



Nombre: Yannick Perez

NRC: 13930

RESUMEN VIDEOS ALGORITMOS

1. Algoritmo de Bubble Sort

El video aborda tres temas principales: algoritmos de ordenamiento, algoritmos de búsqueda y algoritmos para recorridos en gráficas. Se comienza con la importancia de los algoritmos de ordenamiento en computación, destacando el "ordenamiento de la burbuja" como un ejemplo. Se explica el funcionamiento del algoritmo, que implica comparar pares de elementos consecutivos y realizar intercambios según su magnitud. Se proporciona un ejemplo práctico de cómo aplicar este algoritmo para ordenar una lista de números.

1. Algoritmo de Bubble Sort en Python

El texto guía en la implementación del algoritmo de ordenamiento "Bubble Sort" en Python, utilizando un editor de código como Visual Studio Code. Se detalla la creación de la función llamada "bubble_sort" que recibe una lista, seguida de dos bucles for anidados para comparar y realizar intercambios entre elementos consecutivos. Se sugiere probar la función con una lista desordenada e imprimir tanto la lista original como la ordenada. Además, se menciona la complejidad del algoritmo y se invita al lector a realizar modificaciones para mejorar su rendimiento, sugiriendo optimizar el código para evitar comparaciones innecesarias.

2. Algoritmo de Merge Sort

En la segunda lección del curso de algoritmos de búsqueda y ordenamiento en Python, se introduce el algoritmo de ordenamiento "Merge Sort". El algoritmo sigue el enfoque de "divide y vencerás", dividiendo el problema de ordenar una lista en partes más pequeñas y resolviéndolas para luego combinarlas en orden. Se presenta el pseudocódigo de las funciones "merge_sort" y "merge" y se explica cómo funcionan. Luego, se realiza un ejercicio práctico para ordenar una lista de seis elementos desordenados utilizando el algoritmo "Merge Sort".

2. Algoritmo de Merge Sort en Python

En la programación del algoritmo Merge Sort en Python, se comienza definiendo la función principal "merge_sort". Se sigue el pseudocódigo visto anteriormente, con la implementación de la función "merge" que combina dos listas ordenadas. La función "merge_sort" verifica el caso base (longitud de la lista igual a 1) y luego divide la lista en dos partes, llamando recursivamente a "merge_sort" para cada parte. Se utiliza una función auxiliar "merge" para combinar las listas ordenadas resultantes. Se incluye un ejemplo de prueba con una lista desordenada e impresa en consola tanto la lista desordenada como la lista ordenada con Merge Sort.

3. Algoritmo de QuickSort

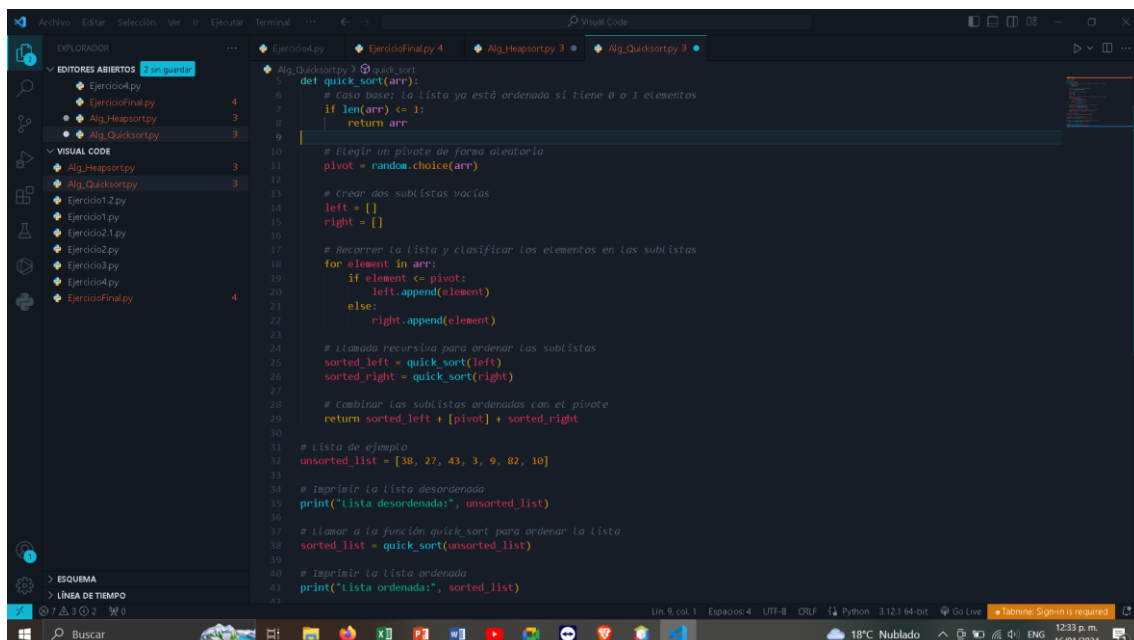
Quick Sort es un algoritmo de ordenamiento eficiente basado en el principio de "divide y vencerás". Funciona dividiendo el problema en subproblemas más pequeños y luego combinando las soluciones. La elección del pivote es crucial, y su mala elección puede llevar a

un rendimiento exponencial en el peor de los casos. El algoritmo consta de una función recursiva, utilizando un caso base cuando la longitud de la lista es 0 o 1. Se elige un pivote (en este caso, el último elemento), y se crean sublistas con elementos menores o iguales y mayores al pivote. Estas sublistas se ordenan recursivamente, y finalmente se combinan para obtener la lista ordenada. Se presenta un ejemplo paso a paso para ilustrar el proceso de Quick Sort en una lista desordenada.

3. Algoritmo de Quicksort en Python

continuemos con la implementación del algoritmo Quick Sort en Python utilizando una elección aleatoria del pivote. Abre tu editor de texto y sigue los pasos indicados a continuación:

- Crear un nuevo archivo: Abre tu editor de texto y crea un nuevo archivo. Dale un nombre significativo, por ejemplo, "quick_sort_random.py".
- Implementar el código: Copia y pega el siguiente código en tu archivo:



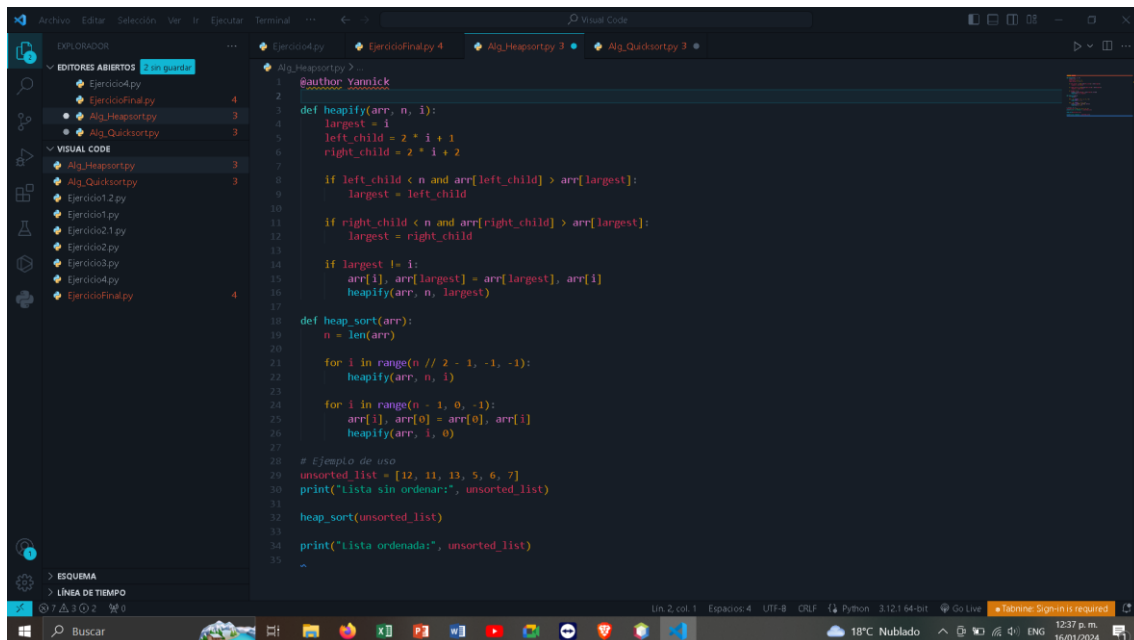
```
1 def quick_sort(arr):
2     # caso base: la lista ya está ordenada si tiene 0 o 1 elementos
3     if len(arr) <= 1:
4         return arr
5
6     # Elegir un pivote de forma aleatoria
7     pivot = random.choice(arr)
8
9     # Crear dos sublistas vacías
10    left = []
11    right = []
12
13    # Recorrer la lista y clasificar los elementos en las sublistas
14    for element in arr:
15        if element <= pivot:
16            left.append(element)
17        else:
18            right.append(element)
19
20    # llamada recursiva para ordenar las sublistas
21    sorted_left = quick_sort(left)
22    sorted_right = quick_sort(right)
23
24    # combinar las sublistas ordenadas con el pivote
25    return sorted_left + [pivot] + sorted_right
26
27 # lista de ejemplo
28 unsorted_list = [38, 27, 43, 3, 9, 82, 10]
29
30 # Imprimir la lista desordenada
31 print("Lista desordenada:", unsorted_list)
32
33 # llamar a la función quick_sort para ordenar la lista
34 sorted_list = quick_sort(unsorted_list)
35
36 # Imprimir la lista ordenada
37 print("Lista ordenada:", sorted_list)
```

- Probar el código: Guarda tu archivo y ejecútalo desde la terminal usando el comando `python quick_sort_random.py`. Deberías ver la lista desordenada y la lista ordenada.

Este código implementa Quick Sort con la elección aleatoria del pivote, lo que ayuda a evitar el peor caso en situaciones específicas.

4. Algoritmo de Heapsort

Hemos seguido el proceso paso a paso y hemos aplicado el algoritmo de Heap Sort para ordenar la lista dada. Es un buen ejercicio para comprender el funcionamiento de este algoritmo basado en montículos (heaps).



```
1 @author: Yannick
2
3 def heapify(arr, n, i):
4     largest = i
5     left_child = 2 * i + 1
6     right_child = 2 * i + 2
7
8     if left_child < n and arr[left_child] > arr[largest]:
9         largest = left_child
10
11     if right_child < n and arr[right_child] > arr[largest]:
12         largest = right_child
13
14     if largest != i:
15         arr[i], arr[largest] = arr[largest], arr[i]
16         heapify(arr, n, largest)
17
18 def heap_sort(arr):
19     n = len(arr)
20
21     for i in range(n // 2 - 1, -1, -1):
22         heapify(arr, n, i)
23
24     for i in range(n - 1, 0, -1):
25         arr[i], arr[0] = arr[0], arr[i]
26         heapify(arr, i, 0)
27
28 # Ejemplo de uso
29 unsorted_list = [12, 11, 13, 5, 6, 7]
30 print("Lista sin ordenar:", unsorted_list)
31
32 heap_sort(unsorted_list)
33
34 print("Lista ordenada:", unsorted_list)
35
```

Este código define dos funciones: `heapify` para convertir un subárbol en un montículo (heap), y `heap_sort` para realizar la ordenación utilizando Heap Sort. El ejemplo de uso muestra cómo utilizar estas funciones para ordenar una lista. Recuerda que Heap Sort ordena la lista in-place.

4. Algoritmo de Heapsort en Python

En este video, se programó el algoritmo Heap Sort en Python. A continuación, se presenta un resumen del código:

- Se define la función `hippie_fai` que recibe una lista como parámetro y el nodo desde el cual se aplicará el algoritmo.
- Se inicializa una lista final (`l2`) que actuará como la lista ordenada.
- Se itera desde $n/2 - 1$ hasta 0 utilizando una variable y que cambiará de valor en cada iteración.
- En cada iteración, se llama a la función `hippie_fai`, que modifica el orden de los nodos del árbol representado como una lista.
- Se itera desde 0 hasta la longitud de la lista menos 1, intercambiando el primer elemento con el último y agregando el menor elemento al final de `l2`.
- Finalmente, se aplica `hippie_fai` a la lista desde el nodo 0 y se retorna la lista final ordenada.

Además, se proporciona un ejemplo de uso donde se imprime una lista desordenada y luego la lista ordenada mediante el algoritmo Heap Sort.

El código puede ser probado guardándolo en un archivo llamado `heapsort.py` y ejecutándolo desde la terminal o el intérprete de Python.

A continuación, te proporciono un ejemplo de cómo implementar el algoritmo Heap Sort en Python según el resumen proporcionado. Este ejemplo incluirá la definición de la función `hippie_fai` y la aplicación del algoritmo a una lista de números desordenados:

```
1 @author: Yannick
2
3 def hippie_fai(lista, nodo):
4     n = len(lista)
5
6     # Verificar si el nodo tiene dos hijos
7     if 2 * nodo + 2 < n:
8         # Obtener el índice del hijo menor
9         hijo_menor = 2 * nodo + 1 if lista[2 * nodo + 1] < lista[2 * nodo + 2] else 2 * nodo + 2
10
11         # Intercambiar si el hijo menor es menor que el nodo padre
12         if lista[hijo_menor] < lista[nodo]:
13             lista[nodo], lista[hijo_menor] = lista[hijo_menor], lista[nodo]
14
15         # Aplicar recursivamente a partir del nodo intercambiado
16         hippie_fai(lista, hijo_menor)
17
18     # Verificar si el nodo tiene un hijo
19     elif 2 * nodo + 1 < n:
20         # Intercambiar si el hijo es menor que el nodo padre
21         if lista[2 * nodo + 1] < lista[nodo]:
22             lista[nodo], lista[2 * nodo + 1] = lista[2 * nodo + 1], lista[nodo]
23
24         # Aplicar recursivamente a partir del nodo intercambiado
25         hippie_fai(lista, 2 * nodo + 1)
26
27 def heap_sort(lista):
28     # Inicializar lista final
29     l2 = []
30
31     # Aplicar hippie_fai desde la mitad de la lista hasta el inicio
32     for y in range(len(lista)//2 - 1, -1, -1):
33         hippie_fai(lista, y)
34
35     # Iterar desde 0 hasta la longitud de la lista - 1
36     for y in range(len(lista)):
37         # Intercambiar primer elemento con el último
38         lista[0], lista[len(lista) - 1 - y] = lista[len(lista) - 1 - y], lista[0]
39
40         # Agregar el menor elemento al final de la lista final
41         l2.append(lista.pop())
42
43     # Aplicar hippie_fai a la lista desde el nodo 0
44     hippie_fai(lista, 0)
45
46     return l2
47
48 # Ejemplo de uso
49 lista_desordenada = [9, 4, 7, 1, 2, 8, 6]
50 print("Lista desordenada:", lista_desordenada)
51
52 # Aplicar Heap Sort
53 lista_ordenada = heap_sort(lista_desordenada)
54 print("Lista ordenada:", lista_ordenada)
55
56 ~
```

```
21 # Intercambiar si el hijo es menor que el nodo padre
22 if lista[2 * nodo + 1] < lista[nodo]:
23     lista[nodo], lista[2 * nodo + 1] = lista[2 * nodo + 1], lista[nodo]
24
25 # Aplicar recursivamente a partir del nodo intercambiado
26 hippie_fai(lista, 2 * nodo + 1)
27
28 def heap_sort(lista):
29     # Inicializar lista final
30     l2 = []
31
32     # Aplicar hippie_fai desde la mitad de la lista hasta el inicio
33     for y in range(len(lista)//2 - 1, -1, -1):
34         hippie_fai(lista, y)
35
36     # Iterar desde 0 hasta la longitud de la lista - 1
37     for y in range(len(lista)):
38         # Intercambiar primer elemento con el último
39         lista[0], lista[len(lista) - 1 - y] = lista[len(lista) - 1 - y], lista[0]
40
41         # Agregar el menor elemento al final de la lista final
42         l2.append(lista.pop())
43
44     # Aplicar hippie_fai a la lista desde el nodo 0
45     hippie_fai(lista, 0)
46
47     return l2
48
49 # Ejemplo de uso
50 lista_desordenada = [9, 4, 7, 1, 2, 8, 6]
51 print("Lista desordenada:", lista_desordenada)
52
53 # Aplicar Heap Sort
54 lista_ordenada = heap_sort(lista_desordenada)
55 print("Lista ordenada:", lista_ordenada)
56
57 ~
```

Este código define las funciones `hippie_fai` y `heap_sort`, y luego aplica el algoritmo Heap Sort a una lista de números desordenados. La lista resultante se imprime después de aplicar el algoritmo.

5. Algoritmo de Radix Sort

- Radix Sort: Es un algoritmo de ordenamiento que difiere de otros algoritmos al no considerar el valor absoluto de cada elemento, sino el valor de sus dígitos. Funciona ordenando primero por el dígito menos significativo y luego avanzando hacia el más significativo.
- Limitaciones del Algoritmo: Radix Sort solo funciona con números enteros y no puede manejar números negativos ni decimales, lo que lo coloca en desventaja frente a otros algoritmos de ordenamiento más flexibles.

- **Complejidad del Algoritmo:** La complejidad del algoritmo depende del número de elementos en la lista y del número de dígitos que componen cada número. La etapa de ordenamiento implica agregar ceros a la izquierda según sea necesario y realizar varios pasos de clasificación por dígitos.
- **Proceso de Radix Sort:** Inicia estableciendo la cantidad de dígitos del número más grande en la lista. Luego, agrega ceros a la izquierda de los elementos para igualar la cantidad de dígitos. En un ciclo, organiza los elementos en grupos según el dígito menos significativo, repite este proceso para los dígitos intermedios y finalmente para el dígito más significativo.
- **Ejemplo de Aplicación:** Se presenta un ejemplo práctico de cómo Radix Sort ordena un conjunto de números. Se identifica el mayor número de dígitos, se agrupan los elementos por dígito, y se repite el proceso hasta que los números estén completamente ordenados.

Aquí tienes un ejemplo simple de implementación del algoritmo Radix Sort en Python:

```

1  @author: Yannick
2
3  def radix_sort(lista):
4      # Encuentra el número de dígitos del número más grande
5      max_num = max(lista)
6      num_digits = len(str(max_num))
7
8      # Aplicar el algoritmo Radix Sort
9      for i in range(num_digits):
10         # Crear listas para cada dígito (0-9)
11         buckets = [[] for _ in range(10)]
12
13         # Organizar los elementos en los buckets según el dígito actual
14         for num in lista:
15             digit = (num // 10**i) % 10
16             buckets[digit].append(num)
17
18         # Reconstruir la lista con los elementos ordenados por el dígito actual
19         lista = [elemento for bucket in buckets for elemento in bucket]
20
21     return lista
22
23 # Ejemplo de uso
24 numeros_desordenados = [115, 102, 149, 58, 16, 3]
25 numeros_ordenados = radix_sort(numeros_desordenados)
26 print("Lista ordenada:", numeros_ordenados)
27

```

Este código define una función `radix_sort` que toma una lista de números y devuelve la lista ordenada utilizando el algoritmo Radix Sort. El ejemplo de uso muestra cómo ordenar una lista de números desordenados. Ten en cuenta que este ejemplo asume que los números son enteros no negativos.

5. Algoritmo de Radix Sort en Python

- **Implementación Inicial:** Se comienza creando una función de Radix Sort en Python que toma una lista como parámetro. Se determina el valor de 'n' como la máxima cantidad de dígitos entre los números de la lista.
- **Preparación de Datos:** La función requiere que los números en la lista sean cadenas en lugar de enteros. Esto permite acceder a cada dígito individualmente, ya que los datos de tipo entero no permiten este acceso directo.

- Normalización de Dígitos: Se agrega un cero a la izquierda de los números según sea necesario para asegurar que todos tengan la misma cantidad de dígitos.
- Agrupación por Dígito: Se crean grupos para cada dígito, y los números se asignan a estos grupos según el dígito en la posición actual. Luego, la lista principal se vacía y se rellena nuevamente con los números ordenados por sus dígitos.
- Conversión Final: Al final, la lista se convierte de nuevo a enteros y se retorna. La función se prueba con una lista aleatoria de diez números y devuelve la lista ordenada.

Aquí tienes un ejemplo sencillo en Python de cómo podrías implementar el algoritmo de Radix Sort utilizando la lógica descrita en el texto:

```

1  @author: Yarnick
2
3  def radix_sort(lista):
4      # Determinar el valor de 'n'
5      n = max(len(str(num)) for num in lista)
6
7      # Normalizar dígitos
8      lista = [str(num).zfill(n) for num in lista]
9
10     for j in range(n - 1, -1, -1):
11         # Crear grupos
12         grupos = [[] for _ in range(10)]
13
14         # Asignar números a grupos según el dígito actual
15         for num in lista:
16             grupos[int(num[j])].append(num)
17
18         # Vaciar y recargar la lista principal
19         lista = [num for grupo in grupos for num in grupo]
20
21     # Convertir la lista final a enteros y retornar
22     return [int(num) for num in lista]
23
24 # Ejemplo de uso
25 lista_ejemplo = [123, 25, 10, 456, 789, 1, 0, 987, 321, 654]
26 resultado = radix_sort(lista_ejemplo)
27
28 print("Lista original:", lista_ejemplo)
29 print("Lista ordenada:", resultado)
30

```

En este ejemplo, la función `radix_sort` toma una lista de números, realiza el algoritmo de Radix Sort y devuelve la lista ordenada. Ten en cuenta que este código está diseñado para entender el concepto, y en aplicaciones reales, es posible que desees considerar casos adicionales y optimizaciones.

6. Algoritmo de Búsqueda lineal

Introducción al Algoritmo de Búsqueda Lineal: El algoritmo de búsqueda lineal o secuencial es el tema principal, y se presenta como el método más fácil e intuitivo para buscar elementos en una lista.

Características de la Búsqueda Lineal: Se destaca que este algoritmo no requiere que la lista esté ordenada y es fácil de programar. Sin embargo, su desventaja radica en su posible lentitud en listas de gran tamaño.

Pseudo código del Algoritmo: Se muestra el pseudocódigo del algoritmo, que implica la comparación de cada elemento de la lista con el elemento buscado. Cuando se encuentra una

coincidencia, se devuelve "verdadero"; de lo contrario, se devuelve "falso" al final del recorrido de la lista.

Ejemplo de Funcionamiento: Se proporciona un ejemplo práctico donde se busca el número 25 en una lista. Se explica que el algoritmo compara cada elemento secuencialmente hasta encontrar una coincidencia, devolviendo "verdadero" si se encuentra y "falso" si se recorre toda la lista sin encontrar coincidencias.

Próximos Pasos: Se menciona que en el próximo video se programará este algoritmo para comprender su implementación práctica, aunque se subraya que el algoritmo en sí no es complicado.

6. Algoritmo de Búsqueda Lineal en Python

En el video, se programa el algoritmo de búsqueda lineal en Python, con el objetivo de buscar un elemento específico en una lista.

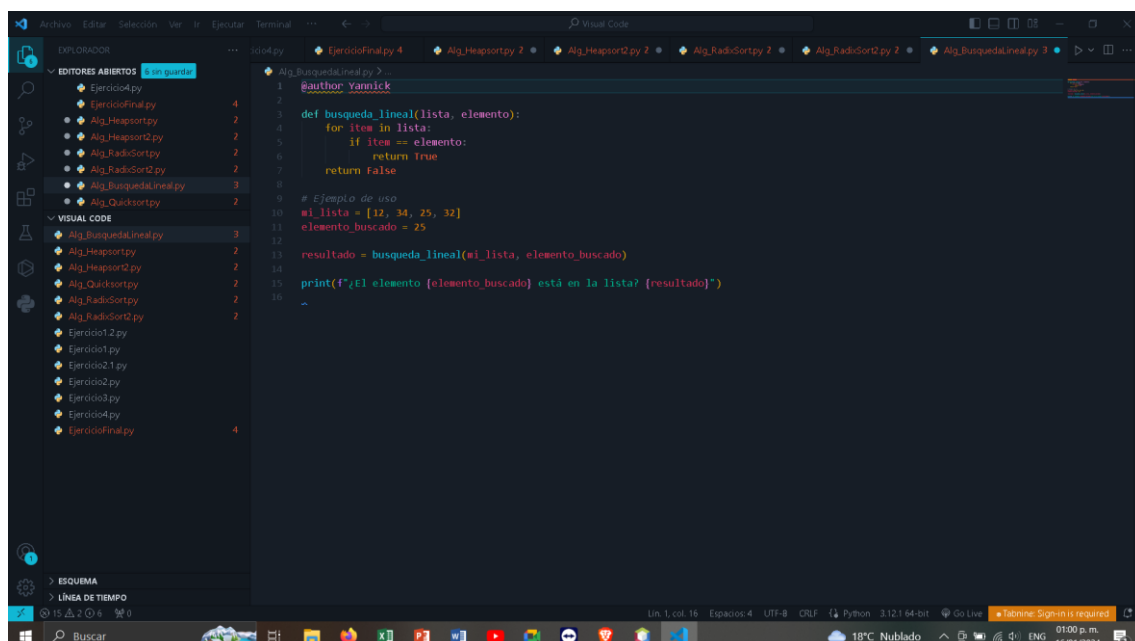
La función principal del código se llama "busqueda_lineal" y recibe dos parámetros: la lista en la que se realizará la búsqueda y el elemento que se desea encontrar.

El algoritmo utiliza un bucle "for" para recorrer cada elemento de la lista, comparándolo con el elemento buscado. Si encuentra una coincidencia, devuelve el valor "True"; de lo contrario, devuelve "False".

Se muestra el ejemplo de búsqueda del número 25 en una lista predefinida, y se comenta que el programa es sencillo y puede ejecutarse en menos de 2 minutos.

Se resalta la posibilidad de cambiar el elemento a buscar y la lista, demostrando que el programa funciona correctamente al devolver "True" o "False" según la presencia del elemento en la lista.

Aquí tienes un ejemplo sencillo en Python del algoritmo de búsqueda lineal basado en la transcripción proporcionada:



```
1 #author: Yannick
2
3 def busqueda_lineal(lista, elemento):
4     for item in lista:
5         if item == elemento:
6             return True
7     return False
8
9 # Ejemplo de uso
10 mi_lista = [12, 34, 25, 32]
11 elemento_buscado = 25
12
13 resultado = busqueda_lineal(mi_lista, elemento_buscado)
14
15 print(f"El elemento {elemento_buscado} está en la lista? {resultado}")
16
```

En este ejemplo, la función busqueda_lineal recibe una lista y un elemento como parámetros y utiliza un bucle for para recorrer la lista. Si encuentra una coincidencia, devuelve True; de lo contrario, devuelve False. Luego, se prueba la función con una lista predefinida y un elemento específico, y se imprime el resultado.

7. Algoritmo de Búsqueda Binaria

La búsqueda binaria es un algoritmo más rápido que la búsqueda lineal, ya que divide la lista en mitades en cada comparación, permitiendo una búsqueda eficiente.

La principal ventaja de la búsqueda binaria es su velocidad, pero requiere que la lista esté previamente ordenada de manera ascendente o descendente.

El algoritmo inicia verificando si la lista tiene elementos; si no los tiene, la búsqueda termina. En caso contrario, se compara el elemento buscado con el valor en la mitad de la lista.

Si el elemento buscado es menor al valor en la mitad, se busca en la sublista de elementos menores; si es mayor, se busca en la sublista de elementos mayores.

Se repite el proceso de búsqueda binaria en la sublista correspondiente hasta encontrar el número buscado o determinar que no está en la lista.

7. Algoritmo de Búsqueda Binaria en Python

Implementación del Algoritmo de Búsqueda Binaria en Python:

Creación de una función llamada "binaria" que recibe una lista ordenada y un valor a buscar.

Verificación de si la lista tiene elementos, devolviendo un mensaje de "número no encontrado" si está vacía.

Determinación del Elemento en la Mitad de la Lista:

Identificación del número en la posición media de la lista mediante la operación de división entera.

Comparación del número encontrado con el valor buscado; si son iguales, se devuelve un mensaje de éxito.

Manejo de Casos en Función de la Comparación:

Si el valor buscado es menor que el número en la mitad de la lista, se realiza una llamada recursiva a la función binaria con la lista hasta la mitad.

Si el valor buscado es mayor, la llamada recursiva se realiza con la lista desde la mitad más uno hasta el final.

Recursividad y Segmentación de la Lista:

Utilización de la recursividad para dividir la búsqueda en segmentos más pequeños de la lista.

Segmentación de la lista según si el valor buscado es menor o mayor que el número en la mitad.

Prueba del Algoritmo con Ejemplos:

Creación de una lista de prueba y llamada a la función binaria para buscar el número 2.

Ejemplo de la ejecución con un número (por ejemplo, 8) que no se encuentra en la lista, mostrando el mensaje de "número no encontrado" en la terminal.

Aquí tienes un ejemplo simple de cómo utilizar la función binaria implementada para buscar un número en una lista en Python:


```
1 @author: Yannick
2
3 def binaria(lista, x):
4     if len(lista) <= 0:
5         return "Número no encontrado"
6
7     m = len(lista) // 2
8     if lista[m] == x:
9         return f"¡Número {x} encontrado en la posición {m}!"
10    elif x < lista[m]:
11        return binaria(lista[:m], x)
12    else:
13        return binaria(lista[m+1:], x)
14
15 # Ejemplo de uso
16 mi_lista = [1, 2, 3, 4, 5, 6, 7]
17 numero_buscado = 2
18
19 resultado = binaria(mi_lista, numero_buscado)
20 print(resultado)
21
22 -
```

En este ejemplo, la función binaria se llama con la lista `mi_lista` y el número a buscar `numero_buscado`. La función devuelve un mensaje indicando si el número se encontró o no, y en qué posición. Ten en cuenta que esta implementación asume que la lista está ordenada de manera ascendente.

8. Búsqueda Hash

Definición de Búsqueda Hash: La búsqueda Hash se basa en el uso de funciones criptográficas llamadas "hash" que convierten datos de entrada en una salida de longitud fija. Este resumen se conoce como el resumen de la información de entrada y siempre es el mismo para la misma entrada.

Construcción de una Tabla Hash: Para implementar la búsqueda Hash, se construye una tabla hash que asocia claves con valores. Los datos de entrada se pasan por la función hash, y el resultado determina la posición en la tabla donde se almacenarán.

Principio del Palomar: Si la tabla hash tiene más elementos que casillas, se puede aplicar el principio del palomar, donde las casillas deben albergar más de un elemento. Esto implica guardar una lista de elementos en cada casilla y realizar una búsqueda lineal si es necesario.

Operación Módulo: Una función simple y común en la búsqueda Hash es la operación módulo, que devuelve valores entre 0 y m (tamaño de la tabla). Esta operación se utiliza para determinar la posición de los elementos en la tabla.

Ejemplo Práctico: Se presenta un ejemplo práctico de cómo llenar una tabla hash con números utilizando la operación módulo y luego realizar búsquedas en la tabla utilizando la función hash. Se destaca la eficiencia de la búsqueda cuando se tiene un solo elemento en la casilla de la tabla.

8. Búsqueda Hash en Python

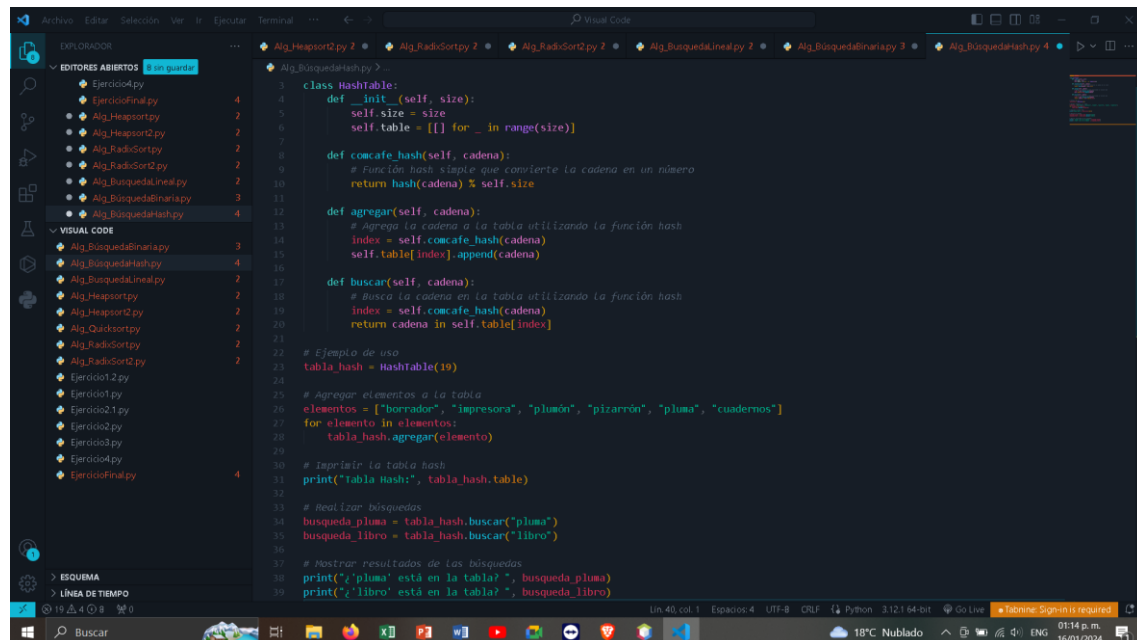
El video trata sobre la implementación de la búsqueda hash en Python, centrándose en la construcción de una tabla de cadenas para realizar búsquedas con cadenas.

Se utiliza una función hash llamada "comcafé" para convertir las cadenas en números y obtener un valor hash. Posteriormente, se multiplica este valor por el tamaño de la tabla hash para obtener la posición en la que se almacenará la cadena.

Se define la función "agregar" que recibe la cadena, la tabla y el tamaño de la tabla. Esta función utiliza la función hash para determinar la posición y agrega la cadena a la tabla.

También se describe la función para buscar en la tabla, que recibe la cadena, la tabla y el tamaño de la tabla. Realiza una búsqueda lineal en la lista de la posición obtenida por la función hash para determinar si la cadena está presente.

Se muestra una demostración práctica, creando una tabla hash con un tamaño de 19 y agregando algunas cadenas. Luego, se realiza una búsqueda de las palabras "pluma" y "libro", mostrando si están presentes en la tabla y el resultado de la búsqueda.



```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def comcafe_hash(self, cadena):
        # Función hash simple que convierte la cadena en un número
        return hash(cadena) % self.size

    def agregar(self, cadena):
        # Agrega la cadena a la tabla utilizando la función hash
        index = self.comcafe_hash(cadena)
        self.table[index].append(cadena)

    def buscar(self, cadena):
        # Busca la cadena en la tabla utilizando la función hash
        index = self.comcafe_hash(cadena)
        return cadena in self.table[index]

# Ejemplo de uso
tabla_hash = HashTable(19)

# Agregar elementos a la tabla
elementos = ["horrorador", "impresora", "plumón", "pizarrón", "pluma", "cuadernos"]
for elemento in elementos:
    tabla_hash.agregar(elemento)

# Imprimir la tabla hash
print("Tabla Hash:", tabla_hash.table)

# Realizar búsquedas
busqueda_pluma = tabla_hash.buscar("pluma")
busqueda_libro = tabla_hash.buscar("libro")

# Mostrar resultados de las búsquedas
print("¿'pluma' está en la tabla?", busqueda_pluma)
print("¿'libro' está en la tabla?", busqueda_libro)
```

Este ejemplo crea una tabla hash con un tamaño de 19, agrega algunas cadenas y luego realiza búsquedas para determinar si ciertas palabras están presentes en la tabla hash. La función comcafe_hash es una implementación simple de una función hash que utiliza la función incorporada hash de Python.

9. Introducción a Gráficas o grafos

Definición de gráfica o grafo: Una gráfica o grafo es un conjunto de vértices que se relacionan entre sí mediante un conjunto de aristas. Matemáticamente, se representa como $g = v$, donde g es la gráfica, v es el conjunto de vértices y e es el conjunto de aristas.

Origen de la teoría de grafos: La teoría de grafos se originó en 1736 gracias a Leonard Euler, matemático que resolvió el problema de los siete puentes de conexión entre cuatro ciudades. Este problema llevó a la conclusión de que un punto debe tener un número par de aristas para regresar al punto de partida.

Importancia de las gráficas en computación: En computación, las gráficas son esenciales para representar mapas y procesos. Son utilizadas en aplicaciones de navegación como Google Maps y Waze. Permiten representar sucesiones de tareas y visualizar relaciones entre nodos y vértices.

Elementos de una gráfica: Además de nodos y vértices, las gráficas pueden tener nombres o etiquetas para identificar cada nodo, y las aristas pueden tener pesos que representan información como tiempo, distancia o procesamiento.

Tipos de gráficas: Hay gráficas dirigidas, donde se indica la dirección del recorrido con flechas, y no dirigidas, donde no hay restricciones en el recorrido. También se clasifican según la presencia o ausencia de pesos en las aristas. El tamaño de una gráfica se determina por el número de aristas, mientras que el orden se determina por el número de vértices.

9. Introducción a Gráficas o grafos en Python

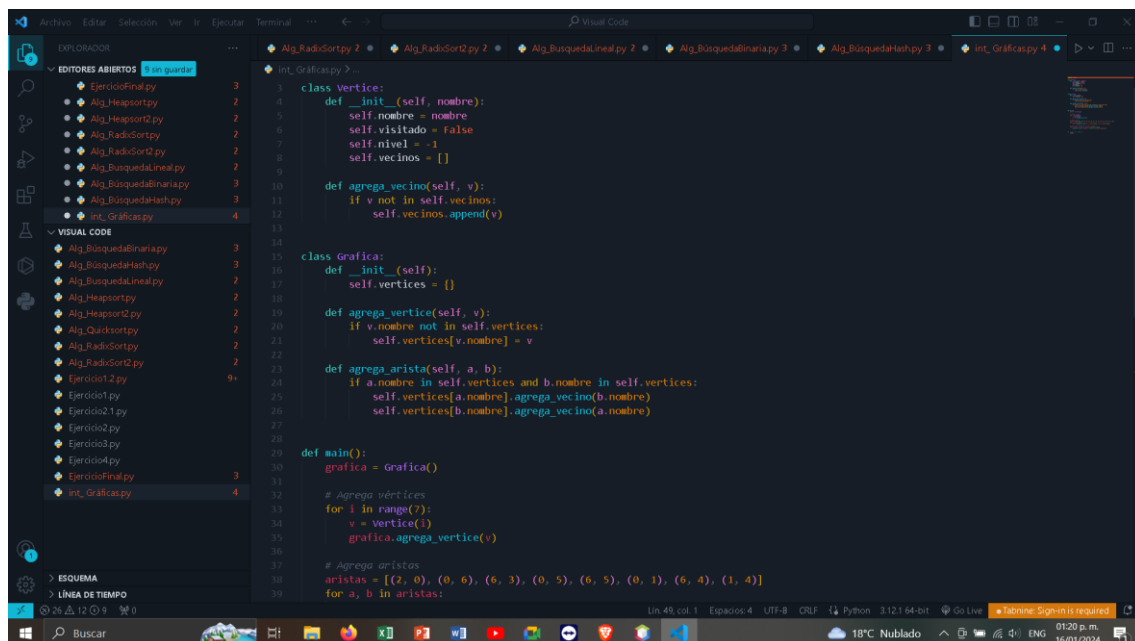
La transcripción trata sobre la introducción a gráficas o grafos en Python, donde se crea una implementación de clases para representar vértices y grafos, con el objetivo de utilizarlos en algoritmos posteriores.

Se inicia creando una clase llamada "Vértice" con un constructor que define atributos como el nombre del vértice, la variable "visitado" y "nivel", necesarios para futuros algoritmos. También se establece una lista de vecinos para representar las conexiones entre vértices.

Posteriormente, se introduce una segunda clase llamada "Gráfica" que representa el conjunto completo de vértices. Se utiliza un diccionario llamado "vértices" para almacenar los vértices de la gráfica, donde la llave es el identificador del vértice y el valor es el objeto vértice en sí.

La clase "Gráfica" tiene métodos para agregar vértices y crear aristas entre ellos, formando una gráfica no dirigida. Se valida que los vértices a conectar existan en el diccionario antes de crear la conexión.

Finalmente, se realiza una prueba creando una gráfica con vértices del 0 al 6 y estableciendo conexiones entre ellos. Se imprime la información de cada vértice y sus vecinos para verificar la correcta creación de la gráfica en Python.



```
class Vertice:
    def __init__(self, nombre):
        self.nombre = nombre
        self.visitado = False
        self.nivel = -1
        self.vecinos = []

    def agrega_vecino(self, v):
        if v not in self.vecinos:
            self.vecinos.append(v)

class Grafica:
    def __init__(self):
        self.vertices = {}

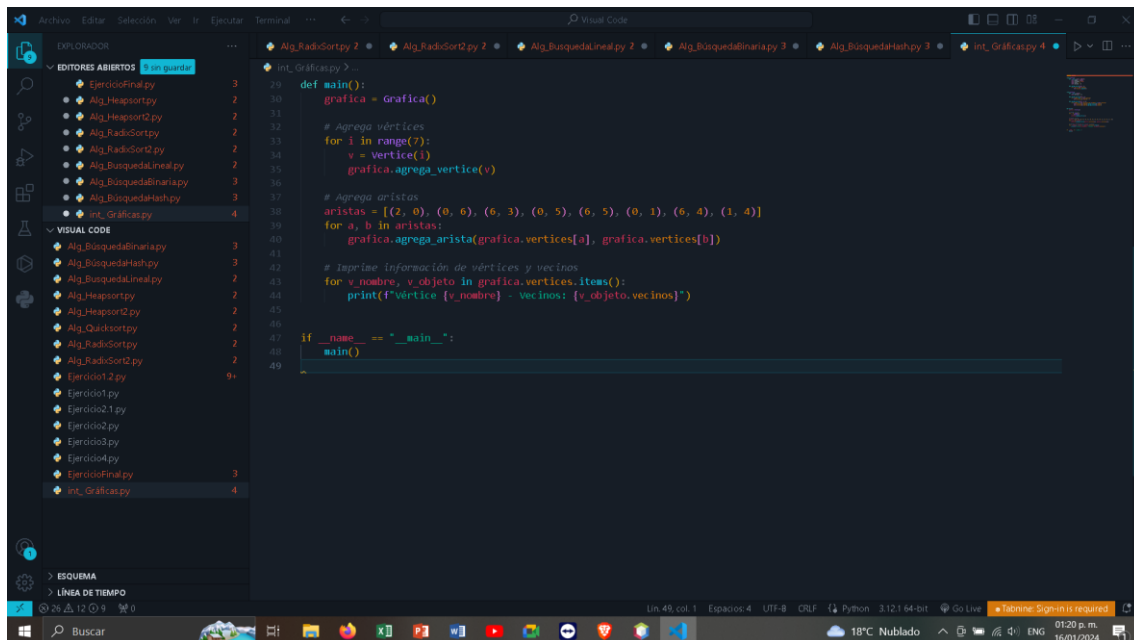
    def agrega_vertice(self, v):
        if v.nombre not in self.vertices:
            self.vertices[v.nombre] = v

    def agrega_arista(self, a, b):
        if a.nombre in self.vertices and b.nombre in self.vertices:
            self.vertices[a.nombre].agrega_vecino(b.nombre)
            self.vertices[b.nombre].agrega_vecino(a.nombre)

def main():
    grafica = Grafica()

    # Agrega vértices
    for i in range(7):
        v = Vertice(i)
        grafica.agrega_vertice(v)

    # Agrega aristas
    aristas = [(0, 0), (0, 6), (6, 3), (0, 5), (6, 5), (0, 1), (6, 4), (1, 4)]
    for a, b in aristas:
```



```
def main():
    grafica = Grafica()

    # Agrega vértices
    for i in range(7):
        v = Vertice(i)
        grafica.agrega_vertice(v)

    # Agrega aristas
    aristas = [(2, 0), (0, 6), (6, 3), (0, 5), (6, 5), (0, 1), (6, 4), (1, 4)]
    for a, b in aristas:
        grafica.agrega_arista(grafica.vertices[a], grafica.vertices[b])

    # Imprime información de vértices y vecinos
    for v_nombre, v_objeto in grafica.vertices.items():
        print(f"Vértice {v_nombre} - Vecinos: {v_objeto.vecinos}")

if __name__ == "__main__":
    main()
```

Este código crea una gráfica con vértices del 0 al 6 y establece las conexiones definidas en las aristas. Luego, imprime la información de cada vértice y sus vecinos. Asegúrate de ejecutarlo en un entorno que admita Python.

10. Algoritmo de Búsqueda por expansión (BFS)

Introducción a Búsquedas Bibliográficas: El video aborda el tema de las búsquedas bibliográficas, destacando su aplicación en la creación de mecanismos autónomos de navegación, como en aplicaciones como Uber y Waze, que buscan la ruta más corta entre dos puntos.

Algoritmo de Búsqueda por Expansión (BFS): Se presenta el algoritmo de Búsqueda por Expansión (también conocido como Búsqueda por Anchura o BCS) como el primero de una serie de algoritmos para realizar búsquedas en gráficas. Este algoritmo se caracteriza por explorar nodos vecinos en capas, buscando caminos más cortos desde un nodo inicial.

Descripción del Algoritmo BFS: Se explica que el algoritmo comienza desde un nodo raíz, explorando todos los nodos vecinos de manera secuencial por capas, priorizando aquellos a una distancia menor del vértice inicial. La búsqueda por expansión busca encontrar caminos más cortos en la gráfica.

Ejemplo Práctico con una Gráfica: Se realiza un ejercicio práctico utilizando una gráfica específica. Se detalla el proceso paso a paso, marcando niveles, utilizando una cola de prioridades, y mostrando cómo el algoritmo explora la gráfica, marcando nodos como visitados y construyendo una lista final.

Conclusión del Ejercicio: Se concluye mostrando la lista final que indica cómo se realizó la búsqueda por expansión en la gráfica, destacando cómo algunos nodos quedaron sin niveles y sin visitar, indicando que no están conectados a la porción de la gráfica recorrida.

10. Algoritmo de Búsqueda por expansión (BFS) en Python

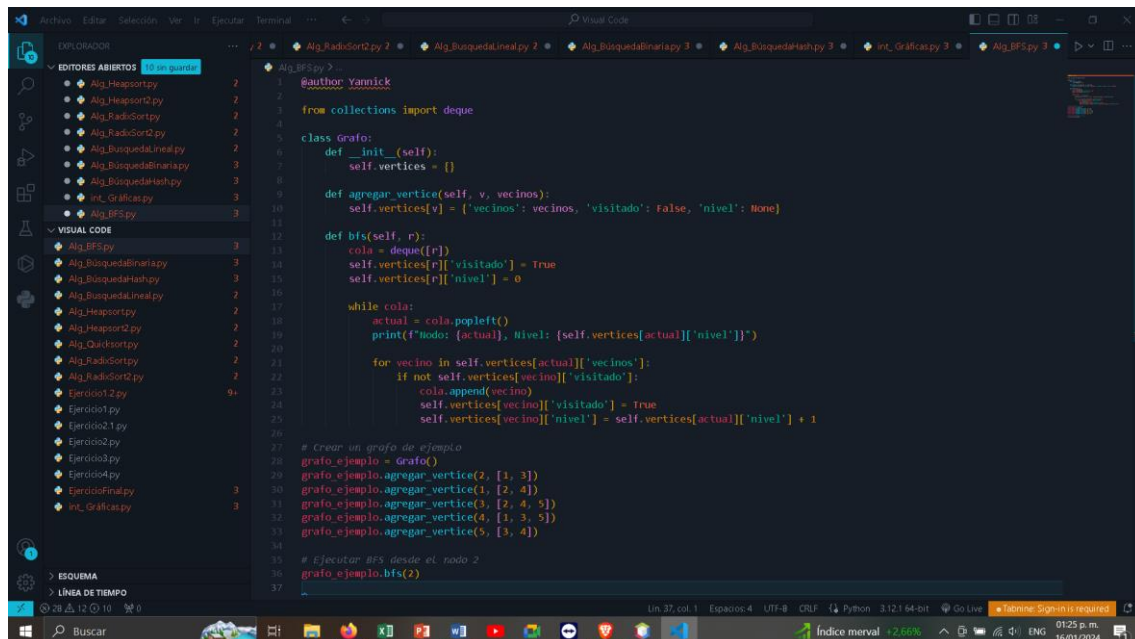
En el video, se programa un algoritmo de Búsqueda por Expansión (BFS) en Python, centrado en la clase gráfica.

Se utiliza una función llamada fs que recibe un nodo inicial (r) para iniciar el recorrido.

El algoritmo marca el nodo inicial como visitado, lo agrega a una cola y asigna su nivel como 0.

Luego, se recorren los vecinos del nodo actual, agregándolos a la cola si no han sido visitados, marcándolos como visitados y asignándoles un nivel.

El video muestra la ejecución del algoritmo, imprimiendo los nodos visitados y sus niveles en la terminal, con el ejemplo de inicio en el nodo 2.



```
1 #author: Yannick
2
3 from collections import deque
4
5 class Grafo:
6     def __init__(self):
7         self.vertices = {}
8
9     def agregar_vertice(self, v, vecinos):
10        self.vertices[v] = {'vecinos': vecinos, 'visitado': False, 'nivel': None}
11
12    def bfs(self, r):
13        cola = deque([r])
14        self.vertices[r]['visitado'] = True
15        self.vertices[r]['nivel'] = 0
16
17        while cola:
18            actual = cola.popleft()
19            print(f"Nodo: {actual}, Nivel: {self.vertices[actual]['nivel']}")
20
21            for vecino in self.vertices[actual]['vecinos']:
22                if not self.vertices[vecino]['visitado']:
23                    cola.append(vecino)
24                    self.vertices[vecino]['visitado'] = True
25                    self.vertices[vecino]['nivel'] = self.vertices[actual]['nivel'] + 1
26
27 # Crear un grafo de ejemplo
28 grafo_ejemplo = Grafo()
29 grafo_ejemplo.agregar_vertice(2, [1, 3])
30 grafo_ejemplo.agregar_vertice(1, [2, 4])
31 grafo_ejemplo.agregar_vertice(3, [2, 4, 5])
32 grafo_ejemplo.agregar_vertice(4, [1, 3, 5])
33 grafo_ejemplo.agregar_vertice(5, [3, 4])
34
35 # Ejecutar BFS desde el nodo 2
36 grafo_ejemplo.bfs(2)
```

Este ejemplo crea un grafo simple con nodos y sus conexiones, luego realiza la Búsqueda por Expansión (BFS) desde el nodo 2, imprimiendo los nodos visitados y sus niveles. Ten en cuenta que el resultado puede variar según la configuración específica del grafo.

11. Algoritmo de Búsqueda en profundidad (DFS)

El algoritmo de búsqueda en profundidad (DFS) se diferencia de la búsqueda por anchura al no recorrer nodos por capas. Busca la máxima expansión, yendo más profundo para llegar de un nodo a otro.

En DFS, se inicia en un nodo, se elige un vecino, y se recorre la arista para llegar a él. Este proceso se repite hasta que no hay más nodos por visitar, momento en el cual se retrocede recursivamente al nodo anterior.

El recorrido se ilustra con un ejemplo, iniciando en el nodo 1, pasando por nodos como 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, y 12. El algoritmo visita los nodos en profundidad antes de retroceder.

A diferencia de la búsqueda por anchura, DFS no es óptimo para encontrar caminos con la menor cantidad de aristas a recorrer. Su enfoque es explorar en profundidad antes de retroceder.

El algoritmo DFS se implementa marcando nodos como visitados y explorando secuencialmente a sus vecinos. La recursividad juega un papel clave en el proceso de retroceso cuando no hay más nodos para visitar desde un punto específico.

11. Algoritmo de Búsqueda en profundidad (DFS) en Python

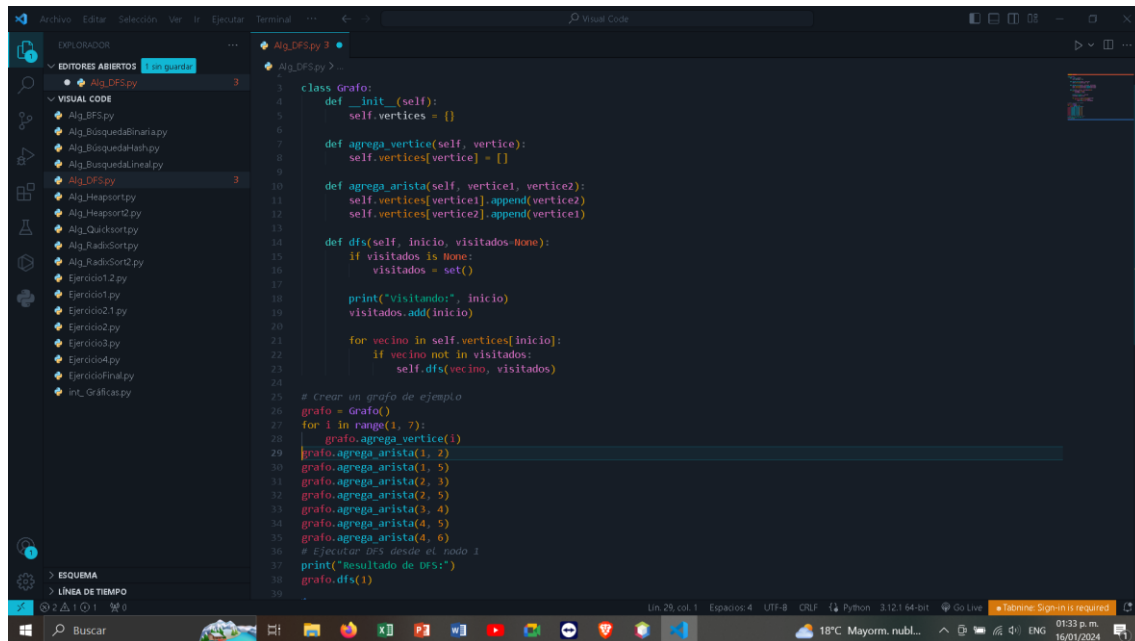
Implementación de DFS en Python: El artículo aborda la implementación del algoritmo de Búsqueda en Profundidad (DFS) en Python. Se sugiere realizar la implementación en el mismo archivo que contiene las clases para gráficos en Python.

Validación de inicio: Antes de comenzar el recorrido, se asegura que el nodo de inicio esté presente en el conjunto de vértices de la gráfica. Si no está presente, se realiza la inserción.

Marca de nodos visitados: El algoritmo marca inicialmente el nodo de inicio como visitado y luego procede a visitar cada vecino del nodo actual. Se verifica si un nodo no ha sido visitado anteriormente antes de explorarlo.

Establecimiento de padre: Se asigna al nodo actual como el padre del vecino que se va a visitar, y se utiliza un atributo llamado "padre" en la clase de vértice para llevar un registro de los padres de los nodos.

Llamada recursiva y prueba del algoritmo: Se realiza una llamada recursiva a la función DFS para seguir explorando la gráfica desde el vecino actual. Finalmente, se prueba el algoritmo con un conjunto de nodos y aristas, mostrando el resultado del recorrido en la terminal.



```
1 class Grafo:
2     def __init__(self):
3         self.vertices = {}
4
5     def agrega_vertice(self, vertice):
6         self.vertices[vertice] = {}
7
8     def agrega_arista(self, vertice1, vertice2):
9         self.vertices[vertice1].append(vertice2)
10        self.vertices[vertice2].append(vertice1)
11
12    def dfs(self, inicio, visitados=None):
13        if visitados is None:
14            visitados = set()
15
16        print("Visitando:", inicio)
17        visitados.add(inicio)
18
19        for vecino in self.vertices[inicio]:
20            if vecino not in visitados:
21                self.dfs(vecino, visitados)
22
23    # Crear un grafo de ejemplo
24    grafo = Grafo()
25    for i in range(1, 7):
26        grafo.agrega_vertice(i)
27    grafo.agrega_arista(1, 2)
28    grafo.agrega_arista(1, 5)
29    grafo.agrega_arista(2, 3)
30    grafo.agrega_arista(2, 5)
31    grafo.agrega_arista(3, 4)
32    grafo.agrega_arista(4, 5)
33    grafo.agrega_arista(4, 6)
34    # Ejecutar DFS desde el nodo 1
35    print("Resultado de DFS:")
36    grafo.dfs(1)
```

En este ejemplo, se crea una clase Grafo que tiene métodos para agregar vértices y aristas. La función dfs realiza la búsqueda en profundidad a partir de un nodo de inicio. Puedes observar cómo se visita cada nodo y se imprime en el orden en que se descubre durante el recorrido DFS.

12. Algoritmo de Bellman-Ford

El tema principal es el "Algoritmo de Bellman-Ford" en el contexto de un curso de algoritmos de búsqueda y ordenamiento en Python.

El algoritmo tiene como objetivo determinar la ruta más corta desde un nodo inicial hacia otros nodos en una gráfica dirigida con pesos, permitiendo el uso de pesos negativos en las aristas.

Fue creado por los matemáticos Richard Bellman y Lester Ford, y se diferencia de otros algoritmos al permitir trabajar con pesos negativos en las aristas.

El algoritmo consiste en asignar distancias y predecesores a los nodos, luego revisar aristas para actualizar distancias si se encuentra una ruta más corta, y repetir este proceso varias veces.

Se destaca la simplicidad del algoritmo a través de un ejercicio práctico donde se asignan distancias y predecesores, y se actualizan basándose en el peso de las aristas en una gráfica dirigida.

12. Algoritmo de Bellman-Ford en Python

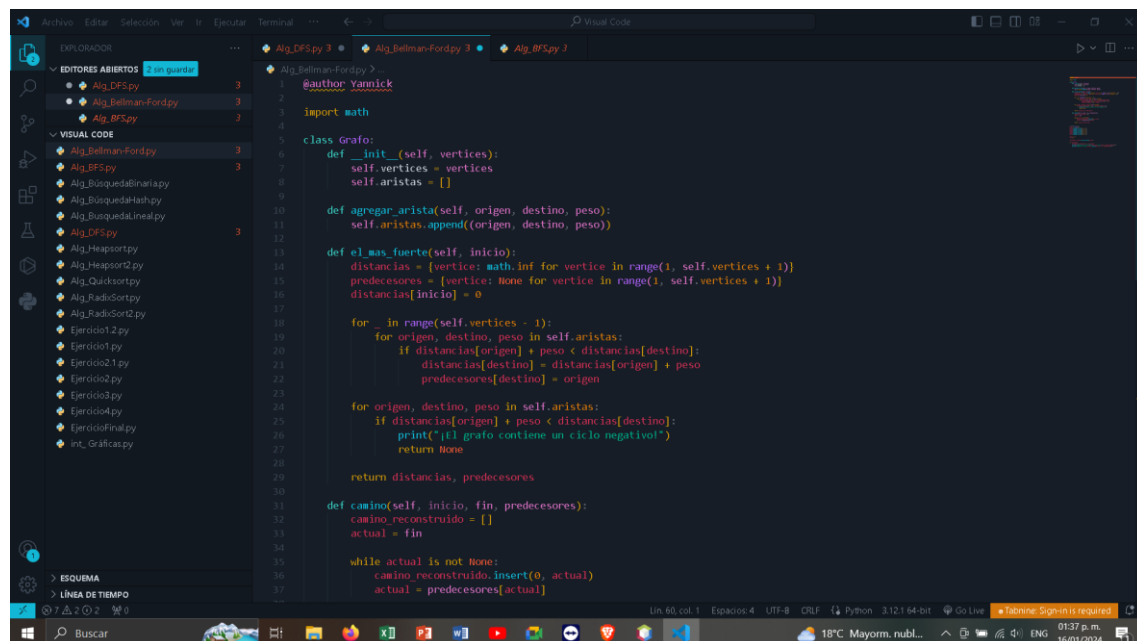
El video trata sobre la implementación del algoritmo de Bellman-Ford en Python.

Se utiliza un archivo de gráficas previamente manejado en videos anteriores para programar el algoritmo.

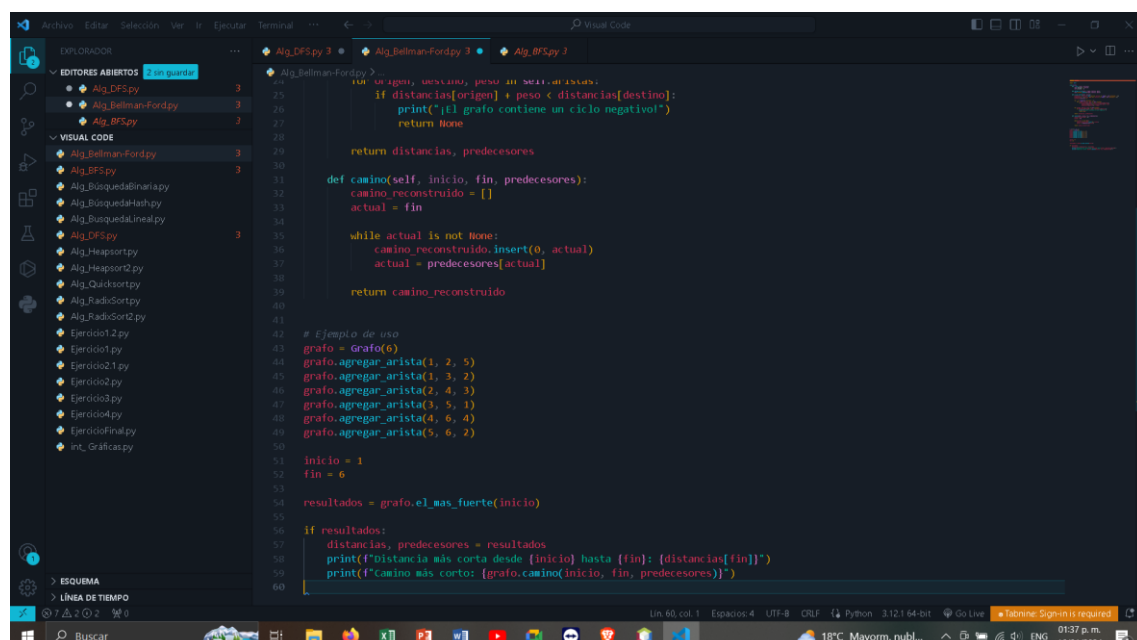
La función principal del algoritmo se llama "el_mas_fuerte" y recibe el parámetro "self" y un nuevo nodo inicial para comenzar el recorrido.

El código del algoritmo incluye pasos como asignar distancias, costos infinitos a nodos no iniciales, predecesores nulos, y actualizar costos según las aristas.

Se menciona la necesidad de verificar la existencia de ciclos negativos en la gráfica y se describe una función adicional llamada "camino" para reconstruir el camino entre dos nodos dados.



```
1  @author Yamick
2
3  import math
4
5  class Grafo:
6      def __init__(self, vertices):
7          self.vertices = vertices
8          self.aristas = []
9
10         def agregar_arista(self, origen, destino, peso):
11             self.aristas.append((origen, destino, peso))
12
13         def el_mas_fuerte(self, inicio):
14             distancias = {vertice: math.inf for vertice in range(1, self.vertices + 1)}
15             predecesores = {vertice: None for vertice in range(1, self.vertices + 1)}
16             distancias[inicio] = 0
17
18             for _ in range(self.vertices - 1):
19                 for origen, destino, peso in self.aristas:
20                     if distancias[origen] + peso < distancias[destino]:
21                         distancias[destino] = distancias[origen] + peso
22                         predecesores[destino] = origen
23
24                 for origen, destino, peso in self.aristas:
25                     if distancias[origen] + peso < distancias[destino]:
26                         print(f"¡El grafo contiene un ciclo negativo!")
27                         return None
28
29             return distancias, predecesores
30
31         def camino(self, inicio, fin, predecesores):
32             camino_reconstruido = []
33             actual = fin
34
35             while actual is not None:
36                 camino_reconstruido.insert(0, actual)
37                 actual = predecesores[actual]
```



```
38
39         # Ejemplo de uso
40         grafo = Grafo(6)
41         grafo.agregar_arista(1, 2, 5)
42         grafo.agregar_arista(1, 3, 2)
43         grafo.agregar_arista(2, 4, 3)
44         grafo.agregar_arista(3, 5, 1)
45         grafo.agregar_arista(4, 6, 4)
46         grafo.agregar_arista(5, 6, 2)
47
48         inicio = 1
49         fin = 6
50
51         resultados = grafo.el_mas_fuerte(inicio)
52
53         if resultados:
54             distancias, predecesores = resultados
55             print(f"Distancia más corta desde {inicio} hasta {fin}: {distancias[fin]}")
56             print(f"Camino más corto: {grafo.camino(inicio, fin, predecesores)}")
57
58         60
```

Este código crea un grafo dirigido con aristas y pesos, luego aplica el algoritmo de Bellman-Ford para encontrar el camino más corto desde un nodo inicial hasta todos los demás nodos. Finalmente, muestra la distancia más corta y el camino más corto desde el nodo inicial hasta un nodo específico (en este caso, desde el nodo 1 hasta el nodo 6).

13. Algoritmo de Dijkstra

Algoritmo de Dijkstra: Presentado como el penúltimo algoritmo en el último día del curso.

Objetivo: Busca el camino más corto en una gráfica con pesos en cada arista, desarrollado por Edsger Dijkstra en 1959.

Funcionamiento: Similar a Bellman-Ford, pero no funciona en gráficas con aristas que contienen pesos negativos. Utiliza una cola de prioridad para el procesamiento.

Ejemplo Práctico: Aplicación del algoritmo desde el nodo 1 para encontrar los caminos más cortos hacia todos los demás nodos en una gráfica específica.

Cola de Prioridad: Diferencia clave con Bellman-Ford, utiliza una cola de prioridad para manejar la selección de nodos de manera más eficiente, asignando distancias tentativas y actualizándolas según el análisis de vecinos.

13. Algoritmo de Dijkstra en Python

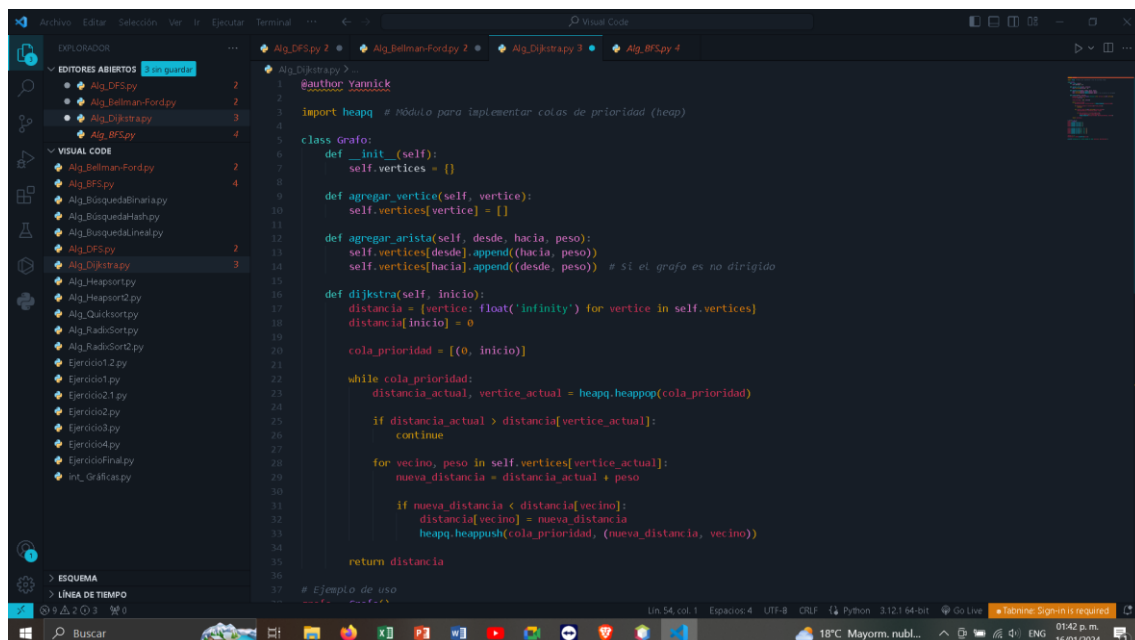
El tutorial aborda la implementación del algoritmo de Dijkstra en Python.

Se realizan modificaciones en las clases relacionadas con la creación de un grafo, incluyendo la adición de un atributo de distancia a la clase vértice.

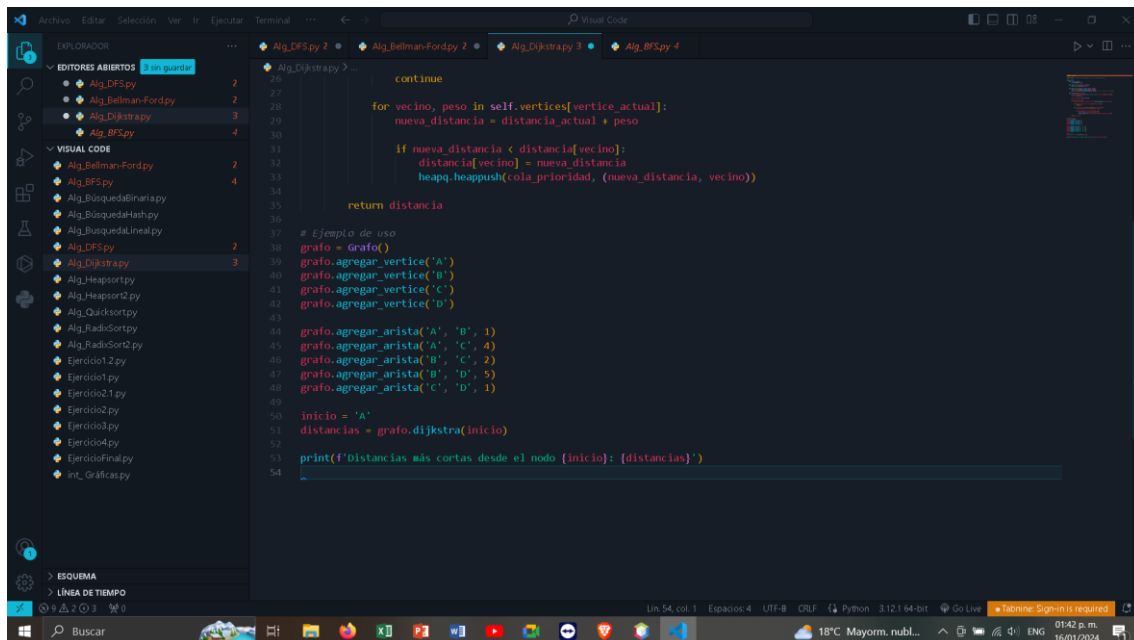
Se introducen cambios en los métodos para agregar vecinos y aristas en las clases relacionadas con el grafo.

Se detalla el proceso de inicialización y asignación de distancias tentativas a los nodos antes de comenzar el algoritmo de Dijkstra.

El código final incluye funciones de apoyo como "imprimir_grafica" y "camino", además de la ejecución y visualización del resultado del algoritmo de Dijkstra para encontrar la ruta más corta entre dos nodos en un grafo dado.



```
1 #author: Yannick
2
3 import heapq # Módulo para implementar colas de prioridad (heap)
4
5 class Grafo:
6     def __init__(self):
7         self.vertices = {}
8
9     def agregar_vertice(self, vertice):
10         self.vertices[vertice] = {}
11
12     def agregar_arista(self, desde, hacia, peso):
13         self.vertices[desde].append((hacia, peso))
14         self.vertices[hacia].append((desde, peso)) # Si el grafo es no dirigido
15
16     def dijkstra(self, inicio):
17         distancia = {vertice: float('infinity') for vertice in self.vertices}
18         distancia[inicio] = 0
19
20         cola_prioridad = [(0, inicio)]
21
22         while cola_prioridad:
23             distancia_actual, vertice_actual = heapq.heappop(cola_prioridad)
24
25             if distancia_actual > distancia[vertice_actual]:
26                 continue
27
28             for vecino, peso in self.vertices[vertice_actual]:
29                 nueva_distancia = distancia_actual + peso
30
31                 if nueva_distancia < distancia[vecino]:
32                     distancia[vecino] = nueva_distancia
33                     heapq.heappush(cola_prioridad, (nueva_distancia, vecino))
34
35         return distancia
36
37 # Ejemplo de uso
```

```
26         continue
27     for vecino, peso in self.vertices[vertice_actual]:
28         nueva_distancia = distancia_actual + peso
29
30         if nueva_distancia < distancia[vecino]:
31             distancia[vecino] = nueva_distancia
32             heapq.heappush(cola_prioridad, (nueva_distancia, vecino))
33
34     return distancia
35
36 # Ejemplo de uso
37 grafo = Grafo()
38 grafo.agregar_vertice('A')
39 grafo.agregar_vertice('B')
40 grafo.agregar_vertice('C')
41 grafo.agregar_vertice('D')
42 grafo.agregar_arista('A', 'B', 1)
43 grafo.agregar_arista('A', 'C', 4)
44 grafo.agregar_arista('B', 'C', 2)
45 grafo.agregar_arista('B', 'D', 5)
46 grafo.agregar_arista('C', 'D', 1)
47
48 inicio = 'A'
49 distancias = grafo.dijkstra(inicio)
50 print(f'Distancias más cortas desde el nodo {inicio}: {distancias}')
51
52
53
54
```

Este código crea un grafo, agrega vértices y aristas, y luego aplica el algoritmo de Dijkstra para encontrar las distancias más cortas desde un nodo inicial. Ten en cuenta que este es un ejemplo básico y puedes adaptarlo según tus necesidades específicas.

14. A*

Algoritmo A*:

A* es un algoritmo informado utilizado en búsqueda de rutas más cortas en grafos.

Se diferencia por ser heurístico, basándose en información adicional, como la heurística de tiempo estimado para llegar de un punto a otro.

La heurística proporciona una estimación del costo, como el tiempo, para moverse de un nodo a otro en una gráfica, considerando experiencias previas.

El algoritmo evalúa tanto la distancia real como la heurística para determinar la mejor ruta, considerando eventos actuales que podrían afectar el tiempo de viaje.

A través de un ejercicio práctico, se demuestra la aplicación del algoritmo A* en la búsqueda del camino más corto entre nodos en una gráfica, con la utilización de conjuntos y la actualización de distancias tentativas.

Inicialización del algoritmo:

Se inicia creando un conjunto abierto con el nodo inicial y asignando predecesores nulos y distancias tentativas.

La heurística se utiliza para asignar distancias tentativas iniciales, considerando experiencias previas en la gráfica.

El proceso de inicialización establece las bases para la ejecución del algoritmo A*.

Iteraciones del algoritmo:

Se realizan iteraciones mientras haya elementos en el conjunto abierto.

Se selecciona el nodo con la menor distancia tentativa para explorar sus vecinos.

Los vecinos se agregan al conjunto abierto si no han sido visitados, y se actualizan las distancias tentativas si se encuentra un camino más corto.

El proceso continúa hasta que se alcanza el nodo objetivo.

Reconstrucción del camino:

Una vez se llega al nodo objetivo, se reconstruye el camino desde el nodo final hasta el nodo inicial utilizando los predecesores.

Cada nodo se agrega a una lista, mostrando la ruta más corta en la gráfica.

Aplicación práctica:

Se demuestra la aplicación del algoritmo A* a través de un ejercicio específico que involucra la búsqueda del camino más corto en una gráfica.

Se ilustra el proceso de selección de nodos, actualización de distancias y reconstrucción del camino en el contexto del algoritmo A*.

14. A* en Python

Heurística y Costos Iniciales:

Se modifica la clase vértice para agregar nuevos campos como la heurística y costos (g y f) al inicializar la función de la estrella en Python.

Se asigna un valor de cero a la heurística si no se proporciona al crear la función vértices.

Inicialización de Nodos:

Se inicializan los costos (g y f) de los vértices, excepto el nodo inicial, con infinito.

Se asigna un padre nulo a cada vértice y se crea un conjunto abierto conteniendo solo el nodo inicial.

Algoritmo A*:

Se implementa el algoritmo A*, donde se realiza la búsqueda del camino más corto desde el nodo inicial al final.

Se evalúan y actualizan los costos de los nodos vecinos en el conjunto abierto.

Construcción del Camino:

Si el nodo actual es igual al objetivo, se reconstruye el camino utilizando una función previamente creada.

Se imprime el camino y su costo asociado.

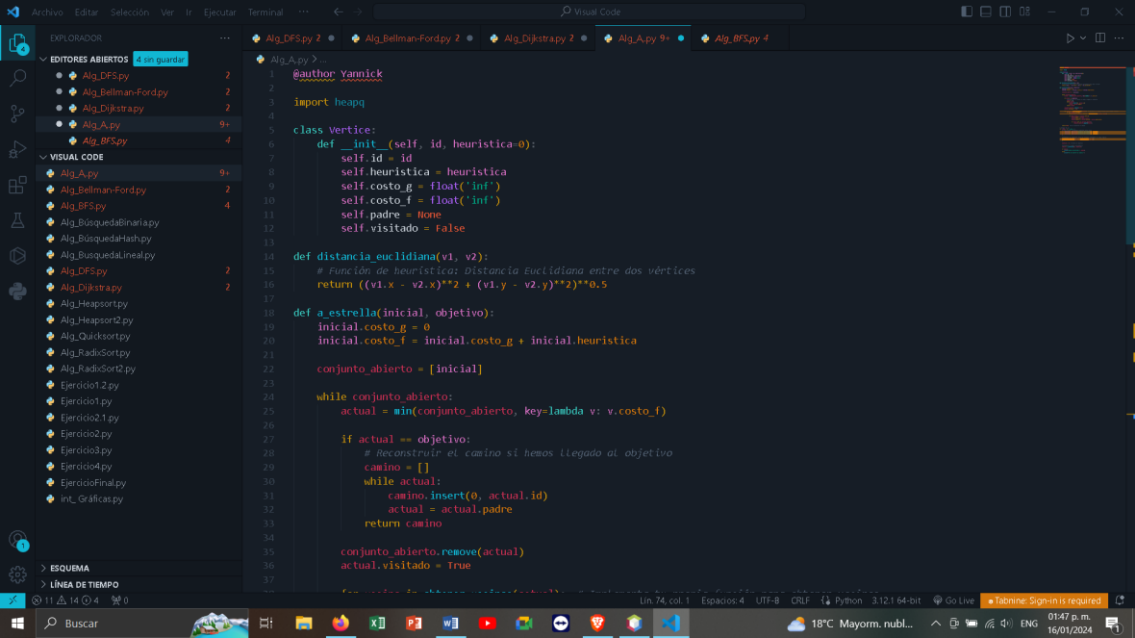
Función Mínimo H:

Se crea una función para encontrar el elemento con la menor distancia en el conjunto abierto basándose en la heurística.

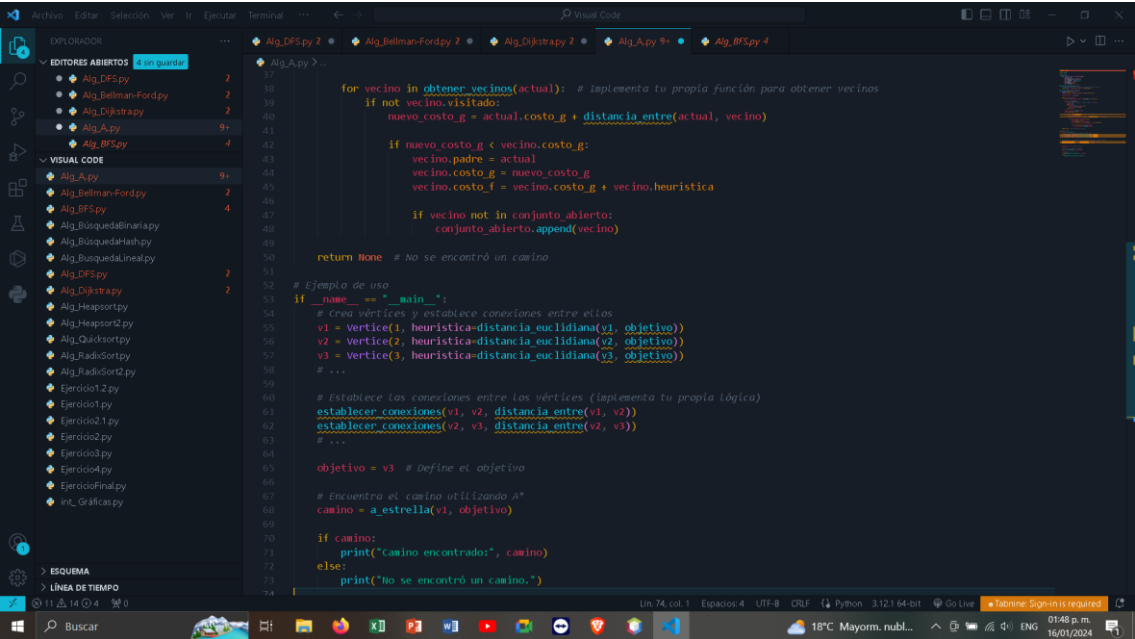
Se utiliza esta función en el algoritmo A* para seleccionar el siguiente nodo a evaluar.

Este resumen destaca los puntos clave relacionados con la implementación del algoritmo A* en Python, incluyendo la inicialización de nodos, el proceso de búsqueda, y la construcción del camino más corto.

Aquí tienes un ejemplo simple de cómo implementar el algoritmo A* en Python. Este ejemplo asume que ya tienes una implementación de la clase Vertice y sus funciones asociadas. Además, se utiliza una función de distancia Euclidiana como heurística. Puedes adaptar este código según tus necesidades y la implementación específica de tu clase Vertice.



```
1  @author: Yarnick
2
3  import heapq
4
5  class Vertice:
6      def __init__(self, id, heuristica=0):
7          self.id = id
8          self.heuristica = heuristica
9          self.costo_g = float('inf')
10         self.costo_f = float('inf')
11         self.padre = None
12         self.visitado = False
13
14     def distancia_euclidiana(v1, v2):
15         # Función de heurística: Distancia Euclidiana entre dos vértices
16         return ((v1.x - v2.x)**2 + (v1.y - v2.y)**2)**0.5
17
18     def a_estrella(inicial, objetivo):
19         inicial.costo_g = 0
20         inicial.costo_f = inicial.costo_g + inicial.heuristica
21
22         conjunto_abierto = [inicial]
23
24         while conjunto_abierto:
25             actual = min(conjunto_abierto, key=lambda v: v.costo_f)
26
27             if actual == objetivo:
28                 # Reconstruir el camino si hemos llegado al objetivo
29                 camino = []
30                 while actual:
31                     camino.insert(0, actual.id)
32                     actual = actual.padre
33                 return camino
34
35             conjunto_abierto.remove(actual)
36             actual.visitado = True
37
38         return None
```



```
37
38     def obtener_vecinos(actual): # Implementa tu propia función para obtener vecinos
39         if not actual.visitado:
40             nuevo_costo_g = actual.costo_g + distancia_entre(actual, vecino)
41
42             if nuevo_costo_g < vecino.costo_g:
43                 vecino.padre = actual
44                 vecino.costo_g = nuevo_costo_g
45                 vecino.costo_f = vecino.costo_g + vecino.heuristica
46
47             if vecino not in conjunto_abierto:
48                 conjunto_abierto.append(vecino)
49
50         return None # No se encontró un camino
51
52     # Ejemplo de uso
53     if __name__ == "__main__":
54         # Crea vértices y establece conexiones entre ellos
55         v1 = Vertice(1, heuristica=distancia_euclidiana(v1, objetivo))
56         v2 = Vertice(2, heuristica=distancia_euclidiana(v2, objetivo))
57         v3 = Vertice(3, heuristica=distancia_euclidiana(v3, objetivo))
58         # ...
59
60         # Establece las conexiones entre los vértices (implementa tu propia lógica)
61         establecer_conexiones(v1, v2, distancia_entre(v1, v2))
62         establecer_conexiones(v2, v3, distancia_entre(v2, v3))
63         # ...
64
65         objetivo = v3 # Define el objetivo
66
67         # Encuentra el camino utilizando A*
68         camino = a_estrella(v1, objetivo)
69
70         if camino:
71             print("Camino encontrado:", camino)
72         else:
73             print("No se encontró un camino.")
74
```

Este ejemplo es un punto de partida y puede requerir ajustes según tu implementación específica y necesidades del problema. Asegúrate de adaptar las funciones según la estructura de tu clase Vertice y las características de tu grafo.