

Análisis y Diseño de Algoritmos

Sistema de Asignación de Mecánicos y Averías

Francisco Javier Mercader Martínez
Pedro Alarcón Fuentes

En esta memoria se explicará el código utilizado para crear una sistema para la asignación de mecánicos a diferentes averías, evaluando las capacidades de los mecánicos para reparar averías específicas y optimizando el proceso de asignación. La estructura permite procesar varios casos de prueba y validar los resultados de cada asignación.

Índice de Funciones

1. `encontrar_archivos_in`
 2. `encontrar_archivos_out`
 3. `leer_entrada`
 4. `voraz`
 5. `factible`
 6. `select`
 7. `solution`
 8. `procesar_matriz`
 9. `generar_salida`
 10. `analyse_complexity`
-

1. Función `encontrar_archivos_in`

```
def encontrar_archivos_in(directorio='.'):
    """
    Busca todos los archivos .in en el directorio especificado.

    :param directorio: Ruta del directorio donde buscar archivos `.in`.
    :return: Lista de rutas de archivos `.in` encontradas.
    """
    archivos_in = []
    for root, _, files in os.walk(directorio):
        for file in files:
            if file.endswith('.in'):
                archivos_in.append(os.path.join(root, file))
    return archivos_in
```

Descripción: Busca y devuelve una lista de archivos con extensión `.in` en el directorio especificado, que se utilizarán como entradas de datos para el programa.

2. Función encontrar_archivos_out

```
def encontrar_archivos_out(directorio='.'):
    """
    Busca todos los archivos .out en el directorio especificado.

    :param directorio: Ruta del directorio donde buscar archivos `.out`.
    :return: Lista de rutas de archivos `.out` encontradas.
    """
    archivos_out = []
    for root, _, files in os.walk(directorio):
        for file in files:
            if file.endswith('.out'):
                archivos_out.append(os.path.join(root, file))
    return archivos_out
```

Descripción: Busca y devuelve una lista de archivos con extensión **.out** en el directorio especificado, que se utilizarán para realizar la validación de los resultados obtenidos.

3. Función leer_entrada

```
def leer_entrada(file_path):
    """
    Lee el archivo de entrada y convierte la información en una estructura de
    datos adecuada.

    :param file_path: Ruta del archivo de entrada.
    :return: Número de casos de prueba (P) y lista de casos de prueba.
    """
    with open(file_path, 'r') as file:
        lineas = file.readlines()

    P = int(lineas[0].strip())
    casos = []
    indice = 1

    for _ in range(P):
        M, A = map(int, lineas[indice].split())
        indice += 1

        capacidades = []
        for i in range(M):
            capacidades.append(list(map(int, lineas[indice].strip().split())))
            indice += 1

        casos.append({'M': M, 'A': A, 'capacidades': capacidades})
```

```
return P, casos
```

Descripción y objetivo de la función: La función `leer_entrada` se encarga de leer un archivo de entrada `.in` y extraer los datos necesarios para cada caso de prueba. Almacena el número de mecánicos y averías, así como las capacidades de cada mecánico para reparar averías específicas en una estructura de datos organizada.

Parámetros:

- `file_path`: La ruta al archivo de entrada

Proceso:

1. Lee todas las líneas del archivo de entrada y extrae el número de casos de prueba.
 2. Para cada caso, lee el número de mecánicos (**M**) y averías (**A**).
 3. Crea una matriz **capacidades** que indica las habilidades de cada mecánico para reparar las averías.
 4. Almacena cada caso en una lista de diccionarios.
-

4. Función voraz

```
def voraz(c, num_averias, mecanicos):  
    '''  
    Función que usa el algoritmo voraz visto en clase adaptado a este problema  
    '''  
    s = []  
    while len(c) > 0 and not solution(s, num_averias):  
        x = select(c, mecanicos)  
        c.remove(x)  
        if factible(s, x):  
            s.append(x)  
    return s
```

Descripción: Esta función implementa un algoritmo voraz para seleccionar las asignaciones de mecánicos y averías. La función itera sobre una lista de candidatos (pares de mecánico y avería) y selecciona aquellos que son factibles hasta que se encuentra una solución o se agotan los candidatos.

Parámetros:

- `c`: Lista de candidatos (pares de mecánico y avería).
- `num_averias`: Número total de averías.
- `mecanicos`: Matriz de capacidades de los mecánicos.

Proceso:

1. Inicializa una lista vacía **s** para almacenar las asignaciones.
 2. Mientras haya candidatos y no se haya encontrado una solución:
 - Selecciona un candidato **x** usando la función **select**.
 - Elimina el candidato seleccionado de la lista **c**.
 - Si el candidato es factible (usando la función **factible**), se agrega a la lista de asignaciones **s**.
-

5. Función factible

```
def factible(mecanico, averia, capacidades, asignaciones):  
    """  
    Verifica si un mecánico puede ser asignado a una avería.  
  
    mecanico: Índice del mecánico.  
    averia: Índice de la avería.  
    capacidades: Matriz C que indica las capacidades de los mecánicos.  
    asignaciones: Lista que indica si una avería ya fue asignada.  
  
    Retorna True si el mecánico puede reparar la avería y aún no ha sido  
↪ asignada.  
    """  
    return capacidades[mecanico][averia] == 1 and asignaciones[averia] == 0
```

Descripción: Verifica si un mecánico específico puede ser asignado a una avería dada. Esto depende de la capacidad del mecánico para repararla y de si la avería ha sido asignada.

Parámetros:

- **mecanico:** Índice del mecánico.
 - **averia:** Índice de la avería.
 - **capacidades:** Matriz que representa las capacidades de los mecánicos.
 - **asignaciones:** Lista de asignaciones para controlar qué averías han sido ya asignadas.
-

6. Función select

```
def select(mecanico, capacidades, asignaciones, A):  
    """  
    Selecciona la mejor avería que un mecánico puede reparar, si es posible.  
  
    mecanico: Índice del mecánico.
```

capacidades: Matriz C que indica las capacidades de los mecánicos.
asignaciones: Lista que indica si una avería ya fue asignada.
A: Número de averías.

↪ *Retorna el índice de la avería seleccionada o -1 si no hay ninguna disponible.*

```
"""
for averia in range(A):
    if factible(mecanico, averia, capacidades, asignaciones):
        return averia
return -1
```

Descripción: Esta función busca una avería que el mecánico puede reparar. Si encuentra una, devuelve el índice de la avería; si no, devuelve **-1**.

Proceso:

1. Recorre cada avería posible.
2. Usa la función **factible** para verificar si el mecánico puede asignarse a ella.
3. Devuelve el índice de la primera avería factible.

7. Función solution

```
def solution(P, casos):
    """
    Resuelve el problema para P casos de prueba

    P: Número de casos de prueba
    casos: Lista de casos de prueba, cada uno con M mecánicos, A averías y la
    ↪ matriz de capacidades

    Retorna una lista con las soluciones de cada caso
    """
    resultados = []

    for caso in casos:
        M, A, capacidades = caso['M'], caso['A'], caso['capacidades']

        asignaciones = [0] * A
        averias_reparadas = 0

        for mecanico in range(M):
            averia_seleccionada = select(mecanico, capacidades, asignaciones, A)
            if averia_seleccionada != -1:
                asignaciones[averia_seleccionada] = mecanico + 1
                averias_reparadas += 1
```

```
resultados.append((averias_reparadas, asignaciones))
```

```
return resultados
```

Descripción y objetivo de la función: `solution` implementa el proceso de asignación para cada caso de prueba. Para cada mecánico, selecciona una avería que pueda reparar, y si la encuentra, se asigna y actualiza el número total de averías reparadas.

Proceso:

1. Para cada caso, inicializa `asignaciones` con 0 para indicar avería ha sido asignada.
 2. Para cada mecánico, selecciona la avería adecuada mediante la función `select`.
 3. Lleva un conteo de averías asignadas y almacena el resultado de cada caso en una lista.
-

8. Función `procesar_matriz`

```
def procesar_matriz(matriz):  
    '''  
    Esta función procesa la matriz dada y saca el número de averias, candidatos y  
    ↪ usa el algoritmo voraz para obtener una solución  
    '''  
    num_averias = matriz.shape[1]  
    candidatos = [(i, j) for i in range(matriz.shape[0]) for j in  
    ↪ range(matriz.shape[1]) if matriz[i][j] == 1]  
    solucion = voraz(candidatos, num_averias, matriz)  
    resultado = [0] * num_averias  
    for i, j in solucion:  
        resultado[j] = i + 1 # Asignar el mecánico (i+1) a la avería (j)  
    return resultado
```

Descripción y objetivo de la función: `procesar_matriz` toma una matriz de capacidades y aplica el algoritmo voraz para generar una lista de asignaciones de mecánicos

9. Función `generar_salida`

```
def generar_salida(matrices):  
    casos = []  
    for matriz in matrices:  
        M = len(matriz)  
        A = len(matriz[0]) if M > 0 else 0  
        capacidades = matriz  
  
        casos.append({'M': M, 'A': A, 'capacidades': capacidades})  
    resultados = solution([], casos)  
    return resultados
```

```

file_paths_in = encontrar_archivos_in('.')
file_paths_out = encontrar_archivos_out('.')

P, casos = leer_entrada(file_paths_in[0])
matrices = [caso['capacidades'] for caso in casos]
resultados = generar_salida(matrices)
print(P)

```

```
## 20
```

```

for resultado in resultados:
    print(resultado[0])
    print(textwrap.fill(' '.join(map(str, resultado[1])), width=86))

```

```

## 4
## 1 2 3 5
## 7
## 1 2 3 5 4 9 7 0
## 3
## 1 2 3 0 0
## 7
## 1 4 2 5 3 7 6 0 0 0 0
## 1
## 0 2
## 5
## 1 3 2 6 4 0
## 4
## 3 2 4 1 0 0 0 0 0 0 0 0 0 0 0
## 2
## 1 6
## 11
## 1 2 6 4 3 8 9 7 5 0 10 11 0 0 0 0
## 8
## 3 2 4 1 5 6 7 8
## 4
## 0 2 1 4 3 0 0 0 0 0 0 0 0 0 0
## 2
## 1 0 2
## 9
## 3 1 6 2 4 5 8 7 9
## 3
## 1 2 4
## 11
## 2 3 1 4 7 5 8 10 9 11 6 0
## 6
## 3 1 2 4 5 6
## 10
## 2 1 6 3 7 5 8 10 4 11
## 2
## 1 4 0

```

```
## 8
## 1 2 3 4 5 8 6 7 0 0 0 0 0 0 0
## 11
## 1 2 4 6 3 5 9 7 13 8 10
```

```
print()
```

Descripción: `generar_salida` organiza los resultados para múltiples matrices de capacidades, aplicando `solution`.

Resultado

Análisis de los resultados con el validador

```
import os
for file_path in file_paths_in:
    output_name = os.path.splitext(file_path)[0] + '_out.txt'
    output_file = os.path.join(directorio, output_name)

    P, casos = leer_entrada(file_path)

    matrices = [caso['capacidades'] for caso in casos]
    resultados = generar_salida(matrices)

    with open(output_file, 'w') as file:
        file.write(f"{P}\n")
        for resultado in resultados:
            file.write(f"{resultado[0]}\n")
            file.write(" ".join(map(str, resultado[1])) + "\n")

    corresponding_out_file = os.path.splitext(file_path)[0] + '.out'
    if os.path.exists(corresponding_out_file):
        print(f"\nValidando {corresponding_out_file}...")
        validador_E_AR(fichero_entrada=file_path,
                       fichero_salida=output_file,
                       fichero_salida_profesor=corresponding_out_file)
        # Se eliminan las salidas del algoritmo para evitar archivos residuales
        os.remove(output_file)
    else:
        print(f"Archivo de salida del profesor no encontrado para {file_path}")

## Validando 701a.out...
## En el caso 0: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 1: Todo correcto y el porcentaje de averías reparadas (7,8) es del 85.71%.
## En el caso 2: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 3: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 4: Todo correcto y el porcentaje de averías reparadas es del 100%.
```


[illegible]

[illegible]

```

## En el caso 137: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 138: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 139: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 140: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 141: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 142: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 143: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 144: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 145: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 146: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 147: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 148: Todo correcto y el porcentaje de averías reparadas es del 100%.
## En el caso 149: Todo correcto y el porcentaje de averías reparadas es del 100%.
## La media porcentual para 150 casos es del 99.66%.

```

Análisis del Desempeño del Algoritmo Voraz

```

import matplotlib.pyplot as plt

def analyse_complexity(file_path):
    """
    Procesa el archivo, calcula el tiempo acumulado para cada iteración y lo
    ↪ grafica
    """
    # Leer todos los casos del archivo
    P, casos = leer_entrada(file_path)

    overall_time = []      # Lista para almacenar el tiempo acumulado en cada caso
    accumulated_time = 0   # Tiempo inicial acumulado

    # Procesar cada caso (matriz) individualmente
    for caso in casos:
        # Medir el tiempo de inicio para el caso actual
        start_time = time.time()

        M, A, capacidades = caso['M'], caso['A'], caso['capacidades']
        num_averias = A
        candidatos = [(i, j) for i in range(M) for j in range(A) if capacidades[i][j]
        ↪ == 1]

        # Algoritmo voraz para representar el progreso en cada caso
        s = []
        asignaciones = [0] * A

        for candidato in candidatos:
            mecanico, averia = candidato
            # Verificar si es factible asignar el mecánico a la avería
            if factible(mecanico, averia, capacidades, asignaciones):
                s.append(candidato)
                asignaciones[averia] = mecanico + 1

```

```

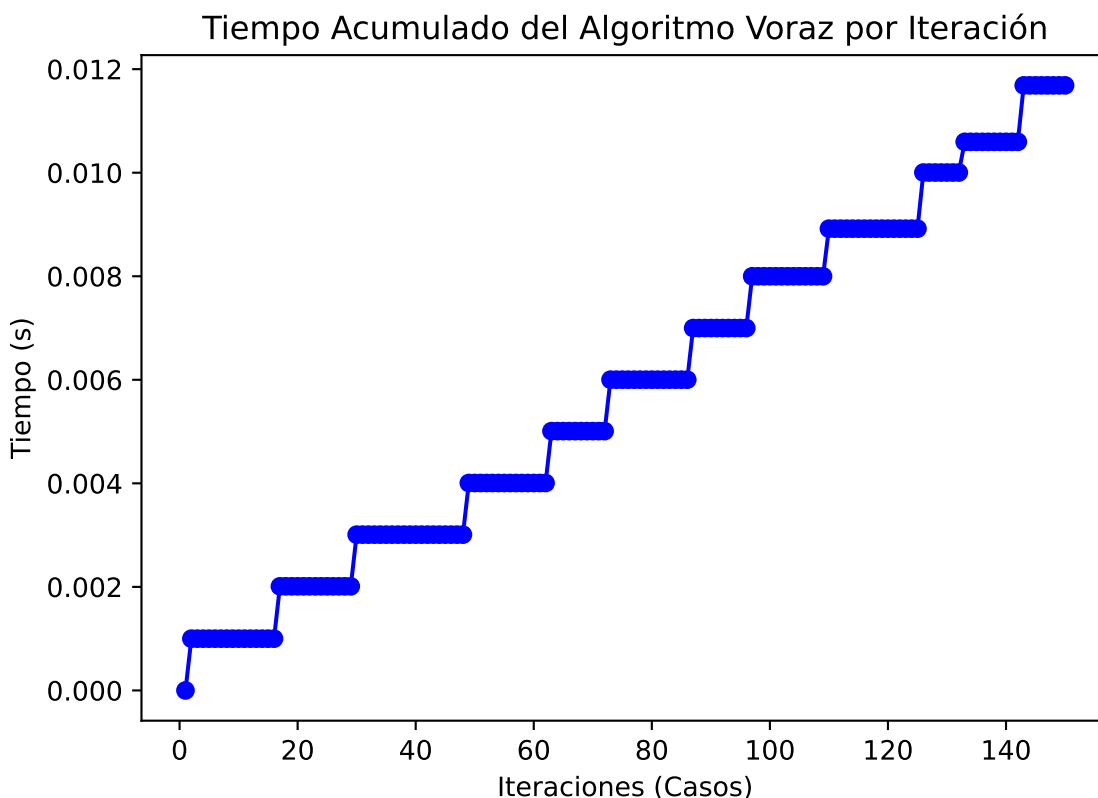
# Medir el tiempo de finalización y calcular el tiempo de procesamiento para
↳ el caso actual
end_time = time.time()
accumulated_time += (end_time - start_time)

# Agregar el tiempo acumulado a la lista general
overall_time.append(accumulated_time)

# Graficar el tiempo acumulado en función de las iteraciones
if overall_time:
    plt.figure()
    plt.plot(range(1, P + 1), overall_time, marker='o', color='blue')
    plt.xlabel("Iteraciones (Casos)")
    plt.ylabel("Tiempo (s)")
    plt.title("Tiempo Acumulado del Algoritmo Voraz por Iteración")
    plt.show()
else:
    print(f"No se realizaron asignaciones en ningún caso")

```

```
analyse_complexity(file_path="tests/T3/701c.in")
```



En el gráfico, observamos que el tiempo de ejecución del algoritmo voraz aumenta de manera progresiva conforme se incrementa el número de casos. Este crecimiento parece ser bastante lineal, lo cual es positivo, ya que el algoritmo maneja bastante bien el aumento de datos sin volverse excesivamente lento.

Además, el patrón escalonado en el gráfico sugiere que algunas operaciones del algoritmo un poco más de tiempo en ciertos puntos, posiblemente debido a ciertas decisiones o procesos internos que

demandan un esfuerzo adicional. A pesar de esto, el incremento sigue siendo controlado y predecible.