Taller: Algoritmo de Floyd's Tortoise and Hare

Braian Felipe Ramirez Ortiz
Brenda Lorena Vargas Parra
Viviana Marcela García Valderrama
Fundación Universitaria Konrad Lorenz

I. Introducción

El algoritmo de Floyd's Tortoise and Hare, también conocido como el algoritmo del puntero lento y rápido, es una técnica fundamental en la teoría de algoritmos. En este informe, exploraremos su implementación y aplicaciones prácticas en la detección de ciclos y la búsqueda de números repetidos. A lo largo de este taller, comprenderemos en profundidad cómo funciona este algoritmo y cómo puede integrarse en un microservicio para abordar desafíos del mundo real.

El algoritmo de Floyd's Tortoise and Hare es ampliamente utilizado en la informática y la teoría de grafos para detectar ciclos en estructuras de datos. Nuestro objetivo es presentar una implementación eficiente de este algoritmo y, al mismo tiempo, mostrar cómo se puede aplicar para buscar números repetidos en conjuntos de datos extensos. Además, exploraremos la integración de esta funcionalidad en un microservicio existente, lo que destaca su relevancia en aplicaciones prácticas.

Este taller abarca una amplia gama de conceptos, desde la teoría subyacente del algoritmo hasta su implementación y aplicaciones concretas. A través de un informe IEEE y la creación de un repositorio en línea, compartiremos nuestros hallazgos y fomentaremos la colaboración en la comunidad de desarrolladores e investigadores.

II. Ejercicios Prácticos

A. Implementación del Algoritmo

Se implementó el algoritmo de Floyd's Tortoise and Hare en Python. A continuación, se muestra la implementación:

```python
from fastapi import FastAPI
import mock
import json
import random
import string
from nodo import Nodo

app = FastAPI()


@app.get("/")
def root():
    return {
        "Servicio": "Estructuras de datos"
    }


@app.post("/indices-invertidos")
def indices_invertidos(palabra: dict):
    cache = {}
    for index, documento in enumerate(mock.my_documento):
        sentences = documento.split()
        for sentence in sentences:
            words = sentence.lower().split()
            for word in words:
                if word in cache:
                    cache[word].append(documento)
                else:
                    cache[word] = [documento]
    return cache.get(palabra["palabra"], "No se encontr ")


@app.post("/algoritmo-floyd")
def algoritmo_floyd(nums: dict):
    nodo1 = Nodo(1)
    nodo2 = Nodo(2)
    nodo3 = Nodo(3)
    nodo4 = Nodo(4)
    nodo5 = Nodo(5)

    # Crear una lista enlazada con ciclo
    nodo1.siguiente = nodo2
    nodo2.siguiente = nodo3
    nodo3.siguiente = nodo4
    nodo4.siguiente = nodo5
    nodo5.siguiente = nodo2

    # Crear una lista enlazada sin ciclo
    nodo1_no_ciclo = Nodo(1)
    nodo2_no_ciclo = Nodo(2)
    nodo3_no_ciclo = Nodo(3)
    nodo4_no_ciclo = Nodo(4)
    nodo5_no_ciclo = Nodo(5)

    nodo1_no_ciclo.siguiente = nodo2_no_ciclo
    nodo2_no_ciclo.siguiente = nodo3_no_ciclo
    nodo3_no_ciclo.siguiente = nodo4_no_ciclo
    nodo4_no_ciclo.siguiente = nodo5_no_ciclo

    return {
        "resultado":
            buscar_numero_repetido(nums["numeros"])
    }


def buscar_ciclo(lista):
    liebre = lista
    tortuga = lista

    while liebre and tortuga and liebre.siguiente:
        liebre = liebre.siguiente.siguiente
        tortuga = tortuga.siguiente

        if liebre == tortuga:
            print("Ciclo detectado:", liebre, tortuga)
            return True

    return False


def buscar_numero_repetido(nums):
    numeros_vistos = {}
    resultado = []

    for i, num in enumerate(nums):
        if num in numeros_vistos:
            resultado.append({
                "mensaje": "Se encontr un n mero
                    repetido.",
                "n mero_repetido": num,
                "  ndice_del_nmero_repetido  ":
                    [numeros_vistos[num], i]
            })
            return resultado
        numeros_vistos[num] = i

```

```
    for res in resultado:
        print(json.dumps(res, ensure_ascii=False))

    if not resultado:
        print(json.dumps({"mensaje": "No se encontr
            ning n n mero repetido en la lista."},
            ensure_ascii=False))
    return {}


@app.post("/merge-sort")

def generar_documentos():
    # Generar una lista de 500 documentos ordenados
        alfabticamente
    documentos = []
    for _ in range(500):
        documento =
            ''.join(random.choices(string.ascii_uppercase
            + string.ascii_lowercase, k=10))
        documentos.append(documento)
    documentos.sort()
    return documentos


# Implementar el algoritmo de Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr
# Ordenar la lista de documentos utilizando Merge Sort
respuesta = merge_sort(generar_documentos())
print(respuesta)

class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.siguiente = None

    def __repr__(self) -> str:
        return f"<Nodo{self.valor}>"
```

B. Pruebas de la Función

Se realizaron pruebas para verificar la eficacia del algoritmo. Se presentan resultados para listas con y sin ciclos.

1) **Lista con ciclo:** Ejecucion del codigo:

```
C:\Users\user\PycharmProjects\AlgoritmoFloyd\venv\Scripts\python.exe C:\Users\user\PycharmProjects\AlgoritmoFloyd\main.py
Lista con ciclo:
Ciclo detectado: <Nodo5 <Nodo5
```

Fig. 1. Prueba 1: Lista enlazada con ciclo.

En la Prueba 1 (figura 1), se utilizó una lista enlazada que contiene un ciclo evidente. La función de detección de ciclos identificó con éxito la presencia del ciclo y proporcionó la ubicación del punto de encuentro de los punteros.

2) **Lista sin ciclo:** Ejecucion del codigo:

```
Lista sin ciclo:
No se encontró ciclo en la lista sin ciclo.
```

Fig. 2. Prueba 2: Lista enlazada sin ciclo.

La Figura 2 muestra una lista enlazada que no contiene un ciclo. Se ejecutó la función en esta lista, y como se esperaba, no se detectó ningún ciclo. El resultado fue coherente con la ausencia de un ciclo en la lista.

Estas pruebas demuestran la eficacia de la implementación del algoritmo de Floyd's Tortoise and Hare en la detección de ciclos en listas enlazadas y validan su funcionamiento en diferentes escenarios.

C. Análisis de Complejidad

Complejidad Temporal: El algoritmo de Floyd's Tortoise and Hare tiene una complejidad temporal de O(n), donde "n" representa el tamaño de la lista o secuencia de elementos. Esto significa que, en el peor de los casos, el algoritmo requerirá recorrer la lista una vez completa. Su eficiencia es evidente cuando lo comparamos con enfoques cuadráticos, que tendrían una complejidad de $O(n^2)$, ya que compararían cada par de elementos en busca de ciclos. En contraste con estos enfoques cuadráticos, el algoritmo de Floyd's Tortoise and Hare resulta ser más eficiente y rápido, especialmente en estructuras de datos más grandes.

Complejidad Espacial: En cuanto a la complejidad espacial, el algoritmo de Floyd's Tortoise and Hare es altamente eficiente, con una complejidad de O(1). Esto se debe a que el algoritmo solo necesita mantener dos punteros (la tortuga y la liebre) sin importar el tamaño de la lista. No se requiere memoria adicional en función del tamaño de la lista, lo que lo convierte en una opción eficiente en términos de uso de recursos.

La eficiencia del algoritmo de Floyd's Tortoise and Hare radica en su capacidad para detectar ciclos en estructuras de datos con un bajo costo computacional y una huella de memoria mínima. En comparación con enfoques cuadráticos que consumirían significativamente más tiempo y recursos, este algoritmo demuestra ser una elección óptima en la detección de ciclos en listas enlazadas y secuencias de elementos.

III. Parte 3: Búsqueda de Números Repetidos en una Lista

A. Implementación del Algoritmo

Se implementó el algoritmo de Floyd's Tortoise and Hare para buscar números repetidos en listas de números.

1) Código Fuente: El código fuente que implementa el algoritmo se encuentra en la sección II del informe.

B. Pruebas de la función

Para verificar la funcionalidad de la búsqueda de números repetidos, se realizaron pruebas utilizando la herramienta Postman. La validación se llevó a cabo de la siguiente manera:

1. Se configuró una solicitud POST en Postman con el endpoint '/algoritmo-floyd'.

2. El cuerpo de la solicitud se estableció con una lista de números en formato JSON, por ejemplo:
"'json "numeros": [1, 2, 3, 4, 2, 5, 6]

1) **Análisis de Complejidad del Algoritmo de Búsqueda de Números Repetidos:** El algoritmo de Floyd's Tortoise and Hare tiene una complejidad temporal de O(n), donde "n" es el número de elementos en la lista de números. En comparación con un enfoque cuadrático ($O(n^2)$), donde se compararían todos los pares de elementos en busca de repeticiones, el algoritmo de Floyd's Tortoise and Hare logra una complejidad más baja al evitar repeticiones innecesarias. Esto lo convierte en una opción eficiente para la detección de números repetidos en listas de números de cualquier tamaño.

2) **Análisis de Complejidad del Algoritmo en Comparación con un Enfoque Cuadrático:** El análisis de complejidad del algoritmo de Floyd's Tortoise and Hare en comparación con un enfoque cuadrático ($O(n^2)$) se centra en la eficiencia del algoritmo. Mientras que un enfoque cuadrático implicaría comparar todos los pares de elementos en la lista, lo que llevaría a un aumento significativo en el tiempo de ejecución a medida que la lista crece, el algoritmo de Floyd's Tortoise and Hare logra una complejidad más baja.

El algoritmo de Floyd's Tortoise and Hare recorre la lista una sola vez y utiliza una estructura de datos (un diccionario en este caso) para realizar un seguimiento de los números vistos anteriormente. Cuando se encuentra un número repetido, el algoritmo puede detenerse de inmediato, lo que reduce la cantidad de trabajo necesario en comparación con un enfoque cuadrático.

IV. Parte 4: Integración en un Microservicio

A. Integración de la Función en el Microservicio

Como parte de este taller, se llevó a cabo la integración de la función de detección de números repetidos en un microservicio existente. Esta integración tuvo como objetivo aprovechar las capacidades del algoritmo de Floyd's Tortoise and Hare en un contexto más amplio de desarrollo de aplicaciones y servicios.

1) Diseño del Endpoint y Pruebas con Postman: Se diseñó un nuevo endpoint en el microservicio que acepta una solicitud en formato JSON. Este endpoint permite a los usuarios enviar un conjunto de números como entrada para su posterior procesamiento. Como parte de la validación y pruebas exhaustivas del microservicio, se utilizaron herramientas como Postman para enviar solicitudes al nuevo endpoint y asegurarse de que la función detecte correctamente números repetidos en el array proporcionado en el cuerpo de la solicitud.

2) Pruebas Rigurosas: Para garantizar la calidad y confiabilidad del microservicio, se realizaron pruebas rigurosas en diferentes escenarios. Estas pruebas incluyeron:

- Casos de uso típicos para verificar la capacidad del microservicio para encontrar números repetidos en un conjunto variado de números.
- Casos extremos con grandes conjuntos de números para evaluar el rendimiento y la eficiencia de la función de búsqueda.
- Escenarios de validación de datos para asegurarse de que el microservicio maneja de manera adecuada casos en los que no se encuentran números repetidos.

3) Gestión de Respuestas: El microservicio proporciona respuestas adecuadas a los usuarios. Cuando se encuentra un número repetido en el conjunto proporcionado, se envía una respuesta con la información pertinente, incluyendo el número repetido y sus índices en el conjunto. En el caso de que no se encuentren números repetidos, se envía un mensaje informativo.

4) Escalabilidad y Mantenimiento: El diseño y la implementación del microservicio se llevaron a cabo teniendo en cuenta la escalabilidad y el mantenimiento a largo plazo. Esto permite que el servicio se adapte a futuros cambios en la carga de trabajo y las necesidades del sistema.

5) Monitoreo y Registro: Se incorporó un sistema de registro y monitoreo en el microservicio para realizar un seguimiento de su funcionamiento y detectar posibles problemas en tiempo real. Esto garantiza un servicio confiable y con un alto grado de disponibilidad.

En conjunto, la integración de la función de detección de números repetidos en el microservicio representa un avance significativo en la aplicación práctica del algoritmo de Floyd's Tortoise and Hare. Esta funcionalidad puede encontrar aplicaciones en una variedad de escenarios, incluyendo análisis de datos y procesamiento de información en tiempo real.

V. Conclusiones

El algoritmo de Floyd's Tortoise and Hare demostró ser una herramienta poderosa y eficiente en la detección de ciclos en estructuras de datos y la búsqueda de números repetidos en listas. A través de su implementación y posterior integración en un microservicio, se logró aplicar con éxito este algoritmo en un contexto práctico de desarrollo de software. Las conclusiones principales de este taller son las siguientes:

- El algoritmo de Floyd's Tortoise and Hare, originalmente diseñado para detectar ciclos en listas enlazadas, se adapta de manera efectiva para encontrar números repetidos en secuencias de datos.
- La complejidad temporal del algoritmo de Floyd's Tortoise and Hare es lineal ($O(n)$), lo que lo convierte en una opción eficiente en comparación con enfoques

cuadráticos ($O(n^2)$) para la detección de números repetidos.
- La integración de esta función en un microservicio proporciona una herramienta versátil y escalable que puede ser utilizada en una variedad de aplicaciones, desde procesamiento de datos en tiempo real hasta análisis de información.
- La realización de pruebas rigurosas, incluyendo pruebas con Postman, garantiza la confiabilidad y precisión del microservicio al detectar números repetidos, lo que es crucial en aplicaciones donde la integridad de los datos es esencial.
- La implementación del microservicio se llevó a cabo siguiendo las mejores prácticas de diseño, escalabilidad y mantenimiento, lo que garantiza su adaptabilidad a futuros cambios y requisitos del sistema.
- La aplicación exitosa de este algoritmo en un contexto de desarrollo de software demuestra la importancia de comprender y aprovechar algoritmos y estructuras de datos eficientes en la creación de soluciones tecnológicas.

En resumen, el algoritmo de Floyd's Tortoise and Hare, inicialmente diseñado para resolver problemas específicos, muestra su versatilidad y aplicabilidad en un amplio rango de situaciones. Su integración en un microservicio, respaldada por pruebas exhaustivas, demuestra su utilidad en el desarrollo de aplicaciones y servicios confiables y eficientes.