_1 - cifra_2 - operación_2 - cifra_3 - operación_3 - cifra_4 - operación_4 - cifra_5
  - Para las cifras, de las 9 disponibles, seleccionamos 5, lo cual implica calcular las combinaciones de 9 cifras tomadas de 5 en 5, y luego permutar estas 5 cifras seleccionadas

dentro de su propio conjunto. Esto se debe a que, aunque solamente se seleccionan 5 cifras, el orden en que se colocan estas cifras importa.

  - Para los signos de operación, dado que se usan todos los 4 signos disponibles sin repetición, y el orden en que se usan importa (ya que afecta el resultado de la expresión matemática), se calcula como una permutación de los 4 signos.

- Cálculo de las posibilidades con las restricciones especificadas
  - Permutaciones de 9 cifras tomadas de 5 en 5
  - permutaciones_cifras = factorial(9) / factorial(9 - 5)
  - Permutaciones de los 4 signos de operación
  - permutaciones_signos = factorial(4)
  - Cálculo del total de posibilidades:
    * total_posibilidades_restricciones_especificadas = permutaciones_cifras * permutaciones_signos = 362.880

2.2 (*) Modelo para el espacio de soluciones

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

2.2.1 Respuesta (Obligatoria):

Para modelar el espacio de soluciones del problema planteado, que implica generar todas las secuencias posibles de 5 cifras y 4 signos de operación bajo las restricciones específicas de no repetición y alternancia, la estructura de datos ideal debe soportar una generación eficiente y flexible de permutaciones, así como una manipulación fácil de sus elementos. Considerando estas necesidades, el **árbol de decisión** se presenta como la estructura de datos más adecuada por varias razones:

2.2.2 Árbol de Decisión

- Generación Dinámica de Soluciones. Un árbol de decisión permite generar dinámicamente todas las combinaciones posibles de cifras y signos de operación, expandiendo un nodo para cada elección posible en cada paso del camino. Esto es particularmente útil dado que debemos explorar todas las combinaciones posibles que cumplen con las restricciones de alternancia y no repetición.
- Fácil Implementación de Restricciones. Las restricciones de no repetición y la alternancia entre cifras y operaciones pueden implementarse naturalmente en un árbol de decisión. Al expandir los nodos, simplemente evitamos agregar ramas para las cifras o signos de operación ya utilizados en la secuencia actual.
- Backtracking Eficaz. La búsqueda en un árbol de decisión se puede optimizar con técnicas de backtracking, donde el algoritmo puede "volver atrás" si una rama no lleva a una solución válida, minimizando así el tiempo de búsqueda al evitar explorar secuencias inválidas.
- Flexibilidad y Escalabilidad. Los árboles de decisión ofrecen una gran flexibilidad para adaptarse a cambios en las restricciones o en la formulación del problema. Si las restricciones

se modifican, ajustar la generación del árbol para acomodar estos cambios es relativamente sencillo.
- Visualización Intuitiva del Espacio de Soluciones. La estructura de un árbol de decisión proporciona una representación visual intuitiva del espacio de soluciones, donde cada camino desde la raíz hasta una hoja representa una solución potencial. Esto puede ser útil para el análisis y la depuración.

2.2.3 Cambio de Estructura de Datos

Inicialmente se consideró utilizar una 'lista o un array' para generar y almacenar las secuencias, aunque estas estructuras son simples y directas para almacenar secuencias de cifras y signos de operación, no ofrecen la misma eficiencia y flexibilidad que un árbol de decisión para implementar restricciones y realizar backtracking. Las listas o arrays requerirían una gestión adicional para evitar repeticiones y garantizar la alternancia, mientras que un árbol de decisión maneja estos aspectos de manera más natural y eficiente. Dado el problema específico y sus restricciones, un árbol de decisión no solo facilita la generación de soluciones válidas sino que también optimiza el proceso de exploración del espacio de soluciones, haciendo de esta estructura de datos la opción más adecuada.

2.3 (*) Según el modelo para el espacio de soluciones

(x) ¿Cual es la función objetivo?

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

2.3.1 Respuestas (Obligatoria):

2.3.2 Función Objetivo

En este caso es generar una expresión matemática que utilice exactamente cinco cifras (sin repetirlas) del 1 al 9 y los cuatro signos de operaciones básicas, de manera que el resultado de evaluar dicha expresión sea igual a un valor específico dado. La función objetivo podría formalizarse como maximizar o minimizar la diferencia entre el valor obtenido de la expresión y el valor objetivo, dependiendo de cómo se plantee el problema.

Sin embargo, el objetivo principal aquí no es típicamente maximizar o minimizar en el sentido tradicional de buscar el valor máximo o mínimo posible; en cambio, es encontrar una o más soluciones válidas que cumplan con la restricción de igualar a un número dado. Esto se alinea más estrechamente con un problema de cumplimiento de restricciones que con un problema de optimización clásico.

2.3.3 Maximización o Minimización

Dado lo anterior, el problema no se clasifica estrictamente como de maximización o minimización. El problema presentado es uno de **búsqueda** (y cumplimiento de restricciones) más que de optimización en el sentido tradicional. La función objetivo es encontrar combinaciones de cifras y operaciones que resulten en un valor específico, sin una directriz de maximización o minimización aplicable en este contexto. En su lugar, el desafío se centra en:

- Cumplimiento de Restricciones identificando si existe una combinación de cifras y operaciones que satisfaga la condición de igualar al valor objetivo.
- Exploración de Soluciones encontrando todas las combinaciones posibles que cumplan con la condición establecida, dentro del espacio de soluciones definido por las restricciones del problema.

2.4 Diseña un algoritmo para resolver el problema por fuerza bruta

2.4.1 Respuesta (Opcional):

Para resolver el problema mediante un enfoque de fuerza bruta, el algoritmo intentará todas las combinaciones posibles de cinco cifras del 1 al 9 y los cuatro signos de operaciones matemáticas (+, -, *, /), respetando las restricciones de no repetición de cifras y alternancia entre cifras y operaciones.

Algoritmo Fuerza Bruta

- Paso 1: Generar Todas las Permutaciones de Cifras y Operaciones
  - Generar todas las permutaciones posibles de las cifras del 1 al 9, seleccionando 5 de ellas para cada permutación.
  - Generar todas las permutaciones posibles de los cuatro signos de operaciones.
- Paso 2: Combinar Cifras y Operaciones
  - Para cada permutación de 5 cifras y cada permutación de 4 operaciones, intercalarlas para formar una expresión matemática válida siguiendo el patrón: cifra-operación-cifra-operación-cifra-operación-cifra-operación-cifra.
- Paso 3: Evaluar Cada Expresión
  - Utilizar la función eval() de Python para evaluar cada expresión generada en el paso anterior.
  - Verificar si el resultado de la evaluación coincide con el valor objetivo dado.
- Paso 4: Almacenar y Devolver Soluciones Válidas
  - Si una expresión evalúa al valor objetivo, almacenar esta expresión como una solución válida.
  - Continuar hasta que todas las posibles combinaciones hayan sido probadas.
  - Devolver todas las soluciones válidas encontradas.

Notas:
- Este algoritmo es intensivo en términos computacionales, especialmente debido a la gran cantidad de permutaciones a evaluar.
- Se maneja la excepción ZeroDivisionError para evitar errores durante la evaluación de expresiones que incluyan divisiones por cero.

[4]:
```python
# Definir las cifras y los signos de operación + cifras del 1 al 9 como cadenas
cifras = [str(i) for i in range(1, 10)]
operaciones = ['+', '-', '*', '/']
```

```python
def generar_expresiones():
    soluciones = []
    # Permutaciones de 5 cifras
    for perm_cifras in permutations(cifras, 5):
        # Permutaciones de 4 operaciones
        for perm_operaciones in permutations(operaciones):
            expresion = ''.join(sum(zip(perm_cifras, perm_operaciones + ('',)),␣
 ↪()))
            try:
                if eval(expresion) == valor_objetivo:
                    soluciones.append(expresion)
            # Ignorar la división por cero
            except ZeroDivisionError:
                pass
    return soluciones

# Valor objetivo a obtener
valor_objetivo = 4
soluciones = generar_expresiones()
print(f"Número de soluciones: \n{len(soluciones)}\n")
#print(f"Soluciones encontradas: \n{soluciones}")

primeras_20 = ' '.join(soluciones[:20])
print("Primeras 20 soluciones:\n", primeras_20, "\n")

# Verificar si hay más de 40 soluciones para evitar superposición en la␣
 ↪impresión
if len(soluciones) > 40:
    ultimas_20 = ' '.join(soluciones[-20:])
    print("Últimas 20 soluciones:\n", ultimas_20)
else:
    print("Menos de 40 soluciones en total, algunas soluciones se muestran en␣
 ↪ambos conjuntos.")
```

```
Número de soluciones:
2112

Primeras 20 soluciones:
 1-2*3/6+4 1-2/3*6+7 1/2*4-3+5 1/2*4+5-3 1/2*4-5+7 1*2+4-6/3 1/2*4-6+8 1/2*4+7-5
1/2*4-7+9 1/2*4+8-6 1-2/4*8+7 1/2*4+9-7 1*2+5-9/3 1-2/6*3+4 1*2-6/3+4 1/2*6-3+4
1-2*6/3+7 1/2*6+4-3 1/2*6-4+5 1/2*6+5-4

Últimas 20 soluciones:
 9+7-8*6/4 9-8+1/2*6 9+8*1/4-7 9-8+1*6/2 9-8*1+6/2 9-8/2*3+7 9-8+2/4*6 9-8+2*6/4
9-8*3/2+7 9-8*3/4+1 9+8/4-1*7 9+8/4*1-7 9-8/4*3+1 9-8/4*6+7 9+8/4-7*1 9-8+6*1/2
9-8+6/2*1 9-8+6*2/4 9-8+6/4*2 9-8*6/4+7
```

2.5 Calcula la complejidad del algoritmo por fuerza bruta

2.5.1 Respuesta (Opcional):

Para calcular la complejidad del algoritmo de fuerza bruta diseñado para combinar cifras y operaciones matemáticas con el fin de obtener un valor objetivo, debemos considerar los siguientes aspectos del algoritmo:

1. **Generación de Permutaciones de Cifras:** Seleccionamos 5 cifras de un conjunto de 9, lo que nos da permutaciones de 9 elementos tomados de 5 en 5. La fórmula para calcular el número de permutaciones es $P(n, r) = \frac{n!}{(n-r)!}$
   donde n es el número total de elementos y r es el número de elementos seleccionados para permutar.
   En este caso, $P(9, 5) = \frac{9!}{(9-5)!} = \frac{9!}{4!}$.
2. **Generación de Permutaciones de Operaciones:** Hay 4 operaciones, y todas ellas deben usarse una vez, lo que nos da permutaciones de 4 elementos tomados todos a la vez, es decir, 4!.
3. **Combinación de Cifras y Operaciones:** Para cada combinación de cifras, intercalamos las operaciones de todas las maneras posibles. Sin embargo, dado que las operaciones ya están permutadas, esto simplemente se reduce a calcular la expresión basada en cada permutación de cifras y operaciones.
4. **Evaluación de Expresiones:** Para cada combinación única de cifras y operaciones, evaluamos la expresión matemática resultante.

2.5.2 Cálculo de la Complejidad

- La generación de permutaciones de cifras tiene una complejidad de $O(\frac{9!}{4!})$.
- La generación de permutaciones de operaciones tiene una complejidad de $O(4!)$.

La complejidad total del algoritmo es el producto de estas dos complejidades, ya que para cada permutación de cifras, probamos todas las permutaciones de operaciones: $O(\frac{9!}{4!}) \times O(4!) = O(9!)$

2.5.3 Complejidad Final

La complejidad final del algoritmo es $O(9!)$, reflejando el número total de evaluaciones de expresiones que se realizan. Este cálculo asume que la evaluación de cada expresión (usando `eval` en Python, por ejemplo) tiene una complejidad constante, lo cual puede no ser cierto en todos los casos dependiendo de la complejidad de la implementación de la función de evaluación.

2.5.4 Conclusión

El algoritmo de fuerza bruta es, por tanto, exponencial en términos de complejidad, lo que lo hace impracticable para valores de n mucho mayores o para problemas similares con espacios de solución significativamente más grandes. La eficiencia puede ser aceptable para este problema específico debido al limitado número de cifras y operaciones, pero en general, esta aproximación es computacionalmente costosa.

2.6 (*) Algoritmo Mejorado: Backtracking con Poda

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

2.6.1 Respuesta (Obligatoria):

[5]:
```
# Evaluamos la expresión construida + Alternar entre añadir una cifra o una operación
# Intentar añadir la cifra y continuar la búsqueda + la operación y continuamos la búsqueda

def backtrack(expresion=[], cifras_usadas=set(), operaciones_usadas=set(), depth=0, objetivo=0, current_eval=0):
    # La secuencia completa tiene 9 elementos: 5 dígitos y 4 operaciones
    if depth == 9:
        if eval(''.join(expresion)) == objetivo:
            print(f"Solución encontrada: {''.join(expresion)} = {objetivo}")
            return True
        return False

    # Es turno de añadir una cifra
    if depth % 2 == 0:
        for cifra in range(1, 10):
            if cifra not in cifras_usadas:
                new_cifras_usadas = cifras_usadas.copy()
                new_cifras_usadas.add(cifra)
                if backtrack(expresion + [str(cifra)], new_cifras_usadas, operaciones_usadas, depth + 1, objetivo, current_eval):
                    return True
    else:
        for operacion in ['+', '-', '*', '/']:
            if operacion not in operaciones_usadas:
                new_operaciones_usadas = operaciones_usadas.copy()
                new_operaciones_usadas.add(operacion)
                if backtrack(expresion + [operacion], cifras_usadas, new_operaciones_usadas, depth + 1, objetivo, current_eval):
                    return True

    return False


# Ejemplo de uso.
objetivo = 4
backtrack(objetivo=objetivo)
```

```
Solución encontrada: 1+3*8/2-9 = 4
```

[5]: `True`

Para mejorar la complejidad del algoritmo de fuerza bruta en el problema de combinar cifras y operaciones matemáticas para obtener un valor objetivo, podemos adoptar un enfoque más inteligente que reduzca el espacio de búsqueda mediante el uso de programación dinámica o backtracking con poda. Aquí presento un enfoque basado en **backtracking**:

2.6.2 Backtracking con Poda

Idea Principal. El algoritmo de backtracking explora el espacio de soluciones construyendo candidatos a solución uno a uno y descartando rápidamente aquellos que no conducen a una solución válida. La "poda" se refiere a detener prematuramente la exploración de ramas del espacio de soluciones que claramente no cumplirán con el criterio de búsqueda, mejorando así la eficiencia.

- Paso 1: Inicialización de una lista para almacenar la secuencia actual de cifras y operaciones.
- Paso 2: Backtracking
  - Se comineza con la primera cifra y exploran todas las cifras posibles (del 1 al 9) que no se hayan usado aún.
  - Después de seleccionar una cifra, se selecciona una operación entre las cuatro posibles que aún no se haya usado en la secuencia actual.
  - Continuamos alternando entre la selección de cifras y operaciones, aplicando las restricciones de no repetición y de alternancia.
  - En cada paso, calculamos el resultado parcial de la expresión actual. Si en algún momento el resultado parcial hace imposible alcanzar el valor objetivo (por ejemplo, si el valor objetivo es muy pequeño/grande comparado con el resultado parcial), realizamos la poda y la vuelva atrás.
- Paso 3: Verificación de la Solución
  - Si la secuencia completa de cifras y operaciones produce el valor objetivo, almacenamos esta secuencia como una solución válida.
  - Si no, deshacemos el último paso del backtracking y probamos una diferente combinación de cifras/operaciones.
- Paso 4: Continuamos hasta explorar todas las posibilidades repitiendo el proceso hasta que todas las combinaciones posibles hayan sido exploradas o se haya encontrado una solución válida.

2.6.3 Explicación del Código

- La función `backtrack` explora recursivamente todas las combinaciones posibles de cifras y operaciones, alternando entre cifras y operaciones en cada paso.
- Utiliza listas `cifras_usadas` y `operaciones_usadas` para llevar un registro de las cifras y operaciones ya utilizadas en la expresión actual, asegurando así que no haya repeticiones.
- Evalúa la expresión construida cuando se alcanza la longitud deseada (5 cifras y 4 operaciones) y compara el resultado con el valor objetivo.
- Implementa la poda del espacio de búsqueda al detener prematuramente la exploración de ramas que ya no pueden satisfacer el valor objetivo debido a las restricciones del problema.

2.7 (*) Calcula la complejidad del algoritmo

2.7.1 Respuesta (Obligatoria):

2.7.2 Explicación de la Complejidad del Algoritmo de Backtracking

La complejidad de este algoritmo es difícil de determinar de manera precisa sin conocer la profundidad a la que se podrá podar efectivamente el espacio de búsqueda en escenarios típicos. Sin embargo, es claro que este enfoque reduce significativamente el número de combinaciones exploradas en comparación con un método de fuerza bruta puro, debido a la poda de ramas inviables y al hecho de no explorar combinaciones que violen las restricciones de no repetición y alternancia. La eficiencia real dependerá de la cantidad de podas que se puedan aplicar durante la ejecución. La complejidad del algoritmo de backtracking diseñado para encontrar combinaciones de cifras y operaciones que cumplan con un objetivo específico depende de varios factores, incluyendo el número de decisiones posibles en cada paso y el grado en que la poda reduce el espacio de búsqueda. Vamos a desglosar estos factores para entender mejor la complejidad del algoritmo:

- Decisiones en cada paso. En cada paso del backtracking, el algoritmo elige entre una cifra o una operación, dependiendo de la profundidad actual (si es par o impar).
  - Para las cifras (números del 1 al 9), inicialmente hay 9 opciones. Sin embargo, debido a la restricción de no repetición, el número de opciones disminuye en cada paso sucesivo cuando se elige una cifra.
  - Para las operaciones (+, -, *, /), hay 4 opciones sin repetición.
- Profundidad del Árbol de Decisión. El árbol de decisión generado por este algoritmo tiene una profundidad máxima de 9, alternando entre cifras y operaciones (5 cifras y 4 operaciones).
- Poda. La eficacia de la poda (descartar partes del espacio de búsqueda que claramente no llevarán a una solución) depende de cómo se implemente el algoritmo. La poda reduce significativamente el número de ramas del árbol de decisión que se deben explorar.

2.7.3 Cálculo de la Complejidad

- Sin poda, la complejidad sería de $O(9! \times 4!)$, ya que sería el producto de todas las permutaciones posibles de 5 cifras y 4 operaciones.
- Con poda efectiva, la complejidad se reduce. Sin embargo, calcular la complejidad exacta es desafiante porque depende de cuánto se pueda reducir el espacio de búsqueda. La poda puede variar dramáticamente la cantidad de trabajo necesario dependiendo del valor objetivo y de las restricciones específicas aplicadas durante el backtracking.

La complejidad exacta es difícil de expresar en términos de O-grande sin más detalles sobre la función de poda y cómo varía el espacio de búsqueda en diferentes casos. Sin embargo, es seguro decir que la complejidad es significativamente menor que $O(9! \times 4!)$ debido a la poda efectiva del espacio de búsqueda.

2.7.4 Por qué Mejora la Complejidad

- Reducción del Espacio de Búsqueda. A diferencia del enfoque de fuerza bruta que evalúa todas las combinaciones posibles sin considerar su viabilidad, el backtracking con poda descarta rápidamente las ramas del espacio de soluciones que no conducen a una solución, evitando así exploraciones innecesarias.

- Eficiencia en la Evaluación. Al evaluar la viabilidad de la solución parcial en cada paso, el algoritmo evita el cálculo de muchas expresiones matemáticas completas que claramente no satisfarán el criterio de búsqueda.
- Optimización Dinámica. Al reutilizar y ajustar las soluciones parciales durante el proceso de backtracking, el algoritmo se adapta dinámicamente, lo que puede llevar a una solución más rápidamente que el enfoque de fuerza bruta.

2.7.5 Conclusión

Este algoritmo mejora la complejidad respecto al de fuerza bruta al reducir significativamente el número de combinaciones que necesitan ser evaluadas para encontrar una solución. Al aplicar una estrategia de poda, solo se exploran las combinaciones que tienen el potencial de satisfacer el valor objetivo, lo que resulta en una eficiencia considerablemente mayor, especialmente en espacios de solución grandes. Esto lo hace más eficiente, aunque su complejidad exacta puede variar. La poda efectiva es clave para mejorar el rendimiento y hacer que el algoritmo sea práctico para este tipo de problemas.

2.8 Juego datos aleatorio

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

2.8.1 Respuesta (Opcional):

[6]:
```python
# Generamos un valor objetivo aleatorio dentro de un rango razonable [-100, 100]
valor_objetivo = random.randint(-100, 100)
print(f"Valor objetivo aleatorio: {valor_objetivo}")
if not backtrack([], set(), set(), 0, valor_objetivo):
    print(f"No se encontró solución para el valor objetivo {valor_objetivo}")
```

```
Valor objetivo aleatorio: 64
Solución encontrada: 3+7*9-2/1 = 64
```

Para generar un juego de datos de entrada aleatorios adecuado para el problema de encontrar combinaciones de cifras y operaciones que cumplan con un valor objetivo dado, podemos seguir un enfoque estructurado.
Este conjunto de datos incluiría un valor objetivo aleatorio y una selección de cifras y operaciones, aunque el problema original ya define las cifras (1 al 9) y las operaciones (+, -, *, /) de forma fija. La variabilidad entonces proviene principalmente del valor objetivo y de cómo se generan secuencias aleatorias para testear el algoritmo fuera de la búsqueda de soluciones específicas.
Este enfoque de generación de datos de entrada aleatorios y su aplicación práctica en pruebas ayuda a validar la robustez del algoritmo frente a una variedad de condiciones, asegurando que es capaz de manejar diferentes valores objetivos de manera efectiva.

2.8.2 Generación de un Valor Objetivo Aleatorio

Dado que el objetivo es formar una expresión matemática que evalúe a un número específico, el primer paso es generar este valor objetivo de manera aleatoria. Dado que las combinaciones de cifras y operaciones pueden variar ampliamente en sus resultados, es razonable limitar el rango de

valores objetivos para garantizar que el conjunto de pruebas sea manejable y tenga sentido en el contexto del problema.

2.8.3 Generación de Secuencias Aleatorias para Pruebas (Opcional)

Aunque el problema especifica que se deben usar todas las cifras del 1 al 9 sin repetición y los cuatro signos de operación exactamente una vez, se podría considerar generar secuencias aleatorias de cifras y operaciones para pruebas más generales o para explorar variantes del problema. Sin embargo, para el problema tal como se presenta, esta parte no es aplicable directamente.

2.8.4 Uso del Juego de Datos Aleatorio

El valor objetivo aleatorio generado se puede utilizar para probar la capacidad del algoritmo de backtracking para encontrar una secuencia válida de cifras y operaciones que cumpla con este objetivo. Esto proporciona una manera de evaluar la eficacia y eficiencia del algoritmo en diferentes escenarios y contribuye a una comprensión más profunda de su comportamiento en casos variados.

2.9 Enumera las referencias

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

2.9.1 Respuesta (Opcional):

Para la elaboración de las respuestas a este conjunto de preguntas, no se utilizaron referencias externas directas. Las respuestas se basaron en conocimientos generales sobre algoritmos, programación en Python, y principios de matemáticas y computación, incluyendo:

- Conceptos básicos de la teoría de la computación y algoritmos, como permutaciones, combinaciones, backtracking y poda.
- La documentación oficial de Python para detalles específicos sobre funciones y sintaxis del lenguaje.
- Principios matemáticos básicos relacionados con operaciones aritméticas y evaluación de expresiones.

Dado que este trabajo se centró en explicar y desarrollar soluciones algorítmicas a partir de principios básicos de computación y programación, no se hizo referencia directa a materiales o publicaciones específicas. Sin embargo, para profundizar en los temas tratados o para obtener una explicación más detallada sobre temas específicos, los siguientes recursos pueden ser útiles:

- Documentación oficial de Python: https://docs.python.org/3/ - Para referencia sobre la sintaxis de Python y las funciones de sus bibliotecas estándar, como `itertools` y operaciones matemáticas básicas.
- "Introduction to Algorithms" por Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, y Clifford Stein - Un texto fundamental que proporciona una introducción completa a los algoritmos y sus técnicas de diseño, incluyendo la backtracking y la optimización.
- "Python for Data Analysis" por Wes McKinney - Aunque enfocado en análisis de datos, este libro también ofrece una buena introducción a Python y sus capacidades para manipular y evaluar datos, lo que puede ser relevante para la implementación de algoritmos.

- "Grokking Algorithms" por Aditya Bhargava - Un libro accesible para principiantes que explica conceptos algorítmicos complejos en un lenguaje simple, incluyendo backtracking y otras estrategias de búsqueda.

2.10 Avances

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

2.10.1 Respuesta (Opcional):

El estudio del problema de combinar cifras y operaciones para obtener un valor dado puede avanzar en varias direcciones interesantes, tanto en la profundización teórica como en la aplicación práctica. Estas líneas de avance no solo contribuirían a una mejor solución del problema original sino que también abrirían nuevas áreas de investigación y aplicación práctica, enriqueciendo nuestra comprensión de la interacción entre números y operaciones matemáticas. A continuación, describo algunas líneas de avance:

2.10.2 Optimización del Algoritmo

- Heurísticas de Poda Mejoradas. Investigar estrategias de poda más sofisticadas que reduzcan aún más el espacio de búsqueda sin necesidad de explorar todas las posibilidades.
- Paralelización. Dado que la evaluación de diferentes ramas del árbol de decisión es independiente, se podrían explorar técnicas de paralelización para distribuir el trabajo y acelerar la búsqueda de soluciones.

2.10.3 Variaciones del Problema

- Extensión a Más Cifras y Operaciones. Explorar cómo se comporta el algoritmo y qué técnicas son necesarias para manejar conjuntos más grandes de cifras y operaciones, incluyendo potencias, raíces cuadradas, y operadores lógicos.
- Restricciones Adicionales. Añadir nuevas restricciones, como limitar el número de veces que se puede usar una operación, o incluir paréntesis para alterar el orden de las operaciones.

2.10.4 Aplicaciones Prácticas

- Generación de Ejercicios Matemáticos. Utilizar el algoritmo para generar problemas matemáticos para fines educativos, ajustando el nivel de dificultad según el rango de valores objetivo y la complejidad de las operaciones permitidas.
- Análisis de Sensibilidad. Estudiar cómo pequeñas variaciones en las cifras o en el valor objetivo afectan la cantidad y la naturaleza de las soluciones encontradas, lo cual puede tener aplicaciones en la enseñanza de conceptos matemáticos o en la planificación financiera.

2.10.5 Mejoras en la Implementación

- Interfaz Gráfica de Usuario (GUI). Desarrollar una interfaz que permita a los usuarios introducir valores objetivo y observar cómo el algoritmo explora el espacio de soluciones, mejorando así la interactividad y la comprensión del problema.

- Implementación en Otros Lenguajes. Explorar cómo la implementación del algoritmo varía en diferentes lenguajes de programación, lo cual puede ofrecer insights sobre eficiencia y legibilidad del código.

2.10.6 Investigación Teórica

- Análisis de Complejidad Detallado. Realizar un estudio formal sobre la complejidad del algoritmo en diferentes configuraciones del problema, identificando casos peores, mejores y promedio.
- Teoría de Números y Operaciones. Investigar desde una perspectiva matemática más profunda cómo las propiedades de los números y las operaciones influyen en la solución del problema, posiblemente descubriendo técnicas para predecir la existencia de soluciones sin necesidad de búsqueda exhaustiva.

[ ]: