

Taller 7

Jose David Ramirez Beltran - 506222723

14 de octubre de 2023

1. Introducción

Clase Nodo

La clase `Nodo` se utiliza para definir un nodo en una lista enlazada. Cada nodo consta de dos atributos: `valor`, que almacena el valor del nodo, y `siguiente`, un puntero al siguiente nodo en la lista.

Creación de Nodos y Ciclos

En este código, se crean tres nodos denominados `nodo1` a `nodo3`. Luego, se establece un ciclo en la lista enlazada haciendo que el último nodo (`nodo5`) apunte al segundo nodo (`nodo2`), creando así una lista enlazada circular.

Parte 2

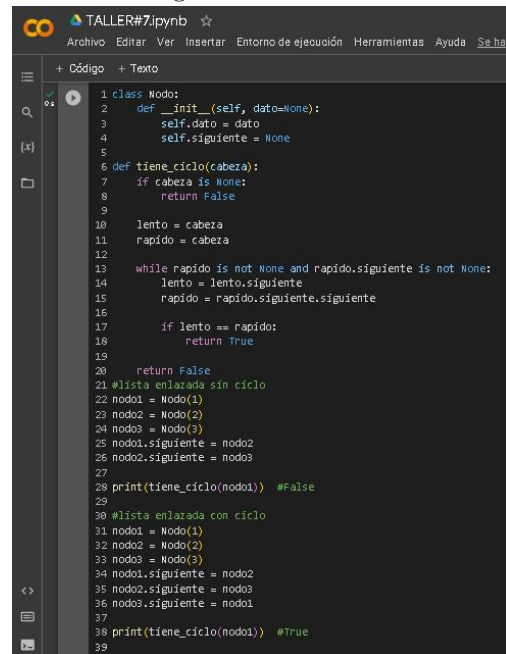
El algoritmo de Floyd's Tortoise and Hare, también conocido como el algoritmo del ciclo de detección, es una solución eficiente para encontrar ciclos en una lista enlazada.

La complejidad temporal del algoritmo es $O(n)$, ya que n es el número total de nodos en la lista y esto es porque cada nodo se visita a lo sumo dos veces y en cuanto a la complejidad espacial, es $O(1)$, ya que solo se utilizan dos punteros (tortuga y liebre), independientemente del tamaño de la lista enlazada.

Y respondiendo a la pregunta, en comparación con otros métodos para detectar ciclos en listas enlazadas, como el uso de un conjunto de datos hash para almacenar nodos visitados (que la complejidad espacial es de $O(n)$), el algoritmo de Floyd es más eficiente en términos de uso de memoria. Sin embargo, si la lista es extremadamente larga y el ciclo está cerca del final, puede ser más lento que el método del conjunto hash que puede detectar un ciclo tan pronto como se visita un nodo por segunda vez.

2. Desarrollo

Para llevar el desarrollo del algoritmo, se puede plantear de la siguiente manera:



```
1 class Nodo:
2     def __init__(self, dato=None):
3         self.dato = dato
4         self.siguiente = None
5
6     def tiene_ciclo(cabeza):
7         if cabeza is None:
8             return False
9
10        lento = cabeza
11        rapido = cabeza
12
13        while rapido is not None and rapido.siguiente is not None:
14            lento = lento.siguiente
15            rapido = rapido.siguiente.siguiente
16
17            if lento == rapido:
18                return True
19
20        return False
21
22 #lista enlazada sin ciclo
23 nodo1 = Nodo(1)
24 nodo2 = Nodo(2)
25 nodo3 = Nodo(3)
26 nodo1.siguiente = nodo2
27 nodo2.siguiente = nodo3
28 print(tiene_ciclo(nodo1)) #False
29
30 #lista enlazada con ciclo
31 nodo1 = Nodo(1)
32 nodo2 = Nodo(2)
33 nodo3 = Nodo(3)
34 nodo1.siguiente = nodo2
35 nodo2.siguiente = nodo3
36 nodo3.siguiente = nodo1
37
38 print(tiene_ciclo(nodo1)) #True
39
```

3. Funcionamiento

El código funciona de la siguiente manera: primero, se define una clase `Nodo` que tiene dos atributos: `dato` y `siguiente`. El atributo `dato` es el valor que se almacena en el nodo y `siguiente` es una referencia al siguiente nodo en la lista enlazada.

La función `tiene_ciclo(cabeza)` toma como argumento la cabeza de la lista enlazada y devuelve "True" si la lista tiene un ciclo y "False" si no lo tiene. Dentro de esta función se utilizan dos punteros, `bicicleta` y `moto`, que recorren la lista enlazada a diferentes velocidades; `bicicleta` avanza un nodo

```

17     if lento == rapido:
18         return True
19
20     return False
21 #lista enlazada sin ciclo
22 nodo1 = Nodo(1)
23 nodo2 = Nodo(2)
24 nodo3 = Nodo(3)
25 nodo1.siguiente = nodo2
26 nodo2.siguiente = nodo3
27
28 print(tiene_ciclo(nodo1)) #False
29
30 #lista enlazada con ciclo
31 nodo1 = Nodo(1)
32 nodo2 = Nodo(2)
33 nodo3 = Nodo(3)
34 nodo1.siguiente = nodo2
35 nodo2.siguiente = nodo3
36 nodo3.siguiente = nodo1
37
38 print(tiene_ciclo(nodo1)) #True
39

```

False
True

Figura 1: Diagrama de funcionamiento

a la vez, mientras que moto avanza dos nodos a la vez.

Si la lista enlazada tiene un ciclo, eventualmente bicicleta y moto se encontrarán en el mismo nodo. Si esto sucede, la función devuelve True. Esto se puede observar en la ejecución del segundo caso.

Si moto llega al final de la lista (es decir, encuentra un nodo cuyo atributo siguiente es None), entonces sabemos que la lista no tiene un ciclo y la función devuelve False. Esto se puede observar cuando se imprime el primer caso.

Finalmente, el código crea dos listas enlazadas, una con un ciclo y otra sin él, y prueba la función `tiene_ciclo(cabeza)` con ambas. Como se esperaba, devuelve “False” para la lista sin ciclo y “True” para la lista con ciclo.

4. Parte 3 Búsqueda de Números Repetidos en una Lista

El código se llevó a cabo de la siguiente manera:

```

1 def encontrar_duplicado(numeros):
2
3     tortuga = numeros[0]
4     liebre = numeros[0]
5     while True:
6         tortuga = numeros[tortuga]
7         liebre = numeros[numeros[liebre]]
8         if tortuga == liebre:
9             break
10
11     # Fase de búsqueda
12     puntero1 = numeros[0]
13     puntero2 = tortuga
14     while puntero1 != puntero2:
15         puntero1 = numeros[puntero1]
16         puntero2 = numeros[puntero2]
17
18     return puntero1
19
20 #lista de números con al menos un número repetido
21 numeros = [3, 1, 3, 4, 2]
22
23 # Encontrar el número repetido en la lista
24 print(encontrar_duplicado(numeros))
25

```

3

El primer paso es la fase de la Tortuga y la Liebre. En esta fase, se inicializan dos punteros, la tortuga y la liebre, al primer elemento de la lista. Luego entramos en un bucle donde la tortuga avanza un paso a la vez (`tortuga = numeros[tortuga]`) y la liebre avanza dos pasos a la vez (`liebre = numeros[numeros[liebre]]`). Este bucle continúa hasta que la tortuga y la liebre se encuentran, lo que indica que hay un número duplicado en la lista.

A continuación, está la fase de búsqueda. Una vez que hemos encontrado un punto de encuentro con la tortuga y la liebre, inicializamos un nuevo puntero (`puntero1`) al primer elemento de la lista y mantenemos otro puntero (`puntero2`) en el punto de encuentro. Luego avanzamos `puntero1` y `puntero2` un paso a la vez hasta que se encuentren. El punto donde se encuentran es el número duplicado en la lista (3).

Respondiendo a la pregunta, la complejidad temporal del algoritmo de la Tortuga y la Liebre de Floyd es lineal, es decir, $O(n)$, donde n es el número de elementos en la lista. Esto es mucho más eficiente que un enfoque cuadrático $O(n^2)$ que compara cada par de elementos.

El algoritmo de “Floyd’s Tortoise and Hare” logra una complejidad más baja al evitar comparaciones innecesarias. En lugar de comparar cada par de elementos, este algoritmo utiliza dos punteros que se mueven a diferentes velocidades a través de la lista. Cuando estos dos punteros se encuentran, sabemos que hay un número repetido. Este enfoque garantiza que cada elemento se visite solo una vez, lo que resulta en una complejidad temporal lineal.

5. Conclusiones

El código proporcionado es una manera efectiva para detectar ciclos en una lista enlazada, ya que utiliza el enfoque de dos punteros que avanzan a diferentes velocidades, comúnmente conocido como el algoritmo del corredor rápido y lento. Este algoritmo es eficiente ya que solo necesita recorrer la lista una vez, con una complejidad de tiempo de $O(n)$, donde n es el número de nodos en la lista. La detección de ciclos en listas enlazadas es un problema común en ciencias de la computación y este código ofrece una solución sólida y eficiente para resolverlo, pero como en todo es eficiente dependiendo de lo que nos pidan o de lo que tenga que almacenar o recorrer el código.

6. Bibliografía

<https://github.com/jOsE0134/Taller7.git>

techiedelight.com/es/detect-cycle-linked-list-floyds-cycle-detection-algorithm/

https://colab.research.google.com/drive/1NRFB1d7gyfXCCQkQSa_nFs12OCuWK?hl=es