Instituto Politécnico Nacional

Escuela Superior de Cómputo

Unidad de académica:

Análisis y diseño de algoritmos

Práctica 6:

“Algoritmos Voraces”

Alumno:

Solares Velasco Arturo Misael 2023630538

Solis Lugo Mayra 2023630449

Grupo: 3CV1

Profesora: Floriano García Andres

Fecha:

29 abril 2024

Reporte de Práctica: Algoritmos Voraces Asignación de memoria y captura de ladrones.

Asignación de bloques de memoria

El problema de asignación de bloques de memoria consiste en asignar procesos a bloques de memoria disponibles de manera que se maximice la eficiencia de la asignación. El algoritmo voraz implementado para resolver este problema sigue un enfoque simple pero efectivo. Para cada proceso, busca el bloque de memoria disponible más grande que pueda contener el proceso y lo asigna a dicho bloque. Este proceso se repite para todos los procesos, asegurando que se asignen los bloques de memoria de manera óptima.

El algoritmo en código python es el siguiente:

memory_ram_asignation.py > asignacion_bloques_caso2

```python
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
```

Casos de prueba:

Caso 1:

Bloques de memoria: [100, 500, 200, 300, 600]

Procesos: [212, 417, 112, 426]

Asignaciones: [(212, 1), (417, 1), (112, 2), (426, 4)]

Resultados

```
Caso 1:
    bloques_memoria = [100, 500, 200, 300, 600]
    procesos = [212, 417, 112, 426]
    Asignaciones: [(212, 3), (417, 1), (112, 2), (426, 4)]
```

Caso 2:

Bloques de memoria: [500, 600, 200, 300, 100]

Procesos: [456, 354, 123, 567]

Asignaciones: [(456, 1), (354, 0), (123, 2), (567, 1)]

Resultados

```
Caso 2:
    bloques_memoria = [500,600,200,300,100]
    procesos = [456,354,123,567]
    Asignaciones: [(456, 0), (354, 1), (123, 2), (567, 3)]
```

Caso 3:

Bloques de memoria: [100, 500, 200, 300, 600]

Procesos: [100, 150, 500, 200, 300, 600]

Asignaciones: [(100, 0), (150, 0), (500, 1), (200, 2), (300, 3), (600, 4)]

Resultados

```
Caso 3:
    bloques_memoria = [100, 500, 200, 300, 600]
    procesos = [100,150,500,200,300,600]
    Asignaciones: [(100, 0), (150, 2), (500, 1), (200, 3), (300, 4), (600, 4)]
```

Algoritmo de captura de ladrones

El problema de la captura de ladrones consiste en determinar el número máximo de ladrones que pueden ser capturados dadas ciertas restricciones, como la distancia máxima permitida entre un policía y un ladrón. El algoritmo voraz implementado para este problema sigue un enfoque basado en recorrer la lista de ladrones y policías, asignando a cada policía al ladrón más cercano dentro del rango permitido. Este proceso se repite hasta que se haya considerado a todos los ladrones.

El algoritmo en código python es el siguiente:

policias_ladrones.py > ...

```python
def capturar_ladrones(arr, k):
    n = len(arr)
    policia = ladron = 0
    asignaciones = 0

    while policia < n and ladron < n:
        # Si el ladron está dentro del rango de captura del policía
        if abs(ladron - policia) <= k:
            asignaciones += 1
            # Encontrar el próximo policía y ladron
            next_index = max(policia, ladron) + 1
            while next_index < n and arr[next_index] != ('P' if arr[policia] == 'P' else 'L'):
                next_index += 1
            if next_index == n:
                break
            policia = ladron = next_index
        else:
            # Incrementar el índice del próximo policía o ladron
            min_index = min(policia, ladron)
            next_index = min_index + 1
            while next_index < n and arr[next_index] != ('P' if arr[policia] == 'P' else 'L'):
                next_index += 1
            if next_index == n:
                break
            policia = ladron = next_index

    return asignaciones
```

Ejemplos de uso:

Ejemplo 1:

Lista de personajes: ['P', 'L', 'L', 'P', 'L', 'P']

Distancia máxima permitida: 1

Número de ladrones capturados: 3

```
Ejemplo 1:
Número de ladrones capturados: 3
```

Ejemplo 2:

Lista de personajes: ['P', 'L', 'L', 'P', 'L', 'P', 'L', 'L', 'P', 'P', 'L', 'P', 'L', 'P']

Distancia máxima permitida: 2

Número de ladrones capturados: 7

```
Ejemplo 2:
Número de ladrones capturados: 7
```

Ejemplo 3:

Lista de personajes: ['P', 'L', 'L', 'P', 'L', 'P', 'L', 'L', 'P', 'P', 'L', 'P', 'L', 'P']

Distancia máxima permitida: 3

Número de ladrones capturados: 7

```
Ejemplo 3:
Número de ladrones capturados: 7
```

Ejemplo 4:

Lista de personajes: ['P', 'P', 'P', 'P', 'P', 'L', 'P', 'L', 'L', 'P', 'P', 'L', 'P']

Distancia máxima permitida: 4

Número de ladrones capturados: 9

```
Ejemplo 4:
Número de ladrones capturados: 9
```

conclusión

En ambos casos, los algoritmos voraces proporcionan soluciones eficientes que cumplen con los requisitos del problema y pueden ser aplicados en una variedad de situaciones prácticas. Sin embargo, es importante tener en cuenta las limitaciones y restricciones de cada algoritmo al seleccionar la mejor estrategia para resolver un problema específico.