Instituto Politécnico Nacional

Escuela Superior de Cómputo

Unidad de académica:

Análisis y diseño de algoritmos

Práctica 5:

“Algoritmos Voraces”

Alumno:

Solares Velasco Arturo Misael 2023630538

Solis Lugo Mayra 2023630449

Grupo: 3CV1

Profesora: Floriano García Andres

Fecha:

24 abril 2024

Reporte de Práctica: Implementación de Metodologías Voraces

En esta práctica se implementaron dos metodologías voraces en Python para abordar problemas de asignación de memoria y el problema de la mochila fraccionaria. A continuación, se presenta un resumen de las metodologías y su aplicación en diferentes casos de prueba.

1. Asignación de Memoria

Metodología:

Se implementaron dos criterios voraces para asignar memoria a procesos en bloques de memoria disponibles. Ambos criterios buscan optimizar el uso de la memoria sin considerar futuras asignaciones.

1. **Criterio 1: Primer Ajuste**
    - Este criterio asigna el proceso al primer bloque de memoria disponible que pueda contenerlo.
2. **Criterio 2: Mejor Ajuste**
    - Este criterio busca el bloque con el espacio libre más cercano al tamaño del proceso. Busca minimizar el desperdicio de memoria.

Código fuente:

```python
# Asignación de memoria con el primer criterio voraz
def asignacion_bloques_caso1(bloques, procesos):
    asignaciones = []

    for proceso in procesos:
        for i, bloque in enumerate(bloques):
            if bloque ≥ proceso:
                # Asigna al primer bloque disponible que pueda contener el proceso
                asignaciones.append((proceso, i))
                bloques[i] -= proceso
                break

    return asignaciones


# Asisgnaciòn de memoria con el segundo criterio voraz
def asignacion_bloques_caso2(bloques, procesos):
    asignaciones = []

    for proceso in procesos:
        mejor_espacio = max(bloques)
        mejor_indice = bloques.index(mejor_espacio)

        for i, bloque in enumerate(bloques):
            if bloque ≥ proceso and bloque < mejor_espacio:
                mejor_espacio = bloque
                mejor_indice = i

        asignaciones.append((proceso, mejor_indice))
        bloques[mejor_indice] -= proceso

    return asignaciones
```

Resultados:

Se realizaron pruebas con varios conjuntos de bloques de memoria y procesos para evaluar el rendimiento de cada criterio.

- **Ejemplo de la Clase:**
    - Se aplicaron ambos criterios a un conjunto de bloques de memoria y procesos proporcionado por la clase.
    - Se observaron diferencias en las asignaciones realizadas por cada criterio, destacando la variabilidad en la cantidad de bloques utilizados.

Datos para la ejecución del programa:

```python
bloques_memoria_clase = [150, 80, 120, 200]
procesos_clase = [30, 100, 50, 70]
```

Resultados del ejemplo:

```
Asignaciones caso de la clase:

    La asignaciòn de la memoria en el primer caso es de:
    [(30, 0), (100, 0), (50, 1), (70, 2)]

    La asignaciòn de la memoria en el segundo caso es de:
    [(30, 1), (100, 3), (50, 2), (70, 3)]
```

- **Ejemplo 1:**
    - Se aplicaron los dos criterios a un conjunto de bloques de memoria y procesos.
    - Se observaron diferencias en las asignaciones realizadas por cada criterio, influenciadas por el tamaño de los bloques y procesos.

Datos para la ejecución del programa:

```
bloques_memoria = [100, 500, 200, 300, 600]
procesos = [212, 417, 112, 426]
```

Resultados del ejemplo:

```
Asignaciones caso ejemplo 1:

    La asignaciòn de la memoria en el primer caso es de:
    [(212, 1), (417, 4), (112, 1)]

    La asignaciòn de la memoria en el segundo caso es de:
    [(212, 3), (417, 2), (112, 1), (426, 4)]
```

- **Ejemplo 2 y Ejemplo 3:**
    - Se realizaron pruebas adicionales con conjuntos de datos más grandes para evaluar el desempeño de los criterios en situaciones más complejas.
    - Se identificaron patrones de asignación similares a los observados en los ejemplos anteriores, con diferencias en la eficiencia de utilización de la memoria.

Datos para la ejecución del programa:

```
bloques_memoria_2 = [100, 500, 200, 300, 600]
procesos_2 = [212, 417, 112, 426, 112, 426]
```

```
bloques_memoria_3 = [100, 500, 200, 300, 600]
procesos_3 = [212, 417, 112, 426, 112, 426, 112, 426]
```

Resultados de los ejemplos:

```
Asignaciones caso ejemplo 2:

    La asignaciòn de la memoria en el primer caso es de:
    [(212, 1), (417, 4), (112, 1), (112, 1)]

    La asignaciòn de la memoria en el segundo caso es de:
    [(212, 3), (417, 2), (112, 4), (426, 0), (112, 3), (426, 4)]
Asignaciones caso ejemplo 3:

    La asignaciòn de la memoria en el primer caso es de:
    [(212, 1), (417, 4), (112, 1), (112, 1), (112, 2)]

    La asignaciòn de la memoria en el segundo caso es de:
    [(212, 3), (417, 4), (112, 0), (426, 2), (112, 3), (426, 1), (112, 0), (426, 3)]
```

2. Problema de la Mochila Fraccionaria

Metodología:

Se implementó un algoritmo voraz para resolver el problema de la mochila fraccionaria, donde se deben seleccionar elementos de una lista con valor y peso asociados, maximizando el valor total sin exceder una capacidad máxima.

Código fuente:

mochila_voraz.py > ...

```python
def mochila_fraccionaria(items, capacidad):
    # Calcula la razón valor/peso para cada artículo
    ratios = [(item[0] / item[1], item) for item in items]
    # Ordena los artículos en función de las razones
    ratios.sort(reverse=True)

    total_valor = 0
    mochila = []

    for ratio, (valor, peso) in ratios:
        if capacidad >= peso:
            # Si cabe el artículo completo, lo añade a la mochila
            mochila.append((valor, peso))
            total_valor += valor
            capacidad -= peso
        else:
            # Si no cabe completo, añade una fracción del artículo
            fraccion = capacidad / peso
            mochila.append((valor * fraccion, peso * fraccion))
            total_valor += valor * fraccion
            break  # Termina el bucle

    return total_valor, mochila
```

Resultados:

Se realizaron pruebas con diferentes conjuntos de elementos y capacidades de mochila para evaluar el rendimiento del algoritmo voraz.

- **Ejemplo 1:**
    - Se aplicó el algoritmo a un conjunto de elementos y capacidad de mochila proporcionados.
    - Se obtuvo el valor máximo que se puede llevar en la mochila, junto con la lista de elementos fraccionarios seleccionados.
- **Ejemplo 2:**
    - Se realizaron pruebas adicionales con otro conjunto de elementos y capacidad de mochila.
    - Se observó cómo el algoritmo seleccionó diferentes elementos de manera fraccionaria para maximizar el valor total dentro de la capacidad especificada.

Punto de entrada de ejecución de la función de la mochila fraccionaria:

mochila_voraz.py > ...

```python
if __name__ == "__main__":
    items = [(10, 2), (20, 5), (30, 10)]
    capacidad= 15

    print(f'''

          Mochila fraccionaria Ejemplo 1:

          ''')
    total_valor, mochila = mochila_fraccionaria(items, capacidad)
    print("Valor total en la mochila:", total_valor)
    print("Contenido de la mochila:")
    for valor, peso in mochila:
        print(f"  Valor: {valor}, Peso: {peso}")

    items_2 = [(25, 5), (40, 10), (35, 8)]
    capacidad_2= 20

    print(f'''

          Mochila fraccionaria Ejemplo 2:

          ''')

    total_valor_2, mochila_2 = mochila_fraccionaria(items_2, capacidad_2)
    print("Valor total en la mochila:", total_valor_2)
    print("Contenido de la mochila:")
    for valor, peso in mochila_2:
        print(f"  Valor: {valor}, Peso: {peso}")
```

Resultados de la ejecución del programa:

```
arturo  master  ~/Documents/ESCOM/algoritmos_escom_2024/practica_lab_5
oritmos_escom_2024/practica_lab_5/mochila_voraz.py


          Mochila fraccionaria Ejemplo 1:


Valor total en la mochila: 54.0
Contenido de la mochila:
  Valor: 10, Peso: 2
  Valor: 20, Peso: 5
  Valor: 24.0, Peso: 8.0


          Mochila fraccionaria Ejemplo 2:


Valor total en la mochila: 88.0
Contenido de la mochila:
  Valor: 25, Peso: 5
  Valor: 35, Peso: 8
  Valor: 28.0, Peso: 7.0
```

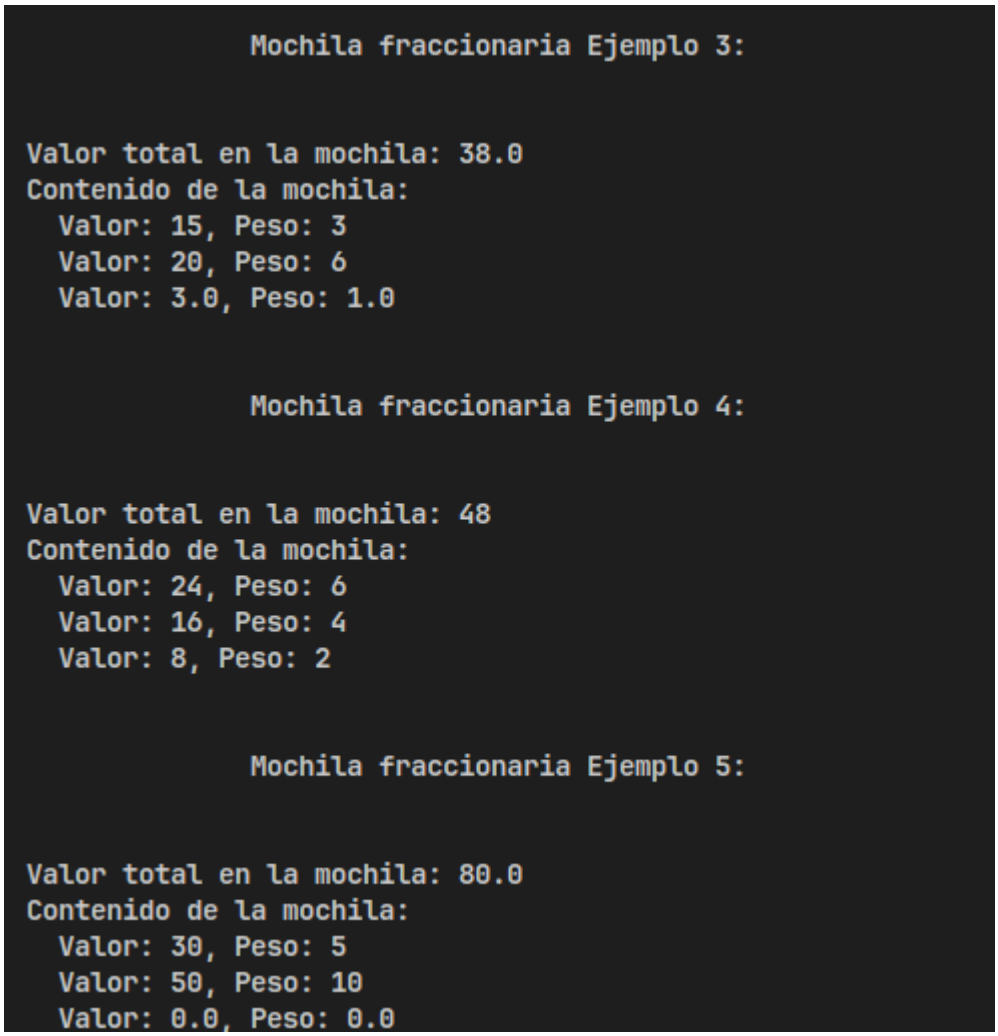
Ejemplos extra:

Datos de entrada

mochila_voraz.py > ...

```
55
56      # Otros 3 ejemplos
57      items_3 = [(15, 3), (12, 4), (20, 6)]
58      capacidad_3 = 10
59
60      items_4 = [(8, 2), (16, 4), (24, 6)]
61      capacidad_4 = 12
62
63      items_5 = [(30, 5), (40, 8), (50, 10)]
64      capacidad_5 = 15
```

resultados:



Conclusiones:

Las metodologías voraces son herramientas poderosas para resolver problemas de optimización, ofreciendo soluciones rápidas y simples. Sin embargo, su aplicación requiere considerar cuidadosamente el equilibrio entre eficiencia y calidad de la solución. La elección del criterio voraz adecuado depende de las características del problema y del conjunto de datos. Es esencial analizar detalladamente los resultados obtenidos y considerar la posibilidad de combinar metodologías voraces con otros enfoques para mejorar la calidad de la solución. En resumen, las metodologías voraces son valiosas, pero su uso efectivo requiere un enfoque estratégico y una evaluación exhaustiva de los resultados.