Análisis y Diseño de Algoritmos

Sistema de Asignación de Mecánicos y Averías

Francisco Javier Mercader Martínez
Pedro Alarcón Fuentes

Esta memoria describe el código desarrollado para asignar mecánicos a diferentes averías en base a sus capacidades, utilizando algoritmos de backtracking y optimización. Este sistema permite procesar múltiples casos de prueba y seleccionar la mejor asignación posible.

Índice de Funciones

---

1. Función `encontrar_archivos_in`

```python
def encontrar_archivos_in(directorio='.'):
    """
    Encuentra todos los archivos con extensión '.in' en el directorio dado.

    :param directorio: Directorio en el que buscar archivos.
    :return: Lista de rutas a los archivos encontrados.
    """
    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: Esta función busca archivos de entrada con extensión **`.in`** en el directorio especificado. Es útil para localizar archivos de datos en los que se almacenarán los casos de prueba.

---

2. Función `leer_entrada`

```python
def leer_entrada(file_path):
    """
    Lee el archivo de entrada y extrae los casos de prueba.

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

    P = int(lineas[0])
    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].split())))
            indice += 1

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

    return P, casos
```

Descripción: `leer_entrada` lee un archivo de entrada y extrae los datos necesarios para cada caso de prueba. Genera una lista con el número de mecánicos, averías y sus respectivas capacidades.

---

3. Función `factible`

```python
def factible(mecanico, averia, capacidades, asignaciones):
    """
    Verifica si un mecánico puede ser asignado a una avería específica.

    :param mecanico: Índice del mecánico.
    :param averia: Índice de la avería.
    :param capacidades: Matriz de capacidades que indica si un mecánico puede
    ↪ reparar una avería.
    :param asignaciones: Lista de asignaciones actuales de averías.
    :return: True si la asignación es factible, False en caso contrario.
    """
    return capacidades[mecanico][averia] == 1 and asignaciones[averia] == 0
```

Descripción: Verifica si un mecánico específico puede ser asignado a una avería en función de su capacidad y la disponibilidad de la avería.

---

4. Función `backtrack`

```
def backtrack(mecanico, averias_reparadas, asignaciones, capacidades, M, A,
↪ soluciones):
    """
    Implementa el algoritmo de backtracking para asignar mecánicos a averías.

    :param mecanico: Índice del mecánico actual.
    :param averias_reparadas: Número de averías reparadas hasta el momento.
    :param asignaciones: Lista de asignaciones actuales de averías.
    :param capacidades: Matriz de capacidades que indica si un mecánico puede
↪ reparar una avería.
    :param M: Número total de mecánicos.
    :param A: Número total de averías.
    :param soluciones: Lista para almacenar soluciones posibles.
    :return: None
    """
    # Si todas las averías han sido asignadas, guardamos la solución
    if averias_reparadas == A:
        soluciones.append(list(asignaciones))  # Guarda una copia de la asignación
↪ actual
        return False  # Continuar buscando más soluciones para optimización

    if mecanico >= M:
        return False  # No hay más mecánicos para asignar

    # Intentamos asignar este mecánico a cada avería posible
    for averia in range(A):
        if factible(mecanico, averia, capacidades, asignaciones):
            # Asignamos la avería al mecánico
            asignaciones[averia] = mecanico + 1
            averias_reparadas += 1

            # Llamada recursiva al siguiente mecánico
            backtrack(mecanico + 1, averias_reparadas, asignaciones, capacidades,
↪ M, A, soluciones)

            # Retroceso: Deshacemos la asignación
            asignaciones[averia] = 0
            averias_reparadas -= 1

    # Intentamos el siguiente mecánico sin asignaciones
    backtrack(mecanico + 1, averias_reparadas, asignaciones, capacidades, M, A,
↪ soluciones)
```

Descripción: La función **backtrack** implementa un algoritmo de backtracking para buscar asignaciones posibles entre mecánicos y averías. Genera soluciones factibles y guarda aquellas que cumplen con los requisitos.

---

5. Función seleccionar_solucion_optima

```python
def seleccionar_solucion_optima(soluciones):
    """
    Selecciona la mejor solución entre las posibles, minimizando el número de
    ↪ mecánicos y dejando los ceros al final.

    :param soluciones: Lista de soluciones posibles.
    :return: La solución óptima según el criterio establecido.
    """
    # Por ejemplo, minimizar el número de mecánicos utilizados y que los ceros
     ↪  queden al final
    soluciones.sort(key=lambda x: (x.count(0), x))
    return soluciones[0]
```

Descripción: Selecciona la solución óptima basada en minimizar el número de mecánicos asignados y asegura que las averías no asignadas (marcadas con ceros) queden al final de la lista.

---

6. Función solution_backtracking

```python
def solution_backtracking(P, casos):
    """
    Encuentra la asignación óptima de mecánicos a averías para cada caso
    ↪ utilizando backtracking.

    :param P: Número de casos de prueba.
    :param casos: Lista de diccionarios con información de cada caso.
    :return: Lista de resultados con el número de averías reparadas y las
    ↪ asignaciones correspondientes.
    """
    resultados = []

    for caso in casos:
        M, A, capacidades = caso['M'], caso['A'], caso['capacidades']
        asignaciones = [0] * A  # Inicialmente, ninguna avería está asignada
        soluciones = []  # Almacena soluciones posibles

        # Llamada inicial al algoritmo de backtracking
        backtrack(0, 0, asignaciones, capacidades, M, A, soluciones)
```

```
        # Seleccionar la mejor solución según el criterio establecido
        if soluciones:
            solucion_optima = seleccionar_solucion_optima(soluciones)
            averias_reparadas = sum(1 for x in solucion_optima if x != 0)
            resultados.append((averias_reparadas, solucion_optima))
        else:
            resultados.append((0, asignaciones))  # En caso de no encontrar
↪   ninguna solución

    return resultados
```

Descripción: Para cada caso de prueba, esta función ejecuta el algoritmo de backtracking para encontrar la asignación óptima de mecánicos a averías. Almacena el número de averías reparadas y las asignaciones óptimas en una lista de resultados.

---

7. Función `generar_salida`

```
def generar_salida(matrices):
    """
    Genera la salida para los casos de prueba a partir de las matrices de
↪   capacidades.

    :param matrices: Lista de matrices de capacidades.
    :return: Lista de resultados para cada caso.
    """
    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_backtracking(len(casos), casos)
    return resultados
```

Descripción: **`generar_salida`** organiza los datos para cada matriz de capacidades y llama a **`solution_backtracking`** para obtener los resultados finales de las asignaciones.