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 un archivo de entrada y organiza los casos de prueba. Cada caso de prueba contiene:
    - El número de mecánicos (M)
    - El número de averías (A)
    - Una matriz de capacidades, donde cada fila representa un mecánico y cada columna una avería.
      Un valor de 1 en la matriz indica que el mecánico puede reparar la avería en esa posición.

    :param file_path: Ruta del archivo de entrada.
    :return: Un par de valores (P, casos), donde:
             - P es el número total de casos de prueba en el archivo.
             - casos es una lista de diccionarios, donde cada diccionario representa un caso con los
               campos 'M', 'A', y 'capacidades'.
    """
    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 = [list(map(int, lineas[indice + i].split())) for i in range(M)]
            casos.append({'M': M, 'A': A, 'capacidades': capacidades})
            indice += M
    return P, casos
```

Descripción: Esta función lee un archivo de entrada y extrae la información necesaria 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 organizada.

---

3. Función `generar`

```python
def generar(nivel, s, M, capacidades, mecanicos_usados):
    """
    Genera una asignación de mecánico para una avería específica.

    :param nivel: Nivel actual del árbol de decisión (corresponde a la avería).
    :param s: Lista de asignaciones actuales.
```

```
    :param M: Número de mecánicos.
    :param capacidades: Matriz de capacidades de los mecánicos.
    :param mecanicos_usados: Lista de mecánicos ya asignados.
    :return: Lista de asignaciones actualizada.
    """
    for mecanico in range(1, M + 1):
        if (0 <= mecanico - 1 < len(capacidades)) and (0 <= nivel - 1 <
        ↪  len(capacidades[0])):
            if not mecanicos_usados[mecanico - 1] and capacidades[mecanico -
            ↪  1][nivel - 1] == 1:
                s[nivel - 1] = mecanico
                mecanicos_usados[mecanico - 1] = True
                return s
    return s
```

Descripción: La función **generar** selecciona un mecánico disponible para una avería dada en el nivel actual, si es posible. El mecánico seleccionado se marca como usado.

---

4. Función `solucion`

```
def solucion(nivel, A):
    """
    Verifica si se ha encontrado una solución completa.

    :param nivel: Nivel actual del árbol de decisión.
    :param A: Número de averías.
    :return: True si se ha alcanzado una solución completa, False en caso
↪   contrario.
    """
    return nivel == A + 1
```

Descripción: Esta función verifica si se ha alcanzado un nivel que indica que todas las averías han sido reparadas, es decir, si se ha llegado a una solución completa.

---

5. Función `criterio`

```
def criterio(nivel, s, mejor_reparadas, soluciones):
    """
    Evalúa si la solución actual es mejor que la solución registrada.

    :param nivel: Nivel actual del árbol de decisión.
    :param s: Lista de asignaciones actuales.
```

```
    :param mejor_reparadas: Número máximo de averías reparadas registrado.
    :param soluciones: Lista con la mejor solución encontrada.
    :return: True siempre, para continuar con el proceso de búsqueda.
    """
    reparadas_actuales = sum(1 for x in s if x > 0)
    if reparadas_actuales > mejor_reparadas[0]:
        mejor_reparadas[0] = reparadas_actuales
        soluciones[:] = s.copy()
    return True
```

Descripción: Esta función evalúa el criterio de optimalidad. Si el número de averías reparadas en la solución actual supera al mejor registrado hasta el momento, se actualiza la solución óptima.

---

6. Función `masHermanos`

```
def masHermanos(nivel, s, M, capacidades, mecanicos_usados):
    """
    Determina si existen más mecánicos que pueden ser asignados a una avería en
 ↪  el nivel actual.

    :param nivel: Nivel actual del árbol de decisión.
    :param s: Lista de asignaciones actuales.
    :param M: Número de mecánicos.
    :param capacidades: Matriz de capacidades de los mecánicos.
    :param mecanicos_usados: Lista de mecánicos ya asignados.
    :return: True si hay más mecánicos disponibles, False en caso contrario.
    """
    mecanico_actual = s[nivel - 1]
    for mecanico in range(mecanico_actual + 1, M + 1):
        if not mecanicos_usados[mecanico - 1] and capacidades[mecanico - 1][nivel
         ↪  - 1] == 1:
            return True
    return False
```

Descripción: Esta función determina si existen más mecánicos candidatos para asignar a la avería del nivel actual.

---

7. Función `retroceder`

```
def retroceder(nivel, s, mecanicos_usados):
    """
    Retrocede un nivel en el árbol de decisiones deshaciendo asignación de un
 ↪  mecánico.
```

```
    :param nivel: Nivel actual del árbol de decisión.
    :param s: Lista de asignaciones actuales.
    :param mecanicos_usados: Lista de mecánicos ya asignados.
    :return: Lista de mecánicos actualizada después del retroceso.
    """
    mecanico_actual = s[nivel - 1]
    if mecanico_actual > 0:
        mecanicos_usados[mecanico_actual - 1] = False
    s[nivel - 1] = 0
    return s
```

Descripción: La función **retroceder** deshace la asignación de un mecánico al nivel actual, permitiendo explorar otras opciones.

---

8. Función backtracking

```
def backtracking(s_inicial, M, A, capacidades, mejor_reparadas, soluciones):
    """
    Implementa el algoritmo de backtracking para encontrar la asignación óptima
    de mecánicos a averías.

    :param s_inicial: Lista inicial de asignaciones (inicialmente vacía).
    :param M: Número de mecánicos.
    :param A: Número de averías.
    :param capacidades: Matriz de capacidades de los mecánicos.
    :param mejor_reparadas: Lista que almacena el número máximo de averías
    reparadas.
    :param soluciones: Lista que almacena la mejor solución encontrada.
    :return: Lista de asignaciones óptima.
    """
    nivel = 1
    s = s_inicial
    mecanicos_usados = [False] * M
    fin = False
    while not fin and nivel != 0:
        s = generar(nivel, s, M, capacidades, mecanicos_usados)
        if solucion(nivel, A):
            fin = True
        elif criterio(nivel, s, mejor_reparadas, soluciones):
            nivel += 1
        else:
            while not masHermanos(nivel, s, M, capacidades, mecanicos_usados) and
            ↪ nivel > 0:
                s = retroceder(nivel, s, mecanicos_usados)
                nivel -= 1
    return s
```

Descripción: La función principal de backtracking explora todas las posibles asignaciones de mecánicos a averías, avanzando o retrocediendo según sea necesario para encontrar la solución óptima.

---

9. Función `backtracking_todas`

```
def backtracking_todas(M, A, capacidades):
    """
    Encuentra la mejor solución posible utilizando el algoritmo de backtracking.

    :param M: Número de mecánicos.
    :param A: Número de averías.
    :param capacidades: Matriz de capacidades de los mecánicos.
    :return: Número máximo de averías reparadas y la lista de asignaciones
    correspondiente.
    """
    soluciones = [0] * A
    mejor_reparadas = [0]
    s_inicial = [0] * A
    backtracking(s_inicial, M, A, capacidades, mejor_reparadas, soluciones)
    return mejor_reparadas[0], soluciones
```

Descripción: Esta función implementa el algoritmo de backtracking para encontrar la solución que maximiza el número de averías reparadas.

---

10. Función `solution_todas`

```
def solution_todas(P, casos):
    """
    Procesa todos los casos de prueba y obtiene las soluciones óptimas para cada
    uno,
    utilizando el algoritmo de backtracking adaptado.

    :param P: Número de casos de prueba.
    :param casos: Lista de casos de prueba, donde cada caso es un diccionario con
    los siguientes datos:
                  - 'M': Número de mecánicos
                  - 'A': Número de averías
                  - 'capacidades': Matriz de capacidades de los mecánicos
    :return: Un string formateado que contiene las soluciones para cada caso de
    prueba en el formato:
             "{P}\n{reparadas}\n{asignaciones}\n...", donde "reparadas" es el
    número de averías reparadas
             y "asignaciones" es la lista de asignaciones de mecánicos.
```

```
    """
    resultados = []
    for caso in casos:
        M, A, capacidades = caso['M'], caso['A'], np.array(caso['capacidades'])
        max_reparadas, solucion = backtracking_todas(M, A, capacidades)
        resultado = f"{max_reparadas}\n{' '.join(map(str, solucion))}"
        resultados.append(resultado)

    return f"{P}\n" + "\n".join(resultados)
```

Resultados

Análisis de los resultados

```
file_paths_in = encontrar_archivos_in('.')

P, casos = leer_entrada(file_path=file_paths_in[0]) # Para demostrar que el código
 ↪  funciona
resultados = solution_todas(P, casos)

print(resultados)
```

```
## 20
## 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 0
## 11
## 1 2 4 6 3 5 9 7 13 8 10
```

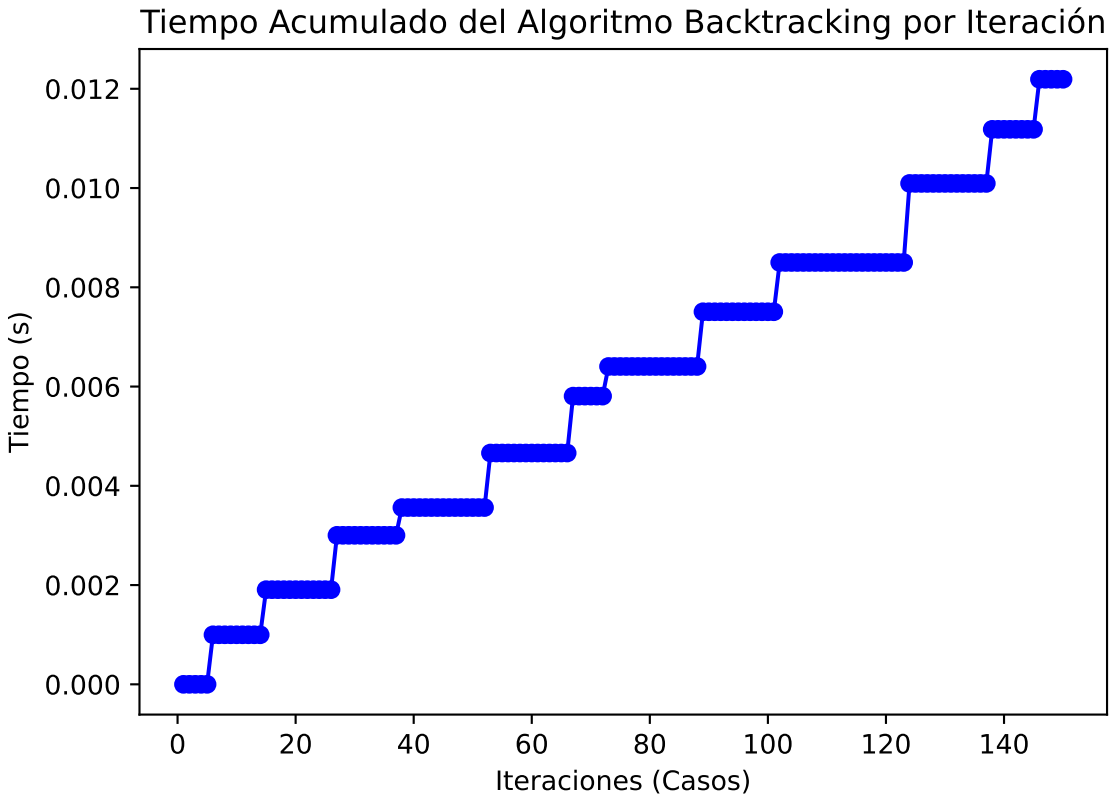
Análisis del Desempeño del Algoritmo de Backtracking

```python
def analyse_complexity(file_path):
    """
    Procesa el archivo, calcula el tiempo acumulado para cada iteración y lo
    grafica.

    :param file_path: Ruta del archivo de entrada.
    :return: None. Genera una gráfica del tiempo acumulado del algoritmo por
    iteración.
    """
    """
    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']
        max_reparadas, solucion = backtracking_todas(M, A, capacidades)

        # 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 Backtracking por Iteración")
        plt.show()
    else:
        print(f"No se realizaron asignaciones en ningún caso")
```

Descripción: Esta función permite analizar el desempeño del algoritmo de backtracking graficando el tiempo acumulado de procesamiento para cada caso de prueba.

```
analyse_complexity(file_path=file_paths_in[2])
```



El análisis de la gráfica del tiempo acumulado del algoritmo de backtracking muestra un crecimiento constante y en forma de escalones. Esto significa que, aunque el tiempo aumenta a medida que se realizan más iteraciones, lo hace de manera bastante controlada y predecible. Los "escalones" en la gráfica nos indican que, en ciertas etapas, el algoritmo requiere un poco más de tiempo para encontrar una solución, pero no vemos un crecimiento drástico que sea señal de problemas de eficiencia graves.

Esto sugiere que el algoritmo implementado está manejando bien la búsqueda de soluciones. Aunque el backtracking suele ser conocido por ser lento cuando se enfrenta a muchos posibles caminos, en este caso parece que las restricciones del problema nos ayudan a mantener el tiempo bajo control.

En general, la gráfica nos da una buena señal de que el algoritmo es eficiente para este problema particular, y el comportamiento observado indica que no se necesitan cambios urgentes.