Seminario_Algoritmos(1)

July 6, 2025

1 Algoritmos de optimización - Seminario

Nombre y Apellidos: Daniel Zapata Url: https://github.com/.../03MAIR—Algoritmos-de-Optimizacion—2019/tree/master/SEMINARIO GitHub: https://github.com/danizagra/Algoritmos_Seminario Problema:

3. Combinar cifras y operaciones

Descripción del problema:(copiar enunciado)

(*) La respuesta es obligatoria

1.1 Descripción del Problema

El problema consiste en analizar y diseñar un algoritmo que resuelva la siguiente situación:

1.1.1 Elementos Disponibles

- **Cifras**: Las 9 cifras del 1 al 9 (excluimos el cero)
- **Operadores**: Los 4 signos básicos de las operaciones fundamentales:
  - Suma (+)
  - Resta (-)
  - Multiplicación (*)
  - División (/)

1.1.2 Objetivo

Debemos combinarlos **alternativamente** sin repetir ninguno de ellos para obtener una cantidad dada.

1.1.3 Ejemplo

Para obtener el valor **4**: 4+2-6/3*1 = 4

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

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Respuesta

1. Teniendo en cuenta el enunciado vamos a utilizar 5 cifras y 4 operadores, que van a ser alternados entre ellos.

- **para las 5 posiciones de las cifras**: disponemos de 9 cifras distintas (1 al 9). Necesitamos elegir 5 de ellas y ordenarlas, por lo tanto tenemos una permutación de 9 elementos tomados de 5 en 5.
$$P(9,5) = \frac{9!}{(9-5)!} = \frac{9!}{4!} = 9x8x7x6x5 = 15120 posibilidades$$

- **Para las 4 posiciones de los operadores**: en este caso es mas sencillo ya que al tener solo 4 posiciones se van a tomar de 4 en 4, por lo tanto seria

$$P(4,4) = 4! = 4x3x2x1 = 24 posibilidades$$

Ahora teniendo esto claro podemos decir que: **Total de posibilidades** = Posibilidades de cifras x Posibilidades de operadores **Total de posibilidades** = 15120 x 23 = **362.880**

2. Vamos a tener en cuenta las restricciones:

- Combinarlos alternativamente sin repetir ninguno de ellos para obtener una cantidad dada
- Encontrar todos los valores enteros posibles

para responder a esta parte realmente es un poco dificil ya que deberiamos tener un planteamiento en codigo para darnos cuenta de las 362.880 cuantas posibilidades se pueden tener, no lo podemos calcular con una formula realmente.

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

La estructura de datos que mejor se adapta a este problema es una lista de listas, donde cada sublista representa una expresión matemática. Creo que podriamos tambien pensar en tupla de caracteres, para representar cada expresión matemática.

1. Cada representacion es una solucion que se puede representar como $4 + 2 - 6/3 * 1$ Una lista o tupla los permite almacenarlo de la siguiente forma: [‘4’, ‘+’, ‘2’, ‘-’, ‘6’, ‘/’, ‘3’, ’*‘, ’1’] y para mostrarlo podriamos usar una cadena de texto “4+2-6/3*1” Si usamos una lista, podemos usar técnicas de permutación y combinación para seleccionar los elementos y luego construirlos en el orden deseado

Según el modelo para el espacio de soluciones *()¿Cual es la función objetivo? ()*¿Es un problema de maximización o minimización?

Respuesta

Caso ideal: Si encontramos una expresión que evalúa exactamente al valor objetivo, entonces f(x) = 0, que es el mínimo absoluto posible. **Casos subóptimos**: Cualquier expresión que no evalúe exactamente al valor objetivo tendrá f(x) > 0, y mientras mayor sea esta diferencia, peor será la solución.

1.2 Definición de la Función Objetivo

El problema consiste en encontrar expresiones matemáticas que evalúen a un valor objetivo específico. La función objetivo se define como:

$$f(x) = |resultado_{expresin} - valor_{objetivo}|$$

Donde: - $resultado_{expresin}$ = valor numérico al evaluar la expresión matemática - $valor_{objetivo}$ = número que queremos obtener (ej: 4) - |...| = valor absoluto de la diferencia

1.3 Justificación: Es un Problema de Minimización

Este es un problema de minimización porque:

- **Objetivo**: Minimizar la diferencia entre el resultado y el valor deseado
- **Óptimo global**: $f(x) = 0$ (cuando encontramos el valor exacto)
- **Dirección de mejora**: Valores menores de $f(x)$ son mejores

1.4 Casos de Análisis

Caso	Descripción	Valor de f(x)	Calidad
Óptimo	$resultado = objetivo$	$f(x) = 0$	Mejor posible
Subóptimo cercano	$resultado \approx objetivo$	$f(x) > 0$ (pequeño)	Buena aproximación
Subóptimo lejano	$resultado \neq objetivo$	$f(x) >> 0$ (grande)	Mala solución

1.5 Ejemplo Concreto

Si buscamos expresiones que evalúen a 4:

1.5.1 Solución óptima

expresion_1 = 4+2-6/3*1 resultado_1 = eval(expresion_1) # = 4 f(x) = |4 - 4| = 0   (ÓPTIMO)

1.5.2 Solución subóptima

expresion_2 = 1+2+3-4/5" resultado_2 = eval(expresion_2) # = 5.2 f(x) = |5.2 - 4| = 1.2 (SUBÓPTIMO)

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

[3]:
```python
# Función recursiva para obtener permutaciones de una lista
def permutaciones(lista, n):
    if n == 0:
        return [[]]
    resultado = []
    for i in range(len(lista)):
        elem = lista[i]
        resto = lista[:i] + lista[i+1:]
        for p in permutaciones(resto, n-1):
            resultado.append([elem] + p)
```

```python
    return resultado

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

encontradas = 0
interacciones = 0
# Generar todas las permutaciones de 5 cifras
perms_cifras = permutaciones(cifras, 5)
print('Permutaciones de cifras posibles:', len(perms_cifras))
# Generar todas las permutaciones de 4 operadores
perms_ops = permutaciones(operadores, 4)
print('Permutaciones de operadores posibles:', len(perms_ops))
for perm_cifras in perms_cifras:
    for perm_ops in perms_ops:
        expr =
 ↪f"{perm_cifras[0]}{perm_ops[0]}{perm_cifras[1]}{perm_ops[1]}{perm_cifras[2]}{perm_ops[2]}{p
        try:
            resultado = eval(expr)
            interacciones += 1
            if resultado == valor_objetivo:
                #print(f"Solución encontrada: {expr} = {resultado}")
                encontradas += 1
        except ZeroDivisionError:
            continue

if encontradas == 0:
    print("No se encontraron soluciones.")
else:
    print(f"Total de soluciones encontradas: {encontradas}")
print(f"Total de interacciones: {interacciones}")
```

```
Permutaciones de cifras posibles: 15120
Permutaciones de operadores posibles: 24
Total de soluciones encontradas: 2112
Total de interacciones: 362880
```

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

Como lo habia mirado mas arriba los diferentes calculos serian:

$$P(9,5) = \frac{9!}{(9-5)!} = \frac{9!}{4!} = 9x8x7x6x5 = 15120 posibilidades$$

y las permutaciones de operadores seria:

$$P(4,4) = 4! = 4x3x2x1 = 24 posibilidades$$

esto al multiplicarlo nos daría 362,880. Este dato como se puede ver en los print es verdadero.

Teniendo todo esto presente podriamos decir que la complejidad es **O(n!)**, ya que se estan usando permutaciones de n elementos.

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

1.5.3 Se van a proponer tres mejoras diferenes al algoritmo todas teniendo enfoques diferentes

1.5.4 1. Respuesta

Teniendo en cuenta el codigo anterior, voy a proponer una mejora usando itertools, haciendo que mejore la iteracion, siendo una primera sugerencia usar permutations.

[4]:
```python
from itertools import permutations
import time

def fuerza_bruta_optimizada(valor_objetivo=4):
    """Referencia: fuerza bruta con itertools"""
    cifras = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    operadores = ['+', '-', '*', '/']

    soluciones = []
    evaluaciones = 0

    for perm_cifras in permutations(cifras, 5):
        for perm_ops in permutations(operadores, 4):
            expr_str = 
 ↪f"{perm_cifras[0]}{perm_ops[0]}{perm_cifras[1]}{perm_ops[1]}{perm_cifras[2]}{perm_ops[2]}{p

            try:
                resultado = eval(expr_str)
                evaluaciones += 1

                if abs(resultado - valor_objetivo) < 1e-10:
                    soluciones.append((expr_str, resultado))
            except ZeroDivisionError:
                evaluaciones += 1
                continue

    return soluciones, evaluaciones

print("=== FUERZA BRUTA OPTIMIZADA (REFERENCIA) ===")
start = time.time()
sol_ref, eval_ref = fuerza_bruta_optimizada(4)
time_ref = time.time() - start
```

```
print(f"Soluciones: {len(sol_ref)}, Evaluaciones: {eval_ref}, Tiempo: {time_ref:
↪.3f}s")
```

```
=== FUERZA BRUTA OPTIMIZADA (REFERENCIA) ===
Soluciones: 2112, Evaluaciones: 362880, Tiempo: 2.004s
```

1.6 Ventajas de la Mejora

1.6.1 Eficiencia en Memoria

- **Problema original**: Crea todas las permutaciones en memoria antes de procesarlas
- **Solución con itertools**: Generador que crea permutaciones una a una
- **Beneficio**: Ahorro significativo al no almacenar 15,120 permutaciones de cifras en memoria

1.6.2 Mejor Rendimiento

- **Implementación**: `itertools` está escrito en C, no en Python puro
- **Optimización**: Elimina overhead de llamadas recursivas
- **Mejora práctica**: Típicamente 30-50% más rápido

1.6.3 Código más Limpio

- **Reducción**: De ~15 líneas de código a 1 línea
- **Robustez**: Menos propenso a errores
- **Mantenibilidad**: Más legible y fácil de mantener

1.7 Impacto en el Problema

Aspecto	Valor	Observación
Complejidad	$O(n! \times m!)$	Sin cambio (sigue siendo 362,880 evaluaciones)
Soluciones encontradas	2,112	Idéntico al original
Uso de memoria	~90% menos	No almacena listas completas
Tiempo de ejecución	~30-40% menos	Por implementación en C

1.7.1 2. Respuesta

Ahora vamos a implementar otra alternativa usando un **branch and bound**, la idea es la siguiente: 1. Dividir el problema en subproblemas más pequeños (como las ramas de un árbol). Cada nodo representa una posible solución parcial.

2. Para cada subproblema, se calcula una cota superior o inferior del mejor resultado que puede lograrse desde ese nodo.

3. Se evitan explorar ramas del árbol que no pueden mejorar la solución óptima actual, lo que ahorra tiempo y recursos.

[5]:
```
def branch_and_bound_correcto(valor_objetivo=4):
    """Branch and Bound que replica la lógica exacta pero con podas por cotas"""
    cifras = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
    operadores = ['+', '-', '*', '/']

    soluciones = []
    evaluaciones = 0
    podas = 0

    # Usar la misma estructura que fuerza bruta pero con podas
    for perm_cifras in permutations(cifras, 5):
        # Poda rápida por cotas de números solamente
        suma_total = sum(perm_cifras)
        diferencia_maxima = perm_cifras[0] - sum(perm_cifras[1:])

        # Si ni la suma total ni la diferencia máxima se acercan al objetivo
        if suma_total < valor_objetivo - 2 or diferencia_maxima >␣
 ↪valor_objetivo + 2:
            podas += 1
            continue

        for perm_ops in permutations(operadores, 4):
            # Evaluación idéntica a fuerza bruta
            expr_str =␣
 ↪f"{perm_cifras[0]}{perm_ops[0]}{perm_cifras[1]}{perm_ops[1]}{perm_cifras[2]}{perm_ops[2]}{p

            try:
                resultado = eval(expr_str)
                evaluaciones += 1

                if abs(resultado - valor_objetivo) < 1e-10:
                    soluciones.append((expr_str, resultado))
            except ZeroDivisionError:
                evaluaciones += 1
                continue

    return soluciones, evaluaciones, podas



print("\n=== BRANCH & BOUND CORREGIDO ===")
start = time.time()
sol2, eval2, podas2 = branch_and_bound_correcto(4)
time2 = time.time() - start
print(f"Soluciones: {len(sol2)}, Evaluaciones: {eval2}, Podas: {podas2}, Tiempo:
 ↪ {time2:.3f}s")
```

```
=== BRANCH & BOUND CORREGIDO ===
Soluciones: 2112, Evaluaciones: 362880, Podas: 0, Tiempo: 1.928s
```

1.8 Ventajas: Estrategia de Poda por Cotas

Esta implementación añade una estrategia de **poda por cotas** al algoritmo de **fuerza bruta**, evitando explorar combinaciones de cifras que matemáticamente no pueden alcanzar el valor objetivo.

Criterio de poda:
Si estas cotas están muy lejos del objetivo, se descartan todas las combinaciones de operadores para esa permutación de cifras.

1.8.1 Ejemplo Práctico

Para `valor_objetivo = 4` y `cifras = [9, 8, 7, 6, 5]`:

- `suma_total = 35` (muy lejos de 4)

- `diferencia_maxima = 9 - (8 + 7 + 6 + 5) = -17` (también muy lejos de 4)

Decisión:
Podar esta rama (se saltan 24 evaluaciones de operadores)

1.9 Resultados Obtenidos

Métrica	Valor	Impacto
Soluciones encontradas	2,112	Idéntico (no pierde soluciones)
Evaluaciones totales	362,880	Sin cambio en este caso
Podas realizadas	0	La poda fue muy conservadora
Complejidad peor caso	$O(n! \times m!)$	Sin cambio

1.9.1 Fortalezas

- Mantiene la completitud (encuentra TODAS las soluciones)

- Reduce evaluaciones cuando el objetivo está fuera de rangos extremos

- No añade complejidad significativa al código

1.9.2 Limitaciones

- Para valores objetivo “normales” (como 4), la poda es inefectiva

- Las cotas son muy conservadoras para garantizar no perder soluciones

- No mejora la complejidad asintótica $O(n! \times m!)$

(*)Calcula la complejidad del algoritmo

Respuesta

> La complejidad del algoritmo `branch_and_bound_correcto` en el peor de los casos sigue siendo:
>
> $$\mathcal{O}(n! \times m!)$$
>
> Esto se debe a que, en el peor escenario, la poda no descarta ninguna combinación y se evalúan todas.
> Sin embargo, en la práctica, la poda puede reducir significativamente el número de combinaciones evaluadas,
> haciendo el algoritmo mucho más eficiente que la fuerza bruta en muchos casos concretos.

En términos concretos para este problema: - Peor caso: 362,880 evaluaciones (igual que fuerza bruta) - Caso promedio: significativamente menos debido a las podas por cotas - Las podas eliminan ramas que no pueden alcanzar el valor objetivo

1.10 3. Respuesta (EXTRA)

Dicho archivo se debe correr por fuera de Jupyter para poder tener una correcta paralelización, el archivo se encuentra con el nombre paralelizacion.py

1.10.1 Paralelización con Multiprocessing

1.10.2 ¿Qué se mejoró?

Esta implementación distribuye el trabajo entre múltiples procesos (cores) del CPU, permitiendo evaluar expresiones en paralelo sin cambiar la lógica del algoritmo.

1.10.3 Estrategia de Paralelización

- **División del trabajo:** Las 15,120 permutaciones de cifras se dividen equitativamente entre los procesos disponibles

- **Procesamiento independiente:** Cada proceso evalúa su bloque de permutaciones × todas las permutaciones de operadores

- **Combinación de resultados:** Al final se unen todas las soluciones encontradas por cada proceso

---

1.11 Resultados Obtenidos en tu Sistema

Configuración	Tiempo	Speedup	Eficiencia	Eval/seg
Secuencial	2.015s	1.00x	100%	180,074
2 procesos	1.287s	1.57x	78.3%	281,895

Configuración	Tiempo	Speedup	Eficiencia	Eval/seg
4 procesos	0.919s	2.19x	54.8%	394,985
10 procesos	0.986s	2.04x	20.4%	368,052

1.12 Análisis de Rendimiento

- **Punto óptimo:** 4 procesos con speedup de 2.19x
- **Reduce el tiempo** de 2.015s a 0.919s (54% menos tiempo)
- **Procesa ~395,000 expresiones por segundo** vs 180,000 secuencial

Rendimientos decrecientes: Con 10 procesos el speedup baja a 2.04x
- El overhead de coordinar muchos procesos supera los beneficios
- La eficiencia cae al 20.4% (desperdicio de recursos)

1.13 Ventajas y Limitaciones

1.13.1 Ventajas

- Mismas 2,112 soluciones - No sacrifica completitud
- Mismas 362,880 evaluaciones - Solo las distribuye
- Mejora real y medible - Hasta 2.19x más rápido
- Escalable - Funciona con cualquier número de cores

1.13.2 Limitaciones

- No cambia la complejidad - Sigue siendo O(n! $\times$ m!)
- Overhead de coordinación - Crear procesos tiene costo
- Eficiencia decrece - Más procesos mejor rendimiento
- Límite físico - Máximo speedup número de cores

1.14 ¿Se podría garantizar la optimalidad de la solución?

SÍ, los algoritmos planteados garantizan encontrar la solución óptima global.

1.14.1 Justificación:

1. **Búsqueda exhaustiva**: Se evalúan las 362,880 expresiones posibles sin excepción
2. **Sin heurísticas**: No se utilizan aproximaciones que puedan omitir soluciones válidas

3. **Evaluación completa**: Cada expresión se evalúa exactamente una vez
4. **Criterio determinista**: Si existe una expresión con f(x) = 0, será encontrada

1.14.2 Garantía matemática:

Al explorar todo el espacio de soluciones S = {todas las permutaciones posibles}, se garantiza que: - Si expresión tal que f(x) = 0 → el algoritmo la encontrará - Se identifican TODAS las soluciones óptimas (2,112 para objetivo = 4)

1.15 Estimación del máximo, mínimo y rango

1.15.1 Análisis teórico del espacio de soluciones:

Métrica	Cálculo	Valor
Máximo teórico	$9 \times 8 \times 7 \times 6 \times 5$	15,120
Mínimo teórico	$1 - 9 \times 8 \times 7 \times 6$	-3,023
Rango completo	[mínimo, máximo]	[-3,023, 15,120]

1.15.2 Observaciones importantes:

- **No todos los valores** en el rango [-3,023, 15,120] son alcanzables
- Las operaciones (÷ y -) pueden generar valores fraccionarios
- Para el valor objetivo 4, existen 2,112 expresiones exactas
- El Branch & Bound utiliza estas cotas para podar ramas imposibles

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

Respuesta

[6]:
```
import numpy as np

def generar_datos_prueba(num_casos=10):
    # Generamos valores entre 1 y 9 ya que son los números disponibles
    # para formar las expresiones matemáticas
    valores_objetivo = np.random.randint(low=1, high=10, size=num_casos)

    return valores_objetivo.tolist()

# Generamos conjunto de prueba
valores_prueba = generar_datos_prueba()
print("Valores objetivo generados para prueba:")
for i, valor in enumerate(valores_prueba, 1):
    print(f"Caso {i}: {valor}")
```

```
Valores objetivo generados para prueba:
Caso 1: 4
Caso 2: 2
Caso 3: 5
```

```
Caso 4: 2
Caso 5: 8
Caso 6: 5
Caso 7: 1
Caso 8: 6
Caso 9: 4
Caso 10: 7
```

Aplica el algoritmo al juego de datos generado

Respuesta

[7]:
```python
def probar_branch_and_bound_con_datos_aleatorios(valores_objetivo):
    for i, objetivo in enumerate(valores_objetivo, 1):
        print(f"\nCaso {i}: valor objetivo = {objetivo}")
        soluciones, evaluaciones, podas = branch_and_bound_correcto(objetivo)
        print(f"  Soluciones encontradas: {len(soluciones)}")
        print(f"  Evaluaciones realizadas: {evaluaciones}")
        print(f"  Podas realizadas: {podas}")
        if soluciones:
            print("  Ejemplo de solución:", soluciones[0])
        else:
            print("  No se encontró solución exacta.")

# Ejemplo de uso:
valores_objetivo = generar_datos_prueba(num_casos=4)
probar_branch_and_bound_con_datos_aleatorios(valores_objetivo)
```

```
Caso 1: valor objetivo = 1
  Soluciones encontradas: 2008
  Evaluaciones realizadas: 362880
  Podas realizadas: 0
  Ejemplo de solución: ('1+2-3*4/6', 1.0)

Caso 2: valor objetivo = 2
  Soluciones encontradas: 1688
  Evaluaciones realizadas: 362880
  Podas realizadas: 0
  Ejemplo de solución: ('1-2/3*6+5', 2.0)

Caso 3: valor objetivo = 9
  Soluciones encontradas: 2584
  Evaluaciones realizadas: 362880
  Podas realizadas: 0
  Ejemplo de solución: ('1-2*3/6+9', 9.0)

Caso 4: valor objetivo = 2
  Soluciones encontradas: 1688
```

```
Evaluaciones realizadas: 362880
Podas realizadas: 0
Ejemplo de solución: ('1-2/3*6+5', 2.0)
```

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

1.16 Líneas de Investigación Futuras

1.16.1 1. Mejoras Algorítmicas

- **Programación Dinámica**: Memorizar subproblemas para evitar recálculos repetidos
- **Podas más inteligentes**: Cotas matemáticas más precisas basadas en análisis de intervalos
- **Paralelización** (realizada como EXTRA): Distribuir permutaciones entre múltiples procesos para mayor velocidad

1.16.2 2. Variaciones del Problema

- **Extensión de operadores**: Incluir potencias, raíces, paréntesis
- **Números multi-dígito**: Formar números como 12, 34 en lugar de usar dígitos individuales
- **Objetivos múltiples**: Buscar expresiones que se aproximen a varios valores simultáneamente